Università degli Studi di Pisa
Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

XQuake: an XML-based Knowledge Discovery Environment

Andrea Romei

Supervisor
Prof. Franco Turini

November 2009

Addr: Largo B. Pontecorvo 3, I-56127 Pisa - Italy
Tel: +39-050-2212708 – Fax: +39-050-2212726 – E-mail: romei@di.unipi.it
Thesis Supervisor:
Franco Turini

External Reviewers:
Jean-François Boulicaut and Taneli Mielikäinen and Carlo Zaniolo

Internal Committee:
Antonio Cisternino and Giorgio Ghelli
Abstract

Data mining is the analysis of large volumes of data to find unsuspected relationships and to summarize the data in novel ways, that are both understandable and useful to the data owner. Nowadays, the rapid growth of semi-structured sources raises the need of designing and implementing environments for data mining out of XML data.

On the basis of the principles of the inductive database theory, this dissertation presents a flexible data mining system with capabilities of obtaining, maintaining, representing and querying induced, deduced and prior knowledge, stored inside native XML databases. In particular, it summarizes our three-years experience in the design and development of XQuake, a query language that extends XQuery to support mining primitives. Features of the language are an intuitive syntax, a good expressiveness, and the capability of dealing uniformly with data mining entities. A detail of its implementation and the evaluation of its performance are also given.
I would like to thank my supervisor Prof. Franco Turini, not only for his professional guidance throughout my studies, but also for his constant support and encouragement, my internal reviewers Prof. Giorgio Ghelli and Dr. Antonio Cisternino for their suggestions and feedbacks.

I am very grateful to my external reviewers, Prof. Jean-François Boulicaut, Prof. Taneli Mielikäinen and Prof. Carlo Zaniolo, for their comments, that have helped me to improve the overall quality of my research.

I would like to thank Valerio Grossi who shared with me this “adventure”, and also all the peoples who supported me thought these years.

Last but not least, special thanks to my family and to my girlfriend Donatella and she knows why.
Contents

Introduction 13
I.1 Why mining XML data? .................................. 13
I.2 How is XML data mined? .............................. 14
I.3 Thesis overview ..................................... 15

1 Inductive Databases 17
1.1 Background ......................................... 17
  1.1.1 What is KDD? .................................. 17
  1.1.2 What is an inductive database? ............... 18
1.2 Problem formulation .............................. 19
1.3 Requirements on inductive databases .......... 20
  1.3.1 Requirements on the KDD process .......... 21
  1.3.2 Requirements on KDD objects .............. 22
  1.3.3 Requirements on the system architecture .. 23
1.4 Query languages for inductive databases .... 24
  1.4.1 State-of-the-art: an overview .............. 25
  1.4.2 $\text{LDL}^{++}$: Logic Database Language ... 27
  1.4.3 ATLaS: Aggregate & Table Language and System 32
  1.4.4 KDDML: a KDD Markup Language ........... 36
1.5 Summary ......................................... 40

2 XML and related technologies 43
2.1 Introduction ....................................... 43
  2.1.1 XML querying .................................. 44
  2.1.2 XML processing ................................ 44
  2.1.3 XML storage .................................. 44
2.2 The XQuery language: an overview .......... 45
  2.2.1 Path expressions ............................. 46
  2.2.2 FL WOR expressions ......................... 46
  2.2.3 The XQuery Data Model .................... 48
2.3 The Predictive Model Markup Language .... 50
2.4 Summary ......................................... 51
3 Foundations of an XML-Based KDD Framework ............. 53
   3.1 The XQuery Data Model for Data Mining .................. 53
      3.1.1 Input data ........................................... 54
      3.1.2 Mining attributes .................................... 56
      3.1.3 Context knowledge ................................... 61
      3.1.4 Mining models ....................................... 62
   3.2 The XQuake philosophy .................................. 62
      3.2.1 Task and method specification ......................... 63
      3.2.2 Domain entities identification ....................... 64
      3.2.3 Exploitation of constraints and user preferences .... 70
      3.2.4 Output construction ................................ 70
      3.2.5 Summary ............................................. 70
   3.3 On the implementation of the mining tasks .............. 71
      3.3.1 Motivation .......................................... 73
      3.3.2 Run-through examples ................................ 74
      3.3.3 \texttt{wfor} for data mining ........................ 77
      3.3.4 External data structures .............................. 80
      3.3.5 Core semantics of \texttt{wfor} ......................... 81

4 XML preprocessing and classification ..................... 83
   4.1 Data discretization ..................................... 83
      4.1.1 Background .......................................... 83
      4.1.2 Task formulation .................................... 86
      4.1.3 Task implementation ................................ 89
   4.2 Data sampling .......................................... 91
      4.2.1 Background .......................................... 91
      4.2.2 Task formulation .................................... 92
      4.2.3 Task implementation ................................ 94
   4.3 Decision tree classification ............................ 96
      4.3.1 Background .......................................... 96
      4.3.2 Task formulation .................................... 101
      4.3.3 Task implementation ................................ 103
   4.4 Summary ................................................ 106

5 XML Frequent Itemsets Mining .......................... 107
   5.1 Frequent Itemsets Mining ............................... 108
      5.1.1 Background .......................................... 108
      5.1.2 Task formulation .................................... 113
      5.1.3 Task implementation ................................ 115
   5.2 Constraints-based FIM .................................. 120
      5.2.1 Background .......................................... 120
      5.2.2 Task formulation .................................... 124
      5.2.3 Task implementation ................................ 127
5.3 Post-processing enhancement ........................................ 130
   5.3.1 Frequent itemsets application ............................... 130
   5.3.2 Frequent itemsets filtering ................................. 132
5.4 Summary .......................................................... 133

6 System architecture and performance evaluation .................... 135
   6.1 The XQuake system prototype .................................. 136
      6.1.1 The database layer ....................................... 137
      6.1.2 The mining engine layer .................................. 138
      6.1.3 The intermediate layer .................................... 141
      6.1.4 The graphical user interface layer ....................... 144
   6.2 Performance evaluation ........................................ 145
      6.2.1 Testing the Apriori algorithm ............................. 146
      6.2.2 Application scenario: mining over the xmark database .. 149

7 Conclusion .......................................................... 161
   7.1 Final remarks ................................................... 161
   7.2 Future research directions .................................... 162
      7.2.1 System architecture optimizations ....................... 162
      7.2.2 Expressiveness on the prior knowledge ................... 163

A XML datasets ....................................................... 167

Bibliography ........................................................ 173
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The process of knowledge discovery in databases (from [63])</td>
<td>19</td>
</tr>
<tr>
<td>1.2</td>
<td>Requirements on IDBs</td>
<td>21</td>
</tr>
<tr>
<td>1.3</td>
<td>A possible representation of frequent itemsets and association rules in ATLaS. Trie representing the frequent itemsets (a). The DECODE table (b), TRIE table (c) and RDA table (d).</td>
<td>35</td>
</tr>
<tr>
<td>2.1</td>
<td>Basic components of the XQuery Data Model (from [119])</td>
<td>49</td>
</tr>
<tr>
<td>2.2</td>
<td>A sample of PMML document containing a set of frequent itemsets.</td>
<td>52</td>
</tr>
<tr>
<td>3.1</td>
<td>Type hierarchy of X(DM)$^2$.</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>An XML representation of the dataset of Table 3.2.</td>
<td>59</td>
</tr>
<tr>
<td>3.3</td>
<td>A different XML representation of the dataset of Table 3.2.</td>
<td>60</td>
</tr>
<tr>
<td>3.4</td>
<td>A possible XML representation of the dataset of Table 3.4.</td>
<td>66</td>
</tr>
<tr>
<td>3.5</td>
<td>General syntax of XQuake.</td>
<td>72</td>
</tr>
<tr>
<td>3.6</td>
<td>Syntax (a) and intuitive semantics (b) of the wfor iterator.</td>
<td>75</td>
</tr>
<tr>
<td>4.1</td>
<td>Equal-width implementation.</td>
<td>90</td>
</tr>
<tr>
<td>4.2</td>
<td>Equal-frequency implementation.</td>
<td>90</td>
</tr>
<tr>
<td>4.3</td>
<td>Simple random sampling implementation.</td>
<td>95</td>
</tr>
<tr>
<td>4.4</td>
<td>A decision tree over the playTennis training set of Table 3.2.</td>
<td>98</td>
</tr>
<tr>
<td>4.5</td>
<td>The main procedure of the ID3 algorithm in XQuake.</td>
<td>105</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of FP-Tree (from [63]).</td>
<td>111</td>
</tr>
<tr>
<td>5.2</td>
<td>The main procedure of the Apriori algorithm.</td>
<td>117</td>
</tr>
<tr>
<td>5.3</td>
<td>The main procedure of the FP-Growth algorithm.</td>
<td>119</td>
</tr>
<tr>
<td>5.4</td>
<td>Characterization of the classes of constraints in FIM (from [14]).</td>
<td>121</td>
</tr>
<tr>
<td>5.5</td>
<td>Monotonicity and anti-monotonicity in the mine itemsets operator.</td>
<td>128</td>
</tr>
<tr>
<td>5.6</td>
<td>The Ex-Ante procedure at the first level of the level-wise computation.</td>
<td>129</td>
</tr>
<tr>
<td>6.1</td>
<td>The XQuake system architecture.</td>
<td>136</td>
</tr>
<tr>
<td>6.2</td>
<td>The core of the wfor interpreter.</td>
<td>139</td>
</tr>
<tr>
<td>6.3</td>
<td>A sample of compiling mapping for the discretization operator.</td>
<td>142</td>
</tr>
<tr>
<td>6.4</td>
<td>The role of the XQuake compiler.</td>
<td>143</td>
</tr>
<tr>
<td>6.5</td>
<td>The GUI of the XQuake system.</td>
<td>145</td>
</tr>
</tbody>
</table>
6.6 Runtime comparison among XQuake and Weka on the connect-4 dataset (top-left), census dataset (top-right) and mushroom dataset (bottom-left). XQuake performance over the synthetic datasets (bottom-right). ........................................ 148

6.7 Time breakdown on the T20I6D300K dataset (top-left), connect-4 dataset (top-right), mushroom dataset (bottom-left) and census dataset (bottom-right). ........................................ 149

6.8 References over the xmark database (from [101]). .................. 150

6.9 Scalability of six benchmark queries w.r.t. the data size (left) and time difference on x-mark-standard among main memory and on disk execution for the four hardest queries (right). .............. 160

A.1 XML fragments of the dblp dataset. ................................. 168
A.2 XML fragments of a department dataset. ............................ 169
A.3 XML fragments of the mondial dataset. .............................. 170
A.4 A sample of the x-mark dataset: items and persons. ............... 171
A.5 A sample of the x-mark dataset: open and closed auctions. ......... 172
Introduction

Inductive Databases (IDBs) are general purpose databases in which both the data and the knowledge are represented, retrieved, and manipulated in an uniform way. A critical aspect in IDBs is the choice of what kind of formalism is more suited to represent models, data sources, as well as the queries one might want to apply to them. A considerable number of different papers propose to integrate a mining system with a relational DBMS [10, 12, 26, 27, 36, 57, 65, 78, 90, 123]. According to the closure principle, the source data, the induced and the background knowledge are represented as relational tables, while queries are expressed in an SQL-like language. Relational databases are fine for storing data in a tabular form, but they are not well suited for representing large volumes of semi-structured data fields. However, data mining need not be necessarily supported by a relational DB.

I.1 Why mining XML data?

The past few years have seen the growth in the adoption of the eXtensible Markup Language (XML). The great success of XML is due to the fact that it makes it possible to define custom tags, describing the data enclosed by them. Such a flexible nature makes it an ideal basis for defining arbitrary languages having a “nested” structure.

Thus, the increasing adoption of XML has raised the challenge of mining large collections of semi-structured data, such as web pages, graphs, geographical data and so on. For example, application-specific XML vocabularies such as XBRL [125] continues to emerge. XML is used in web services both for service descriptions and for data exchange [114]. The Systems Biology Markup Language (SBML) [100] is a machine-readable language for representing models of biochemical reaction networks. Finally, geographical and spatial data are becoming increasingly important, due to the pervasiveness of location-acquisition technologies such as GPS, GSM networks, etc. In this respect, the Geography Markup Language (GML) [80] serves as a modeling language for geographic systems as well as an open interchange format for spatial transactions on the web.

One of the key issues is the capability of mining large collections of semi-structured data and, in general, data for which an XML representation is less cum-
bersome and more optimal compared to the relational format. Clearly, extracting information from this kind of sources is a very important trend in the data mining community.

Summing up, the use of XML permits to represent, in an uniform way, more complex formats, such as trees or clusters as well as the domain knowledge. From the latter perspective, ontologies seems a natural choice for representing the semantics of a given domain. For example, supermarkets can use ontologies to classify products in sections and brands and the mined models may be composed by sections and brands of the items bought by the users. This opportunity is even more substantial in our project, since ontologies are typically represented via the Web Ontology Language (OWL) [108], or via the Resource Description Framework (RDF) [109]. Both the languages are de facto serialized in XML.

I.2 How is XML data mined?

From the XML querying point of view, a relevant on-going effort of the W3C organization is the design of a standard query language for XML, called XQuery [111, 119], which is creating a lot of research and for which a large number of implementations already exist. From the XML data warehousing point of view, native XML databases are designed for seamless storage, retrieval and manipulation of XML data.

The goal of our research effort is the design and implementation of a mining language and system where a native XML database is used as a storage for KDD entities, i.e. both models and raw data, while DM tasks are expressed in an XQuery-like language, in the same way mining languages on relational databases are expressed in an SQL-like format. In short, the contributions of this thesis can be summarized as follows.

- We propose an XQuery-like language powerful enough to support complex mining tasks, including the specification of domain entities, parameters of the operations and, complex queries on the domain knowledge. Features of the language are: (i) an intuitive syntax, inspired by the XQuery world; (ii) flexibility to specify a variety of different mining tasks in a declarative fashion; (iii) a grammar that accepts a certain degree of extensibility in order to introduce user-defined functions in the statements; (iv) capability of using (both explicit or implicit) intrinsic optimizations over native XML databases, such as indexes; (v) a coherent formalism capable of dealing uniformly with raw data, induced knowledge and background knowledge.

- We design a compact system architecture over an existing native XML database. To couple data mining with databases, the more database-aware applications use loosely-coupled approaches in which the core of the algorithm is typically implemented in a host programming language. On the contrary, tightly-
coupled techniques, instead of bringing the records of the database into the application, selectively push parts of the application that performs the mining into the database [97]. We propose here a hybrid architecture capable of providing both a direct specification of mining algorithms in XQuery and the use of external functions for the efficient implementation of critical aspects. Arguably, XQuery is the most promising programming language for this purpose and this kind of solution has the advantage of avoiding “black box” implementations of mining algorithms permitting the exploitation of domain knowledge within the algorithm specification. We will show some adaptations over XQuery in order to allow the specification of the general schema of algorithms.

- We face the problem of modeling an XML-based prediction framework, that is one of the more widely studied data mining tasks in the research community. We study how complex techniques, e.g. discretization or sampling, can be easily combined in an uniform formalism, thus providing an integration between deduced and extracted knowledge.

- Finally, as a second case study, we present how such a technique can be used to implement the Apriori [1] and FP-Growth [50] algorithms. Despite its datedness, the Apriori algorithm is still a significant example of how the background knowledge can be encapsulated as deeply as possible into the extraction process. In this respect, particular attention is given to the evaluation of the capability of our framework to accept both physical and domain-based optimizations proposed in the existing literature.

I.3 Thesis overview

Chapter 1 treats the Knowledge Discovery in Databases process. The Chapter is centered on the Inductive Database research field, for which a formal definition is provided. A review of a set of mining languages proposed in the past literature for dealing with the design of an inductive environment is also presented.

The Chapter 2 presents the eXtensible Markup Language and its related technologies, with particular attention to the XQuery language. The chapter concludes with an introduction to PMML, a first industrial attempt to join the XML world with the data mining world.

In Chapter 3 we present a data model that serves as a flexible theoretical basis for the notion of XML-based inductive language. On such a data model, a mining language allowing an high-level formalization of specific mining tasks is presented. Also, we determine what extensions are required for turning XQuery into a language
supportive of the implementation of data mining primitives.

Once the foundations of an XML-based inductive environment are clarified, Chapter 4 deals with XML tree classification. Furthermore, we show how classification can be combined with ad-hoc preprocessing techniques, such as discretization and sampling. For each technique, we first provide a rigorous formalization over the data model presented in the previous chapter, and then we discuss the implementation details.

Chapter 5 formalizes the notion of XML frequent patterns discovery. After introducing the basic concepts of frequent itemsets mining, the chapter deals with mining constraints and in particular on the way of exploiting domain-dependent constraints inside our framework, both from the expressiveness point of view and from the implementation point of view. We highlight the expressive power and the versatility of the proposed language by means of several examples.

Chapter 6 describes the implementation details of the system architecture over a native XML database, also envisaging some substantial optimizations. Then, we measure the performance of the system with standard benchmarking tools, comparing the performance of the Apriori algorithm with an existing solution based on a different approach. We complete our research proposing a concise, yet comprehensive set of queries which covers the major aspects of the data mining. Queries are designed over an XML scalable benchmark dataset modeling an Internet auction site.

Finally, Chapter 7 contains the final remarks and suggests avenues for future works.
Chapter 1

Inductive Databases

Abstract

Knowledge Discovery in Databases (KDD) and Data Mining (DM) are areas of common interest to researchers in artificial intelligent, machine learning, statistics, databases, knowledge acquisition, data visualization and high performance computing. Providing an easy exploration and summarization of large databases is the main goal of such disciplines. In this respect, Inductive Databases (IDBs) offers a vision of convergence between KDD and databases aiming at defining the concept of data mining as an efficient and effective query process. Thirteen years after, the challenge raised by such a research field is still open.

This chapter gives an overview to some of the characterizing aspects of inductive databases. This overview is clearly functional to the purposes of the thesis, and does not have the claim to be complete. More specifically, we draw a list of desirable properties to be taken into account in the definition of an IDB framework. They involve several dimensions, such as the expressiveness of the language in representing data and models, the capability of handling the KDD as a process, the closure principle, the capability of providing a support for an efficient algorithm programming. These requirements are a basis for the exploitation and the study of existing IDBs languages based on different approaches.

1.1 Background

1.1.1 What is KDD?

Data Mining (DM) is a technique for inferring knowledge via a generalization of the information in the database. The process of searching for knowledge is usually called Knowledge Discovery in Databases (KDD) [63, 48].
People have been “manually” extracting information from data for decades. However, the proliferation of data and information in the last years has created a new opportunity for extracting knowledge from databases “automatically” and both researchers and industrial developers have responded to this need. So, data mining is a technology that overcomes traditional data analysis methods with more sophisticated algorithms for processing large volumes of data. Many application areas require new techniques for data analysis. They include medicine, biology, finance, marketing, banking, business, engineering and several other important fields.

However, data mining analysis has encountered practical difficulties in meeting the challenges above. There are many core problems in KDD that are yet unsolved, including scalability, high dimensionality, representation issues, search complexity, the use of prior knowledge, data ownership and distribution [48]. Nevertheless, it is a relatively young discipline\(^1\), and the research community yet promises great progresses.

At this point, we can give a definition of knowledge discovery. Among the several definitions, we choose the following: “KDD is the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data” [33].

1.1.2 What is an inductive database?

From the definition above emerges that KDD is the process of automatically discovering “nuggets” of knowledge in large data repositories. As a process, it consists of several repeated phases including feature selection, data reduction, data mining, model evaluation and pattern interpretation, as shown in Figure 1.1. The development of KDD solutions requires to specify the tasks at each phase and the interactions/dependencies among them. Most of the times, this results in a complex task, requiring to combine different sources of data and knowledge, and with many phases iterated in order to reach a local optimum.

Instead, as a process, the KDD is multi-step, iterative and interactive. Multi-step because it is composed of several phases; it is iterative because each step can inspire rectification to the preceding phases; it is interactive because interpretation of the model and even data processing techniques require a strong user interaction.

The concept of Inductive DataBase (IDB) is one of the first attempts to formalize the notion of iterative and interactive mining process. In an IDB, the knowledge extraction application can express both queries capable to access and manipulate data, and queries capable to generate, manipulate, and apply patterns. In other terms, IDBs integrate data with knowledge, and such an interaction acts through query languages supporting (even complex) data mining primitives. The data mining application becomes a querying sequence, where the query language has to replay

\(^{1}\)The first official international conference on Knowledge Discovery and Data Mining (KDD) was held in Montreal - Canada - in 1995
to the need of an high expressive power. In contrast, mining systems supporting it have to execute queries as efficiently as possible.

1.2 Problem formulation

The model of inductive database that we consider in this thesis has been originally proposed by Imielinski and Mannila a dozen years ago [56, 71].

Definition 1.1 (Inductive Database). Assume a database $r$, a language $\mathcal{L}$ for expressing properties of subgroups of the data, and a selection predicate $q$ are given. The predicate $q$ is used for evaluating whether a sentence $\varphi \in \mathcal{L}$ defines a potentially interesting subclass of $r$. The task is to find the theory of $r$ w.r.t. $\mathcal{L}$ and $q$, i.e., the set

$$Th(\mathcal{L}, q, r) = \{\varphi \in \mathcal{L} | q(r, \varphi) = true\}.$$ 

The above definition can be explained by means of a simple example, i.e. a simple and compact way of defining the frequent itemsets mining problem [71].

Example 1.1 (from [71]). Given a relation $r$ with $n$ rows over binary-valued attributes $R$, an association rule is an expression of the form $X \Rightarrow A$, where $X \subseteq R$ and $A \in R$. The interpretation of the rule is that those rows in $r$ that have value 1 for the attributes in $X$, also have value 1 for the attribute $A$. Formally, denoting $t(X) = 1$ if and only if row $t \in r$ has a 1 in each column $A \in X$, the frequency $fr(X)$ of $X$ is defined to be $|\{t \in R|t(X) = 1\}|/n$. A set $X \subseteq R$ is frequent, if $fr(X)$ exceeds a given threshold. The problem of finding all frequent sets can be described as:

- $\mathcal{L} = \{X|X \subseteq R\}$;
- $q(r, X) = true$ if and only if $fr(X) \geq \sigma$ where $\sigma$ is the frequency threshold.
A refinement of the Definition 1.1 above introduces the notion of mining query [18, 19].

**Definition 1.2 (from [18]).** An IDB schema is a pair \( R = (R, (Q_R, e, V)) \) where \( R \) is a relation schema, \( Q_R \) is a collection of patterns, \( V \) is a set of result values and \( e \) is the evaluation function that defines pattern semantics in terms of \( V \), i.e. it maps an instance \( r \) of \( R \) and a pattern \( \phi \in Q_R \) in \( V \). Consequently, an inductive database instance is a pair \((r, s)\), where \( r \) is a database over \( R \) and \( s \subseteq Q_R \).

**Example 1.2.** Let us consider the relation \( \text{mondial}(\text{Country}, \text{Population}, \text{Religion}, \text{Percentage}) \) in which \( \text{Country} \) is the country name, \( \text{Population} \) is the population size for that country and \( \text{Religion} \) is a religion pursued in that country by a certain \( \text{Percentage} \) of the overall population. A trivial mining scenario for such a table consists in detecting the religions in the relation with the average number of persons that pursue such a religion more than a given threshold. The inductive database has:

- \( R \equiv \text{mondial} \)
- \( Q_R = \{ (v, v') | v \in \text{dom}(R[\text{Religion}]), v' \in \mathbb{R} \} \)
- \( e(r, (v, v')) = (v, \text{avg}(\{ p \cdot q | (c, p, v, q) \in r \})) \).

It is trivial to show that the Definition 1.1 and the Definition 1.2 are equivalent. The intuitive idea of inductive databases is to provide a unified view of both inferred (deductive) knowledge, and all derived (induced) knowledge over the data. The major problem is the choice of a suitable representation formalism for the inductive part. For example, it is not clear whether decision trees or other semi-structured formats well-fit into the formal definition of IDBs [73]. However, the strength of IDB consists in the capability to formalize the notion of the mining process by exploiting the concepts of compositionality and closure.

### 1.3 Requirements on inductive databases

Before starting presenting three languages for IDBs, we focus in this section on the question if there exists an high-level set of minimal requirements to be taken into consideration in the design of a framework in support to the overall KDD process. We classify them according to three dimensions (Figure 1.2):

1. the KDD as a process,
2. the data source to explore and the patterns to discover,
3. and the system architecture that supports the query language.
The remainder of this section is organized around these three issues. The next section is devoted to the question if these requirements are satisfied by some existing KDD support environment.

### 1.3.1 Requirements on the KDD process

**R1.1 [Closure]** “Closure principle should be pursued in the design of a KDD query language”.

Relational algebra consists of operations, such as selecting, projecting, joining, in which the output as well as the input of each operator are relational tables. It is the classical example of query compositionality. As earlier mentioned, inductive databases extend the closure principle to the KDD field. It simply states that the output of a KDD query can be the input of another query of compatible type\(^2\).

**R1.2 [Completeness]** “The language/system must be capable of managing the life cycle of a KDD process, from the data preparation step to the deployment of the model, supporting the multi-step, iterative and interactive nature of the process”.

The intrinsic complexity of the design of a KDD application is due to difficulty of a-priori defining the best plan to discover knowledge due to the goal-driven and

---

\(^2\)An orthogonal perspective is the *Pattern Discovery Management* (PDM) approach [11]. It introduces a logical separation between raw data and patterns in order to efficiently manage both of them through dedicated management systems. Differently, in IDBs, patterns are stored along with the data they are extracted from. The position that we maintain in this manuscript is that the closure principle should be a basic paradigm in support of a KDD environment.
domain-dependent nature of the problem. A typical wrong approach is to focus efforts toward a single model or a single phase, without considering the whole KDD as a process.

**R1.3 [Reusability]** “The language must include constructs encouraging the reuse of models and raw data previously computed, as well as the integration of fragments of the process previously defined”.

The reusability in the KDD domain deals at least with two aspects: the data sources and the mined patterns. Data is not only an input to the KDD process, but also an intermediate output and a final output. As an example, a clustering model could be used to fill missing field values of corrupted transactions. As for data, extracted mining models should be stored in an appropriate repository, for subsequent analysis. For instance, a set of association rules may be applied on new data to predict which instances support a subset of them.

Also a process repository can play an important role. It is in fact useful to store the process (defined as a complex mixture of KDD operations, algorithm parameters, constraint specifications) and equip the language with constructs that allow the reuse of parts of the process previously defined. In this way, one can directly integrate this sub-query (i.e. sub-process) inside a more complex one, without specifying it again.

### 1.3.2 Requirements on KDD objects

**R 2.1 [Expressiveness]** “The formalism must be equipped with a high level of expressiveness in order to define and represent complex objects as used in the KDD process”.

KDD includes not only the specification of the process and its operations, but also the logical and physical data, the models, some visualization paradigms, etc. From this perspective, at least two distinct issues must be investigated: (i) how to express the raw data, the extracted models as well as the background knowledge representing a certain domain? (ii) How to represent the process, including the specification of algorithms and operations, and the constraints on the background knowledge?

**R2.2 [Meta-schema definition]** “A KDD support system should allow for specifying and manipulating logical metadata in addition to physical data”.

KDD objects can be considered from two distinct point of view: the physical level and a higher abstraction logical level. A classical example of logical element is the type of the attributes (discrete, continuous, etc.). Also weights (or probabilities) are logical data elements that make sense in affecting the role of an attribute in the construction of a model.

Generally, the logical level involves both source data and knowledge. For example, concerning knowledge, it can be useful to keep inside the model information
about the logical schema of the relation used to build the model itself. A well-defined separation between logical and physical level is important for two distinct reasons. From a conceptual point of view, the logical model provides the user with a meaningful abstraction of raw data and patterns, to be directly analyzed and manipulated. From a physical point of view, such operations can be implemented without accessing to the physical level, making the execution more efficient.

**R2.3 [Extensibility]** “The language should be provided with mechanisms to easily integrate new KDD operations and DM algorithms”.

### 1.3.3 Requirements on the system architecture

**R3.1 [Support for efficient algorithm programming]** Both the language and the system must be equipped with constructs and functionalities that permit to embed domain-specific constraints and use ad-hoc data structures in an efficient way.

In [97], the authors consider a spectrum of architectural alternatives for coupling mining with database systems. Two orthogonal solutions are:

- the adoption of an external ad-hoc algorithm implemented in an efficient imperative language, like C or C++, running outside the DBMS on data typically exported from the database into text or binary files in a proprietary format (loosely-coupled approach);

- the direct encoding of DM algorithms and complex operations within the adopted language formalism. This depends on the type of paradigm used (e.g. relational, logic) and, typically, amounts to extend the query language (e.g. SQL, Datalog) with DM features (tightly-coupled approach).

Both the solutions suffer from some limitations. The main limitation of the second kind of approach is that they make it difficult to implement ad-hoc data structures and procedures that allow physical optimizations for the algorithm [97]. On the other side, loosely-coupled systems model mining algorithms as separate modules. The interaction between the DM algorithm and the query system is provided by collecting the data to be mined in an ad-hoc format directly readable by the algorithm. Typically, also the output of the algorithm should be re-adapted. This “black box” approach does not allow a systematic interaction within the search process. For example, we cannot directly evaluate the domain knowledge during the exploration of the search space, making the adoption of domain-dependent constraints hard.

Summing up, the implementation of ad-hoc space-efficient index structures and the capability of applying domain-specific optimizations require mechanisms integrated both in the language and in the system architecture. Our goal is to evaluate the different ways of integrating optimized DM algorithms within existing languages and systems.
R3.2 - Flexibility. “The system architecture must be easily extensible as far as the introduction of new models, new algorithms and new KDD operations is concerned”.

1.4 Query languages for inductive databases

Embedding the data mining in an uniform framework allows the coherent exploitation of induction and deduction in solving problems. In fact, both pre-processing, and especially post-processing, require some deductive reasoning either on the data to be fed to the inductive data mining step, or on the synthetic models produced by the mining step. For example, pre-processing may require the verification of certain domain rules over the data with the purpose of filtering them. On the other hand, forms of metareasoning over the induced models can be useful in many cases, for example for comparing two models extracted from two different sets of homogeneous data.

A critical aspect to take into account is the choice of what kind of formalism is more suitable to represent models, data sources, as well as the queries one might want to apply on them. Given that data mining can be considered as a form of querying, it is natural to embed it in declarative database approaches, as the following:

- **Logic-based query languages**: they use a logic formalism for developing KDD applications and a deductive DBMS for storing both models and raw data. The focus is generally on datalog-based or prolog-based languages.

- **SQL-based query languages**: they include the development of a mining tool integrated with a relational DBMS. According to the closure principle, both the source data and the induced models are represented as database relations. Queries are specified in an SQL-like language.

- **XML-based query languages**: they use XML in order to represent the models, the data sources and the KDD queries as well.

In each of the above approaches the driving idea is to look at the knowledge discovery process as an extension of a querying process. This goes beyond the current practice with data mining oriented commercial systems [54, 81, 102] in which the data mining step is embedded as one of the available functions of a workbench, where data gathering, preprocessing, post-processing, etc. are performed by other separate functions.

In order to be as concrete as possible, we will investigate three proposed query languages for IDBs: the logic-based LDL++ language/system (Section 1.4.2), the SQL-based ATLaS language/system (Section 1.4.3) and the XML-based KDDML language/system (Section 1.4.4). This choices are motivated by the fact that the three languages overcome some limitations posed by previous languages. In the
remainder of this chapter, it is assumed that the reader is familiar with the fundamental of frequent itemsets mining - as originally introduced by Agrawal et al. in [2] - and in particular with the Apriori algorithm [1]. We refer the reader to the Section 5.1.1.

1.4.1 State-of-the-art: an overview

This sub-section gives an overview of the current state-of-the-art in inductive databases.

Logic-based languages

Extending the deductive expressiveness of query languages while remaining computationally tractable is a target pursued by deductive databases, i.e., a DBMS whose query languages and storage structures are designed around a logical model of data.

Datalog++ is an example of highly expressive language that supports deductive databases. A current implementation of Datalog++ is \textit{LDL++} (Logic Database Language) [120, 6] that is equipped with a set of important features suitable to model complex applications. In particular, it provides, in addition to the typical deductive and relational features such as the \texttt{join}, advanced mechanisms for non-deterministic, non-monotonic and temporal reasoning. The use of \textit{LDL++} in order to integrate the inductive and deductive aspects of the mining process has been investigated in [41, 42]. The authors introduce the concept of \textit{Logic-Based KDD Support Environment} demonstrating its flexibility and expressiveness in supporting the KDD process. This project also shows how the query language deals uniformly with data preparation, model extraction, model evaluation and analysis, describing the case studies on the basis of the introduction of mining primitives as logic aggregates.

A related approach is the \textit{Inductive Logic Programming} (ILP) that studies the role of logic in DM from a machine learning perspective. ILP tries to overcome two main drawbacks of the classical machine learning and DM approaches: (i) the use of a limited knowledge representation formalism and (ii) the difficulties in using substantial background knowledge in the mining process. This line of research has been pursued by Morik [77] and, more recently, by Lisi [67].

A vision of convergence between the deductive databases perspective and the ILP perspective is presented in [43]. In this respect, are of particular relevance both the logical language RDM [29] and the work presented by Flack [34].

In the last few years, logic-based approaches to mining inductive databases can be found in [69] and [79].

SQL-based languages

In the past, the idea of integrating knowledge discovery into relational DBMS has been addressed in a considerable number of different papers [10, 27, 36, 57, 78] and
more recently [123] and [12]. The contributions of this research mainly includes the capability of satisfying the closure property, a critical aspect in inductive databases. The use of SQL also facilitates the work for database experts and it is useful for data manipulation [20]. Nowadays, the trend in SQL-based languages is twofold.

On the one hand, while association rules have been extensively studied in SQL-based approaches, other kinds of mining task, such as classification or clustering, has received a little attention, mainly due to difficulty to efficiently represent mining models via relational tables. Past works tried to overcome this limitation, presenting relational-based views of decision trees, e.g. [9, 35].

On the other hand, past experiences in query optimizations have shown that it is very difficult to extend the relational environments to handle powerful optimizations and data structures [97]. In order to overcome these limitations, commercial and industrial database vendors are taking an alternative approach. An appreciable contribute is the Microsoft OLE DB for DM [10] that achieves a closer integration and a better interoperability of the mining task by combining predictive and descriptive mining. OLE DB can be also considered one of the most recent projects in which the development of the application is tightly-coupled with the DBMS. On the academic side, ATLaS [121, 65] (acronym of Aggregate & Table Language and System) is a proposal for handling data streams whereby operations accept a stream of tuples as input and produce a stream of tuples as output. There are two main advantages achieved by ATLaS. First of all, it shows how to obtain a good degree of scalability in the implementation of mining algorithms via a tightly-coupled approach. Second, ATLaS overcomes the lack of extensibility of the other SQL-based approaches by extending SQL with UDAs.

A new inductive database and query language has been recently conceived by Kramer et al. [90] following the tradition of the cited languages. Their goal is to provide rudimentary support for a plethora of different mining operations: discretization, feature selection, frequent patterns, clustering and classification. The final contribution is a new inductive query language extending SQL, with the goal of supporting the whole knowledge discovery process, from pre-processing via data mining to post-processing.

An alternative and quite interesting approach to this field is presented in [26, 12]. The authors propose to extend the DBMS itself rather than the query language by integrating the mining algorithms into the database query optimizer. They introduce the notion of “virtual mining views” that can be queried as they were standard relational views. Every time the system accesses a virtual mining view, a mining algorithm is triggered to materialize the tuples needed to answer the query.

Finally, a past survey comparing SQL-oriented mining languages is [16].

XML-based languages

An important aspect to take into account when dealing with IDBs is how to make all the heterogeneous patterns, sources of data and other KDD objects coexist in a
In [74] XML has been used as the basis on which to define a semi-structured data model designed for KDD, called XDM. The model is characterized by the fact that it stores both data and mining models in the same database. This allows the reuse of patterns by the inductive database management system. In particular, XDM explicitly represents via XML (i) the data item, i.e. a container of data and patterns and (ii) a statement, i.e. a description of an operator application. The KDD process is represented as a set of relationships between data items and statements. The authors also demonstrate the benefits of XML in terms of extensibility to new patterns and new mining operators.

RapidMiner [32], formerly YALE (Yet Another Learning Environment), is a recent open-source environment for KDD and machine learning in which experiments are described via XML files created with a graphical user interface. While the graphical user interface supports interactive design, the underlying XML representation enables automated applications after the prototyping phase. In such a way, KDD processes are modeled as trees.

The perspective suggested by XDM and RapidMiner is also taken in the KDDML (KDD Markup Language) view [91]. KDDML is a middleware XML-based language (and system) whose aim is to support the development of final KDD applications in which a mixture of database access, data preprocessing, mining extraction and deployment is required. The term XML-based means that data, patterns and KDD queries are represented through XML documents. Elements in KDDML are operators with functional semantics: this allows the nesting of operators like in RapidMiner, though RapidMiner uses procedural semantic.

An appreciable contribution in mining association rules from XML data is the XMine operator proposed by Braga et al [23, 22]. The operator is based on XPath and XQuery to express complex mining tasks by simultaneously targeting both the content and the structure of the data. Similarly to our approach, and in contrast with the solutions proposed by XDM, KDDML and RapidMiner, the KDD process is represented by using a superset of the XQuery language.

Mining XML data is used in an instrumental way in [7], to construct summarized representations of XML data. The authors propose to extract association rules from XML databases as the basis for a pattern based representation of XML datasets. The idea is to use the patterns, wherever possible, to answer queries on the datasets.

### 1.4.2 \(\mathcal{LDL}^{++} \): Logic Database Language

#### Overview

As mentioned, the \(\mathcal{LDL}^{++}\) language allows the user to define distributive aggregates (i.e. aggregates computable by means of a distributive and associative operator), such as \texttt{count}, \texttt{sum}, \texttt{min} or \texttt{average}. Informally, a \textit{distributive user-defined aggregate}
$f$ on a set\textsuperscript{3} $S$, satisfies the following inductive definition [6]:

1. **Base step**: the value $f(\{x\})$ on a singleton set must be defined in terms of $x$.

2. **Inductive step**: for any other set $S' = S \cup \{x\}$ the value of $f(S')$ must be defined in terms of $f(S)$ and $x$.

The predicates `single` and `multi` can be used to express the computation of the base step and the inductive step, respectively. For instance, the following rules define the `count` aggregate:

```
single(count, S, 1).
multi(count, S, c, c + 1).
```

The first clause specifies that the count of a set $S$ with a single element $x$ is 1. The second clause specifies that the count of $S \cup \{x\}$ is $c + 1$ for any set $S$ such that the count of $S$ is $c$.

\texttt{LDL++} uses aggregates as the means for introducing mining primitives into the query language. More specifically, the model is based on inductive rules defining a predicate $P$ as a rule of the form

```
P(\alpha, Aggr[^{\beta}] \leftarrow Q_1, \ldots, Q_n,
```

where $Aggr$ is the name of an aggregate and the metavariables $\alpha$ and $\beta$ represent terms obtained from the tuples resulting from the evaluation of the conjunctive query $Q_1, \ldots, Q_n$.

For example, consider the relation `transaction=(TID, Item, Price, Qty)` in which the first attribute identifies the transaction, `Item` identifies the item purchased and the other attributes capture their intuitive semantics (see Table 3.4). Given the following set of logic facts,

```
transaction(Ron, beer, 10, 10).
transaction(Ron, milk, 5, 2).
transaction(Ron, chips, 3, 20).
transaction(Jeff, wine, 1, 30).
transaction(Jeff, chips, 4, 15).
```

the following clause

```
items_count(T, count[^{I}] \leftarrow transaction(T,I,D,P,Q)
```

intensionally specifies the computation of the count aggregate with respect to the `Item` attribute of the `transaction` table. When evaluated, it groups the instances according to the values of $T$ (i.e. the transaction id), computing the extension of the relation

\textsuperscript{3}A logic-based framework uses special functors to define complex objects, such as sets. Thus, `dept_list(show, \{adams, jones, smith\})` denotes the set of employees working in the show department. Practical languages, such as \texttt{LDL++}, provide many ad-hoc predicates that implement typical operations over sets, such as membership/subset testing, union, difference, etc.
items_count(Ron,3).
items_count(Jeff,2).

Aggregates so defined return results at the end of the computation. To support on-line aggregation and other aggregates requiring early returns, the system defines the predicates \texttt{ereturn} and \texttt{freturn} to specify values produced as early returns and as final returns, respectively [120].

**Example 1.3** (average aggregate). The average of a set of integers $S$ can be computed by means of the following definition of the \texttt{avg} function:

\[
\text{single}(\text{avg}, x, (x, 1)).
\text{multi}(\text{avg}, x, (S, c), (S + x, c + 1)).
\text{freturn}(\text{avg}, (S, c), S/c).
\]

The clauses above state that if $S$ contains $c$ elements whose sum is $S$, then $S \cup \{x\}$ contains $c + 1$ elements whose sum is $S + x$. The output of the \texttt{single} and \texttt{multi} predicates is a pair containing the sum and the number of elements counted. The \texttt{freturn} clause returns the final value of the computation.

Early returns act in a similar way: if we add also the clause \texttt{ereturn(avg, (sum, 100), sum/100)} to the example above, then intermediate values will be returned every 100 records counted.

The main limitation of the user-defined aggregate model is the impossibility of defining forms of aggregates more complex than distributive ones. In many cases, simple aggregate may require multiple steps over data in order to be computed. \textit{Iterative user-defined aggregates} are an extension of user-defined aggregates specified by means of an iterative function \texttt{aggr} over a set $S$ [41]:

\[
\text{aggr}(S) \{
\text{init}(\text{res});
\text{do}
\text{res} = f(\text{res}, S) \\
\text{while (iterate(\text{res}))};
\text{return res};
\}
\]

where $f$ is defined, as usual, as a distributive aggregate and \texttt{iterate} is a user-defined condition specifying whether the aggregate computation is completed.

**Example 1.4** (standard deviation aggregate). In order to compute the absolute standard deviation of a set of integers $S$, data are scanned twice. The first scan computes the mean value $M$ (the \texttt{single} and the first \texttt{multi} clauses below), and the second one computes the sum of the absolute difference with the mean value $M$ (the second and third \texttt{multi} clauses). The \texttt{iterate} clause runs when the first scan of data terminates. The \texttt{freturn} clause returns the final output of the computation.
\( \text{single}(\text{stdev}, X, \text{nil}, X, 1)) \).
\( \text{multi}(\text{stdev}, (\text{nil}, S, C), X, (\text{nil}, S+X, C+1)) \).
\( \text{multi}(\text{stdev}, (\text{M}, \text{STD}), X, (\text{M}, \text{STD}+(\text{M}-X))) \quad \text{M}>X \).
\( \text{multi}(\text{stdev}, (\text{M}, \text{STD}), X, (\text{M}, \text{STD}+(\text{X}-\text{M}))) \quad \text{M}<X \).
\( \text{iterate}(\text{stdev}, (\text{nil}, S, C), (S/C, 0)) \).
\( \text{freturn}(\text{stdev}, (\text{M}, \text{STD}), \text{STD}) \).

\( \mathcal{LDL}++ \) and the KDD process

\( \mathcal{LDL}++ \) represents input dataset as extensional predicates. Let us consider again the relation \text{transaction} above. The examples reported below show how a logic-based language is capable of achieving the closure principle.

**Example 1.5** (data preparation). We aim at filtering from the \text{transaction} dataset the low-spending customers, i.e. customers with a total expenditure less than 70 and with at most 4 items bought. This data preparation step can be easily achieved by defining the following \text{datalog++} program:

\[
\text{tSet}(\text{TID},\text{Item},\text{count}\text{Item},\text{sum}\text{V}) \leftarrow \text{transaction}(\text{TID},\text{Item},P,Q),
\quad V = P \times Q, V > 70, \text{Item} \leq 4.
\]

where \text{count} and \text{sum} are distributive aggregates over the set of items \text{Item}.

**Example 1.6** (data mining). Let us consider the result obtained by the application of the \text{tSet} predicate. The following rule specifies the computation of 10\%-frequent itemsets:

\[
\text{frequentPattern}(\text{apriori}<0.1, S>) \leftarrow \text{tSet}(\text{TID}, S, N, V).
\]

where \text{apriori} is an iterative aggregate \cite{41} that allows the discovery of frequent itemsets on the set of items \text{S}. We will formally define it in the remainder of this section.

**Example 1.7.** We can easily combine such two rules and complicate our query with a post-processing task that returns only patterns having exactly three items in the body.

\[
\text{filteredPattern}(P,S) \leftarrow \text{frequentPattern}\{\text{A}, \text{B}, \text{C}\}, S).
\quad \text{frequentPattern}(\text{apriori}<0.1, S>) \leftarrow \text{tSet}(\text{TID}, S,I,V), V > 70, I \leq 4.
\quad \text{tSet}(\text{TID}, \text{Item}, \text{count}\text{Item}, \text{sum}\text{V}) \leftarrow \text{transaction}(\text{TID}, \text{Item}, P,Q), V=P\times Q.
\]

The resulting program now models three steps of the KDD process - data preparation, mining and post-processing - in which the dataset to be mined is filtered and tuned according to mining objectives.
In order to demonstrate the flexibility in dealing with the overall KDD process, \( \mathcal{LDL}^{++} \) has been used for more complicated data mining tasks, such as Bayesian classification, supervised discretization, and clustering \[42\].

Finally, facts obtained by evaluating predicates can be persistently stored in the repository as a text file. For instance, the itemsets obtained by running the program of the Example 1.7 can be easily reused inside a new logic-based KDD application by simply invoking the predicate \( \text{filteredPatter}(P,S) \).

**\( \mathcal{LDL}^{++} \) and the representation of KDD objects**

Both the source data and the patterns are represented by first order predicates defined either extensionally or intensionally. The extension of a relation is viewed as a set of facts, where each fact corresponds to a tuple in the relational model. The following datalog facts contain the patterns obtained by invoking \( \text{filteredPatterns} \):

\[
\text{filteredPattern}\left(\{a,c,d\}, 0.2\right).
\text{filteredPattern}\left(\{d,e,b\}, 0.14\right).
\text{filteredPattern}\left(\{a,b,c\}, 0.116\right).
\ldots
\]

More complex reasoning schemes can be formalized by combining background knowledge with mining. The authors report some examples and they demonstrate how \( \mathcal{LDL}^{++} \) well-adapts to this purpose.

Moreover, in order to add a new KDD operator or a new DM algorithm, one must integrate the language with a distributive or iterative user-defined aggregate, that incorporates the implementation of the operator. The aggregate so defined can be directly used inside a logical rule.

**The \( \mathcal{LDL}^{++} \) system architecture.**

Let us now to explain how the \texttt{apriori} aggregate of Example 1.6 is defined. Basically, it allows the discovery of frequent itemsets on the set of transactions, in which each element is a set of items, e.g. \{\{beer, milk, cheeps\}, \{wine, cheeps\}, \ldots\}. It can be defined as follows \[41\]:

\[
\text{single}(\text{apriori}, (\text{ms}, S), ((\text{ms}, 1), T)) \leftarrow \text{init}(S, T).
\text{multi}(\text{apriori}, (\text{ms}, S), ((\text{ms}, \text{freq}), T), ((\text{ms}, \text{freq}+1), T)) \leftarrow \text{init}(S, T).
\text{iterate}(\text{apriori}, (\text{ms}, \text{freq}), T), (\text{ms} \times \text{freq}, T)) \leftarrow \text{prune}(\text{ms}, T), \text{enhance}(T).
\text{multi}(\text{apriori}, (\text{ms}, S), (\text{ms}, T), (\text{ms}, T)) \leftarrow \text{count}(S, T).
\text{iterate}(\text{apriori}, (\text{ms}, T), (\text{ms}, T)) \leftarrow \text{prune}(\text{ms}, T), \text{enhance}(T).
\text{freturn}(\text{apriori}, (\text{ms}, T), (I, \text{freq})) \leftarrow \text{itemset}(T, (I, \text{freq})).
\]

In such a schema, the first two rules collect and count the frequencies of the 1-itemsets given an input transaction, \( S \), and the minimum support required, \( \text{ms} \). More specifically, the single clause specifies that the number of transactions is 1
for a set $S$ with a single transaction $\{x_1, \ldots, x_n\}$. The multi clause specifies that the number of transactions of a set $S \cup \{x_1, \ldots, x_n\}$ is $freq+1$ for any set $S$ such that the number of transactions is $freq$. The variable $T$ represents, in this case, the reference (i.e. the object identifier) to a hash tree structure implemented in C++. The external C++ predicate init($S$, $T$) initializes and updates the frequencies of the 1-itemsets available from the transaction $S$ in $T$.

The subsequent iterations realize the steps of the Apriori algorithm by counting the candidate itemsets (fourth rule), by pruning unfrequent candidates according to the minimum support, and by generating the new candidates (third and fifth rule). In particular, the first iterate predicate initializes the counter of each candidate itemset, and it activates the computation of its frequency by means of the subsequent multi clause. More precisely:

- the external prune($ms$, $T$) predicate removes all the itemsets in $T$ whose frequencies are less than the minimum support;
- the enhance($T$) predicate combines the frequent $k$-itemsets in $T$ and generates the candidate $(k+1)$-itemsets;
- the count($S$, $T$) predicate updates the frequencies of each itemset in $T$ according to the transaction $S$.

The output answer returned by the freturn clause is a pair $(I, freq)$ such that $I$ is an itemset of frequency $freq > ms$. In particular, the itemset($T$, $(I, freq)$) external predicate extracts the frequent itemset $I$, whose frequency is $freq$, from $T$.

It is important to point out that the implementation of the main operations of the Apriori algorithm is directly specified through the definitions of the predicates init, count, enhance, prune, itemset that are computationally-intensive and play an important role in increasing the overall performance of the algorithm [41].

This “middle-road” approach provides many opportunities of efficiently using data structures implemented in C++. More interestingly, the above framework allows the exploitation of domain-specific optimizations since constraints can be defined directly within the deductive view of the algorithm specification, and they can be evaluated during the exploration of the search space by means of the single, multi and iterate constructs. The schema maintains the structure of the results and, above all, it can be generalized to other algorithms.

### 1.4.3 ATLaS: Aggregate & Table Language and System

**Overview**

ATLaS [121, 65] adds to SQL the ability of defining *User-Defined Aggregates* (UDAs). It adopts from SQL-3 the idea of specifying a new UDA by an INITIALIZE, an ITERATE and a TERMINATE computation [121]. The UDAs accept a stream of tuples as input and produce a stream of tuples as output.
Example 1.8 (Standard average (from [65])). The standard average, \texttt{avg}, can be defined by means of the following aggregate.

\begin{verbatim}
AGGREGATE avg(Next Int) : Real {
    TABLE state(sum Int, cnt Int);
    INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
    }
    ITERATE : {
        UPDATE state SET sum = sum + Next, cnt = cnt + 1;
    }
    TERMINATE : {
        INSERT INTO RETURN SELECT sum/cnt FROM state;
    }
}
\end{verbatim}

where \texttt{state} is a local relational table that stores the temporary result of the computation and contains the current sum and the number of elements visited. The \texttt{INITIALIZE} statement initializes the value taken from the input stream and sets the count to 1, the \texttt{ITERATE} statement updates the table and the \texttt{TERMINATE} statement returns the final result (a single value in this case) of the computation when no more input tuples are available.

The UDA so defined can be invoked as any other built-in aggregate inside the body of a program (i.e. inside a SQL SELECT statement). For example, the following statement computes the average price from the \texttt{transaction} table of the Example 1.5:

\begin{verbatim}
SELECT avg(t.Price) FROM transaction as t.
\end{verbatim}

Recursive queries can be supported in \texttt{ATLaS} without any new construct since UDAs can call other aggregates or call themselves. The innovation of this approach is that they express both traditional \textit{blocking aggregates} and \textit{non-blocking aggregates} - in which the \texttt{TERMINATE} statement is empty or absent - by using the same basic framework. For instance, the on-line computation of the \texttt{avg} aggregate of Example 1.8 can be easily achieved by moving the \texttt{INSERT INTO} clause from the \texttt{TERMINATE} to the \texttt{ITERATE} statement and by adding the condition "\texttt{WHERE cnt \% 100 = 0}". In this way, an intermediate value is returned every 100 tuples counted.

\textbf{ATLaS and the KDD process}

The closure principle is achieved in two cases. On the one side, tables and models are represented as relational tables. On the other side, also UDAs and table functions are defined in SQL itself. UDAs are some minimal extensions of SQL that are particularly effective for expressing efficiently DM tasks. [65] shows that \texttt{ATLaS} can simulate Turing machines and there are no sub-fragments that are not Turing
complete. The authors provide additional examples, such as a decision tree classifier, ROLAP [121], density-based clustering and spatial/temporal applications.

Concerning the reusability feature, data and models are persistently stored in the DBMS as one or more relational tables, and they can be easily obtained by means of a SELECT SQL statement. The process is represented as a mixture of SQL queries and user defined aggregates.

**ATLaS and the representation of KDD objects**

The source data is naturally specified as a relational entity. As mentioned, in order to respect the closure principle, also the extracted knowledge is expressed via a relational format. For example, frequent itemsets are stored in a prefix tree structure called trie. In ATLaS, the trie is represented by an in-memory table in which each record contains an item, as well as a pointer to its parent node that is another record in the trie table (see fig. 1.3). A compact representation of a set of association rules is obtained by means of a relational table with four fields:

1. the reference to the itemset body contained in the trie;
2. the reference to the itemset head contained in the trie;
3. a real value representing the support;
4. a real value representing the confidence.

As an example, Figure 1.3 (d) depicts a relational table for storing the four rules

\[
\begin{align*}
    a & \rightarrow b \quad (\text{support} = 0.416, \text{confidence} = 0.833) \\
    b & \rightarrow a \quad (\text{support} = 0.416, \text{confidence} = 0.714) \\
    a & \rightarrow c \quad (\text{support} = 0.333, \text{confidence} = 0.666) \\
    c & \rightarrow a \quad (\text{support} = 0.333, \text{confidence} = 0.799).
\end{align*}
\]

Other models can be similarly expressed. For example, a decision tree can be represented with a row for each node and with the following columns: the node unique identifier, the pointer to the father, the number of children, the attribute selected at the node and statistical information at the node.

As mentioned, the ATLaS approach also favors the extensibility of the language: adding a new data mining functionality or a new algorithm means integrating the language with a new aggregate that supports the implementation of the operation. In other terms, a one-to-one correspondence between UDAs and KDD operations can be easily achieved.
The ATLaS system architecture

UDAs can be used for a tightly-coupled implementation of the data mining algorithms and for encoding data structures in SQL itself. As an example, the ATLaS implementation of the Apriori algorithm is reported in [121]. We introduce here only the main idea.

The algorithm first scans the dataset to find out frequent 1-itemsets and it inserts them into a trie (see Figure 1.3). Next, it uses the frequent 1-itemsets so computed to generate 2-itemsets. At this time, the algorithm recursively extends the trie level by level until no more frequent itemsets can be found. This is achieved by means of the UDA countset in which the INITIALIZE and ITERATE routines scan the input data and count the occurrences of each candidate (support counting phase). The TERMINATE routine is responsible for extending the trie to a new level by pruning the candidates according to the anti-monotonicity constraint (candidate generation phase). The procedure so defined is recursive, and uses further UDAs to compute frequent itemsets.

Generally speaking, a good degree of performance for data intensive SQL-based KDD applications requires new features for the language/system architecture. The features added to ATLaS include support for in-memory tables and Object Identifiers (OIDs).

In-memory tables are used to store the temporary result of a computation and they play an important role for defining data structures. In the example below, a prefix tree stores frequent itemsets during the candidate generation phase of the Apriori algorithm. An index is defined on the father attribute:

TABLE trie(item Int, father REF(trie), supp Real) INDEX(father) MEMORY;
Currently, ATLaS supports both secondary storage tables indexed by B+ or R+ trees on one or more attributes and main memory tables, with a hash-based index on one or more attributes. Only the first type of table consists of persistent data sources.

OIDs are used to reference tuples, to facilitate the implementation of in-memory data structures, and to navigate through the instances. As an example, consider again the trie table above in which the field father points to another record of the trie table. Similarly, association rules are defined by means of OIDs, since body and head are itemsets contained in the trie table:

\[
\text{TABLE rda(head REF(trie), body REF(trie), support Real, confidence Real).}
\]

Moreover, the UDA also permits the exploitation of domain-dependent optimizations since constraints can be expressed in an SQL-like format and they can be evaluated during the exploration of the search space. For example, we can modify the definition of the UDA that implements the Apriori algorithm in order to filter uninteresting itemsets directly within the mining algorithm.

The language processor translates an ATLaS program into C++ code, which is then compiled and linked with the database storage manager. In order to extend the system, one must provide only the implementation of an UDA, without take into consideration the architecture of ATLaS.

1.4.4 KDDML: a KDD Markup Language

Overview

KDDML (Knowledge Discovery in Databases Markup Language) is a middleware XML-based language (and system). The term middleware means that higher abstraction levels must be built on top of it, such as vertical applications, high-level GUIs, or more declarative languages. The term XML-based refers to the fact that data, patterns (i.e., mining models) and KDD queries are represented through XML documents.

XML tags in KDDML correspond to operators with a functional semantics that allows nesting of KDD tasks, thus permitting to view the KDD as a process. The XML syntax of a generic operator is shown below:

```
<OPERATOR_NAME att1="v1" ... attM="vM">
  <ARG1_NAME> .... </ARG1_NAME>
  ...
  <ARGn_NAME> .... </ARGn_NAME>
</OPERATOR_NAME>
```

The attributes correspond to parameters of the operator (e.g., the target attribute of a tree miner algorithm). Sub-elements `<ARG1_NAME>`, ..., `<ARGn_NAME>` define arguments passed to the operator. As it can be expected, arguments must be of
an appropriate type and sequence, i.e. an operator signature must be specified. Intuitively, we can have different types for data sources, for each mining model and for operators returning a scalar.

The key property is that every above mentioned KDD object is defined in an uniform way by means of an XML DTD (Document Type Definition). Moreover, KDD operations can be incorporated in the language by means of XML DTDs. One can denote the signature of an operator \( f : t_1 \times \ldots \times t_n \rightarrow t \) by defining a DTD for KDD queries that constrain sub-elements to be of type \( t_1, \ldots, t_n \). DTD entities are used to group operators returning the same type as output. As an example, this fragment of the DTD:

```xml
<!ELEMENT RDA_MINER ((%kdd_query_table;), ALGORITHM)>
<!ATTLIST RDA_MINER xml_dest %string #IMPLIED>
```

requires that the first sub-element of the \(<RDA_MINER>\) operator be one of those in the entity \( \text{kdd_query_tables} \) (i.e. all operators returning a data source) and the second one is the specification (i.e. name and parameters) of a DM algorithm.

### KDDML and the KDD process

The KDDML language is typed and compositional, i.e. an operator of a given type can be used as an argument of another operator which expects that type. Type checking is mainly static, performed during the parsing of XML DTDs.

**Example 1.9.** Let us consider the problem of extracting association rules from a set of transactions. The KDDML query is reported below.

```xml
<KDD_QUERY name="mining and post-processing">
  <RDA_FILTER xml_dest="output.xml">
    <RDA_MINER>
      <TABLE_LOADER xml_source="transactions.xml"/>
      <ALGORITHM algorithm_name="DCI">
        <PARAM name="min_support" value="0.3"/>
        <PARAM name="min_confidence" value="0.5"/>
      </ALGORITHM>
    </RDA_MINER>
    <CONDITION>
      <AND_COND>
        <BASE_COND op_type="having" term1="@body" term2="3"/>
        <BASE_COND op_type="is_in" term1="@head" term2="a"/>
      </AND_COND>
    </CONDITION>
  </RDA_FILTER>
</KDD_QUERY>
```

The model extracting takes place in three different steps.
• The operator `<TABLE_LOADER>` allows a transparent access to the XML source, `transactions.xml`, stored in the system repository.

• A first set of association rules is mined from such transactions specifying a minimum support of 30% and confidence of 50% (operator `<RDA_MINER>`). Notice that the `<RDA_MINER>` specification expects a sub-element with input data (tag `<TABLE_LOADER>`) and a second sub-element with the algorithm name and parameters (tag `<ALGORITHM>`). The algorithm used here is DCI (Direct Count & Intersect) [83], an efficient C++ Apriori implementation.

• The operator `<RDA_FILTER>` extract from the rules of the previous step those having exactly three elements in the body and having the item a contained in the head of the rule. Such filtering conditions are specified in the `<CONDITION>` sub-tag. The `xml_dest` attribute contains the output of the operator.

The KDDML system/language incorporates a rich library of operators and algorithms capable of dealing with different phases and aspects of the KDD process. [91] reports some example of preprocessing, mining and post-processing operators and it shows the versatility of XML to manage the entire KDD process.

The data (resp. models) repository is populated by KDDML queries that yield a table (resp. model) as output. It also includes constructs (such as `<TABLE_LOADER>` and `<RDA_LOADER>`) for the direct access to data and models. In addition, a query repository stores the KDD processes as XML documents. In order to modularize long XML queries (i.e. complex KDD processes), the operator `<CALL_QUERY>` retrieves and evaluates sub-queries (i.e. fragments of process) in the query repository. It can be used as input to any other operator of compatible type.

KDDML and the representation of KDD objects

KDDML represents models as an extension of the Predictive Model Markup Language (PMML). As detailed in Section 2.3, PMML offers a typical example of meta-schema definition in which the logical component of the model is separated from the physical one.

The same feature can be achieved for the metadata of a data source. The schema of a table is stored in as XML document that contains information on attributes, which cannot be automatically derived from the attribute values. As an example, the following XML fragment contains statistics of the relation `transaction.xml` of Example 1.9:

```xml
<SCHEMA number_of_attributes="5" number_of_instances="10">
  <ATTRIBUTE name="TID" number_of_missed_values="0" type="string"/>
  <ATTRIBUTE name="item" number_of_missed_values="0" type="string"/>
  <ATTRIBUTE name="price" number_of_missed_values="0" type="numeric">
```
Some kind of reasoning is applicable to the fragment above by defining ad-hoc KDD operators that do not access the physical instances, e.g. preprocessing operators are defined for renaming an attribute or changing its type.

Conditions on the background knowledge are encapsulated in the XML fragment representing the mining query. In the Example 1.9 above, the tag `<CONDITION>` specifies a filter on a set of association rules. More generally, it is defined as an and, or, not combination of primitive cases, that can be used to evaluate boolean operators (such as ≤) on table attributes and/or constants. Here, table attributes stand for both relational (or preprocessing) table columns and model properties (e.g. the confidence of an association rule).

Extending the KDDML language is as simple as possible. Adding a new DM or preprocessing algorithm does not require any DTD modification, since the algorithm name and the parameters are not part of the language syntax. To some extent, also adding new forms of extracted knowledge should not jeopardize the overall design of the system: adding a new mining model to the language means adding a new type in the operator signatures, which amounts to non-destructive changes in the DTD of the operators.

The KDDML system architecture

The language is supported by a Java-based system that is structured in layers in order to achieve high modularity [91]. Language operators are directly implemented in KDDML. However, when a high level of efficiency and scalability is required (for instance for DM algorithms), the algorithm itself is implemented as an (external) C/C++ program or as a library which requires its own input format and provides its own output format.

The interaction between the query system and the DM algorithm is provided by collecting the data to be mined in an ad-hoc proprietary format required by the algorithm itself. Typically, also the output has its own proprietary representation formalism that should be adapted to the KDDML format. As a consequence, the implementation of a mining operator scans data tuple by tuple from the repository, it transforms it into the input required by the external algorithm, it calls the actual algorithm and finally it interprets the output to return an appropriate PMML object. In such an approach, only domain-independent optimizations can be adopted. In fact, such a “black box” design does not allow a systematic interaction within the search process. In other terms, in order to evaluate domain-specific constraints, one should “open” the box modifying the library containing the algorithm.
The modularity of the system architecture is achieved by means of Java interfaces: in order to add a new operator/algorithm, one must implement a specific interface. Interfaces are organized in a Java hierarchical structure, according to the type of algorithm and to the KDD phase involved.

Finally, the KDDML system has been redesigned in order to support KDD on computational grids [92, 93]. These works show how a KDD application can be easily decomposed into autonomous sub-queries, in order to achieve a good degree of parallelization, and how an XML-based design well-adapt to a grid environment.

1.5 Summary

In this chapter we present a set of desirable features for an ideal query language for KDD. We consider three languages: $\mathcal{LDL}++$, ATLaS and KDDML. They share the closure, a crucial principle for IDBs. However, they are based on different paradigms: $\mathcal{LDL}++$ uses an extension of the logical language datalog; ATLaS is an SQL-based language with data streaming functionalities; KDDML is centered on XML, that is used both for knowledge and process representation. In the following we briefly summarize common points and main differences among $\mathcal{LDL}++$, ATLaS and KDDML (see Table 1.1):

- The closure property is achieved by means of compositionality. The main difference among the languages is in type checking: in KDDML there is a static type checking achieved during the parsing of the DTDs; ATLaS and $\mathcal{LDL}++$ support a run-time type checking.

- ATLaS and $\mathcal{LDL}++$ achieve a high degree of extensibility and modularity by means of UDAs and iterative user-defined aggregates, respectively. The KDD process is seen as a mixture of logical rules and distributive and iterative user-defined aggregates (resp. SQL queries and UDAs). Adding a new functionality means adding a new distributive or iterative aggregate (resp. UDA) to the language/system that implement it. On the contrary, KDDML uses XML DTDs in order to assign semantics to algorithms and operators. Adding a new KDD operation means integrating the language with a new function (i.e. a new XML tag). Adding a new model means integrating the DTD with a new type (i.e. a new XML entity).

- The last but not least important aspect concerns the mechanisms provided for an efficient algorithm programming. In KDDML, DM algorithms are modeled as efficient C/C++ external libraries integrated within the system. We can notice, that by using such a decoupled strategy, we cannot directly exploit the domain knowledge within the algorithm. On the contrary, ATLaS investigates a database-oriented solution based on in-memory relational tables with hash-based indexes and OIDs to reference attributes. These mechanisms are
### 1.5. SUMMARY

<table>
<thead>
<tr>
<th>Req</th>
<th>Description</th>
<th>LDL++</th>
<th>ATLaS</th>
<th>KDDML</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Closure Principle</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>1.2</td>
<td>Completeness</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>1.3</td>
<td>Reusability</td>
<td>yes</td>
<td>yes</td>
<td>yes$^1$</td>
</tr>
</tbody>
</table>

#### Requirements on KDD objects

<table>
<thead>
<tr>
<th>2.1.1 Express. on data</th>
<th>Log. facts</th>
<th>Rel. tables</th>
<th>XML &amp; CSV$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.2 Express. on models</td>
<td>Log. facts</td>
<td>Rel. tables</td>
<td>XML</td>
</tr>
<tr>
<td>2.1.3 Express. on the Datalog++ SQL &amp; UDAs XML</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2 Meta schema def.</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>2.3 Extensibility</td>
<td>Distr. &amp; iter. UDAs DTDs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Predicates</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Requirements on the system architecture

<table>
<thead>
<tr>
<th>3.1 Support for alg. implementation</th>
<th>User-defined predicates$^3$ and OIDs</th>
<th>In-memory tables</th>
<th>External Alg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Flexibility</td>
<td>Distr. &amp; iter. UDAs</td>
<td>Java predicates interfaces</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Comparison between $\mathcal{LDL}++$, ATLaS and KDDML. $^1$In addition, KDDML also supports an explicit construct for the direct integration of sub-queries. $^2$KDDML uses XML to represent data schema and CSV (Comma Separated Value) to represent the physical instances. $^3$ $\mathcal{LDL}++$ uses an hybrid approach between the loosely-coupled and tightly-coupled approaches.

also important to define efficient data structures. $\mathcal{LDL}++$ adopts a trade-off between the two solutions. It uses the capability of implementing some functionalities by means of a procedural language while maintaining the rest of the program in $\mathcal{LDL}++$. This “middle-road” approach permits the direct implementation of domain-specific optimizations and ad-hoc data structures (in C++).
Chapter 2

XML and related technologies

Abstract

Since this thesis deals with XML data mining, a brief introduction to XML and its related technologies is in order. In this chapter, particular attention is paid to the XQuery language, that is for XML what SQL is for relational databases\(^1\). It also discusses the XQuery Data Model (XDM), i.e. the data model on which XQuery is based. XDM is fundamental in our context in order to well-define the underlying data model behind our language, that is a simple extension of XDM.

The chapter concludes with an introduction to the Predictive Modeling Markup Language (PMML), a first industrial attempt to join the XML word with the data mining word. We use PMML for knowledge representation.

2.1 Introduction

The \textit{eXtensible Markup Language} (XML) [107] is the rapidly emerging standard for representing semi-structured data. XML facilitates the exchange of data between systems since documents are self-describing, i.e. they don’t need to conform to the internal layout of any system but only to an agreed exchange format. Such a format is specified as a \textit{document type definition} (DTD) [107] or as an \textit{XML Schema}\(^2\) definition [110] associated to the exchanged XML document.

In the remainder of this section we discuss the basics of XML.

\(^1\)For this reason, XQuery is sometimes called the “SQL of XML”.

\(^2\)A DTD specifies the grammar of an XML document, but uses only few simple built-in data types. Its evolution, the XML Schema has richer data types (including user defined ones) and grammar rules. XML Schemas are themselves XML documents.
2.1.1 XML querying

A relevant on-going effort of the W3C organization is the design of a standard query language for XML, called XQuery [111, 119], which is built on intense research in the field and for which a large number of implementations already exists. Its design followed the needs of:

- to be useful for both highly structured and semi-structured documents;
- to be a declarative language rather than a procedural one;
- to be strongly typed, allowing the identification of possible errors and the optimization of the query;
- to allow querying across collections of documents;
- to use and share as much as possible appropriate W3C recommendations, such as XML 1.0, Namespaces, XML Schema.

Section 2.2 provides a brief overview of the foundations of XQuery. For any further detail, we refer the interested reader to [111, 119].

2.1.2 XML processing

API’s for navigating XML documents include the Document Object Model (DOM) and the Simple API for XML (SAX). DOM provides a main-memory hierarchical (tree-like) object oriented model of XML documents. SAX defines an event-based approach (more suitable for external-memory processing) whereby parsers scan through XML data, calling handler functions whenever certain patterns are found. As far as XQuery is concerned, the XQuery API for Java (XQJ) standard is being developed under the Java Community Process [105].

2.1.3 XML storage

As more and more organizations employ XML data, classical data management issues pertaining to XML’s efficient storage, indexing and manipulation arise. Most commercial relational and object-oriented systems offer extensions to support the management of XML data. For example, commercial RDBMS also provide “XML” data types for table columns, extending SQL with operators for XML querying via XQuery or a subset of it. On the other side, transforming relational data into (simple) XML and back can be achieved with existing commercial RDBMS and middleware products. Unfortunately, each product provides its own DTD or schema and there is no standard extension of SQL dealing with this issue. Moreover, storing XML data that bear complex and irregular data structures into a relational database is not trivial. Querying such data also necessitates costly operations to
translate XQuery queries into SQL, and then to translate the result from relations to XML.

In addition to supporting XML within RDBMS, *Native XML Databases* (NXD) have been born. The term “native” generally refers to a database that is designed for XML content from the ground up, as opposed to a traditional relational database. Rather than being oriented around tables and columns, its data model is based on hierarchical documents and collections of documents [119]. Products including a NXD, in general, provide the traditional capabilities of databases, such as data storage, indexing, querying, loading, extracting, backup, and recovery. Most of them also provide some added value in addition to their database capabilities. For example, they might provide advanced full-text searching functionality, document conversion services, or end-user interfaces. In contrast to the previous solution, storing XML data directly in native mode has the advantage of simplicity. However, a great research effort must be spent to optimize the performances of NXDs.

The remainder of this thesis takes you through a set of example and three XML documents are used repeatedly as input documents to the query examples. We refer the reader to Appendix A for a look to the dblp, department and mondial XML documents.

### 2.2 The XQuery language: an overview

XQuery is a language for querying XML data that became an official W3C Recommendation in January 2007. It allows to select the XML data elements of interest, reorganize and possibly transform them, and return the results in a structure of your choice. XQuery has a rich set of features that allow many different types of operations on XML data and documents, including [119]:

- Searching for information within a document or set of documents.
- Filtering out unwanted information.
- Joining data from multiple documents or collections of documents, sorting, grouping, and aggregating data.
- Transforming and restructuring XML data into another XML vocabulary or structure.
- Performing arithmetic calculations on numbers, manipulating strings to reformat text.

In the next sections we explore path expressions and FLWOR clauses, that are two common ways of writing queries in XQuery. Then, we briefly explain the data model on which XQuery is based.
2.2.1 Path expressions

Path expressions are used to traverse an XML tree to select elements and attributes of interest. They are similar to paths used for filenames in many operating systems and they consist of a series of steps, separated by slashes, that traverse the elements and attributes in the XML documents.

Example 2.1. The path expression `doc("dblp")/inproceedings/author` selects all the `<author>` elements from the `dblp` document. There are three steps: (i) `doc("dblp")` calls an XQuery function named `doc`, passing it the name of the file to open (ii) `inproceedings` selects all the conference papers, (iii) the outermost element selects all the `author` children of `inproceedings`. The result of the query will be all the authors of the `dblp` database that published in at least a conference, exactly as they appear and with (possible) duplicate elements.

```xml
<author dept="id392">
  <first-name>Enrico</first-name>
  <last-name>Fermi</last-name>
</author>
<author dept="id923">
  <first-name>Albert</first-name>
  <last-name>Einstein</last-name>
</author>...
```

Path expressions can also return attributes, using the special `@` symbol. The asterisk `*` is used as a wildcard to indicate any element name. Alternatively, we can use a double slash `//` to return elements that appear anywhere in the document.

In addition to traversing the XML document, a path expression can contain predicates that filter out elements or attributes that do not meet a particular criterion. Predicates are indicated by square brackets. For example, the path expression:

```
doc("dblp")/inproceedings/author[@dept = "id453"]
```

selects only those `<author>` elements whose `dept` attribute value is “id453”.

Path expressions are convenient because of their compact, easy-to-remember syntax. However, they have a limitation: they can only return elements and attributes as they appear in input documents. Any element selected in a path expression appears in the results with the same name, the same attributes and contents, and in the same order as in the input document.

2.2.2 FLWOR expressions

The basic structure of XQuery is the FLWOR expression that stands for “for, let, where, order by, return”. FLWORs, unlike path expressions, allow to manipulate, transform, and sort results.
Example 2.2. The query below shows a simple FLWOR that returns the names of all countries in the *mondial* XML document having an extension area greater than 500,000 km$^2$.

```
for $country in doc("mondial")/country
   where $country/@area > 500000
order by $country/name
return $country/name
```

The FLWOR above is made up of several parts:

- the `for` clause sets up an iteration through the country nodes, and the rest of the FLWOR is evaluated once for each of the countries. Each time, a variable named `$country` is bound to a different element. Dollar signs are used to indicate variable names in XQuery;
- the `where` clause has the same effect as a predicate (`[@area < 500000]`) in a path expression;
- the `order by` sorts the results by country name, something that is not possible with path expressions;
- finally, the `return` clause indicates that the country element’s name children should be returned.

Another clause, say the `let` clause is used to set the value of a variable. Unlike a `for` clause, it does not set up an iteration. Example 2.3 shows a FLWOR that returns the same result as Example 2.2.

**Example 2.3.** The second line of the query below assigns the country’s name child to a variable called `$name`. The `$name` variable is then referenced later in the FLWOR, in both the `order by` clause and the `return` clause.

```
for $country in doc("mondial")/country
let $name := $country/name
where $country/@area > 50000
order by $name
return $name
```

These examples have been kept as simple as possible for sake of brevity. However, FLWOR expressions can become very complex. In particular, multiple `for` clauses are permitted, which set up iterations within iterations in order to easily join data from multiple sources. In addition, complex expressions can be used in
any of the clauses in order to satisfy a compositionality principle.

Besides path and FLWOR expressions, XQuery supports other functionalities. For example, XML constructors can be used to create elements and attributes that appear in the query results, in order to reorganize or transform the elements in the input documents into differently named or structured elements.

Moreover, there are over 100 functions built into XQuery, covering a broad range of functionality. Functions can be used to manipulate strings and dates, perform mathematical calculations, combine sequences of elements, and perform many other useful jobs. One can also define its own functions, either in the query itself, or in an external library. Both built-in and user-defined functions can be called from almost any place in a query.

To conclude, XQuery is a strongly typed language, meaning that each function and operator expects its arguments or operands to be of a particular type. The XQuery type system is based on that of XML Schema [110], that has built-in simple types representing common datatypes such as \texttt{xs:integer}, \texttt{xs:string}, and \texttt{xs:date} (see also Figure 3.1) where the \texttt{xs} prefix is used to indicate that these types are defined in the XML Schema specification. Types are assigned to items in the input document during schema validation, which is optional. If no schema is used, the items are untyped. As one could expect, XQuery adopts a syntax to explicitly assign a type to a variable, however, the type system of XQuery is not rigid, since there are a number of automatic type conversions.

### 2.2.3 The XQuery Data Model

Query languages are typically designed to be applied to data corresponding to a particular data model. For instance, SQL is used to create, retrieve and manipulate data represented in the relational model. Similarly, XQuery is used to create and locate (but unfortunately not update, yet) data that is represented in a model officially known as the XQuery 1.0 and XPath 2.0 Data Model, or XDM [70].

The model is used to define formally all the values used within queries, including those from the input document(s), those in the results, and any intermediate value. Its basic components are [119] (Chapter 2):

- \textit{Node}: an XML construct such as an element or attribute.
- \textit{Atomic value}: a simple data value with no markup associated with it.
- \textit{Item}: a generic term that refers to either a node or an atomic value.
- \textit{Sequence}: an ordered list of zero, one, or more items.

We provide in the following a brief overview of such components. The relationship among them is shown in Figure 2.1.
2.2. THE XQUERY LANGUAGE: AN OVERVIEW

XML Nodes

Nodes are used to represent XML constructs such as elements and attributes. They are returned by many expressions, including path expressions and constructors. For example, the path expression \texttt{doc("mondial")/country} returns 260 country element nodes.

XQuery uses six kinds of nodes: \textit{element nodes} for an XML element, \textit{attribute nodes} for XML attributes, \textit{document node} that represents an entire XML document, \textit{text nodes} to represent some character data content of an element, \textit{processing instruction nodes} to represent an XML processing instruction and, finally, \textit{comment nodes} for XML comments. Clearly, element and attribute nodes are the ones most often used within queries, since an XML document (or document fragment) is made up of a hierarchy of nodes.

Atomic values

An atomic value is a simple data value. It can have a specific type, such as \texttt{xs:integer} or \texttt{xs:string}, or it can be untyped.

An atomic value can be constructed from a lexical representation. Given a string and an atomic type, the atomic value is constructed in such a way as to be consistent with schema validation. If the string does not represent a valid value of the type, an error is raised. It can be also extracted from element or attribute nodes using data functions of XQuery or it can be created from literals in queries.
Sequences

Sequences are ordered collections of items. A sequence can contain zero, one, or many items. Each item in a sequence can be either an atomic value or a node. Sequences can be created explicitly using a sequence constructor. However, the most common way that sequences are created is that they are returned from expressions or functions. For example, the expression `doc("mondial")/country` returns a sequence of 260 items, which happen to be `<country>` element nodes.

Sequences do not have names, although they may be bound to a named variable. For example, the let clause:

```xml
let $countries as node()* := doc("mondial")/country
```

binds the sequence of `<country>` elements to the variable `$countries`. Notice that an explicit type is provided for that variable.

A sequence with only one item is known as a singleton sequence. There is no difference between a singleton sequence and the item it contains. Therefore, any of the functions or operators that can operate on sequences can also operate on items, which are treated as singleton sequences. Differently, a sequence with zero items is known as the empty sequence and it is different from a zero-length string or a zero value. Many of the built-in functions and operations accept the empty sequence as an argument, and have defined behaviour for handling it. Similarly, some expressions will return the empty sequence.

An important aspect is that sequences cannot be nested within other sequences since there is only one level of items. Said in other words, if a sequence is inserted into another sequence, the items of the inserted sequence become full-fledged items of the new sequence. For example, the sequence `[(10, (20, 30)), 40]` is equivalent to `(10, 20, 30, 40).

### 2.3 The Predictive Model Markup Language

An industry standard for actual models representation as XML documents is the Predictive Model Markup Language (PMML) [104]. PMML consists of XML schemas for a wide spectrum of models, including association rules, decision trees, clustering, naive-bayes, regression, neural networks. While PMML is becoming a primary standard, adopted by major commercial suites [54, 60, 102, 106], it is worth noting that it does not cover the process of extracting models, but rather the exchange of the extracted knowledge.

Since PMML is an XML based standard, the specification comes in the form of an XML Schema. The schema defines three main component of each mining model.

- A data dictionary containing the definitions for the fields that are used in the mining model, such as the type and the value range. These definitions are assumed to be independent from specific data sets, used for training or scoring a specific model.
• A **mining schema** that lists fields used by the model (i.e., it lists the fields which a user must provide in order to apply the model). These fields are a subset of the fields in the data dictionary. In other terms, the mining schema contains information that is specific to a certain model, while the data dictionary contains data definitions that do not vary with the model. For instance, the mining schema specifies the usage type of an attribute, the outlier treatment, the missing values replacement policy, data transformations such as discretization, normalization and so on.

• A **model description** containing the physical model and varying from a model to another.

It is important to point out here that the first two elements define the logical schema in addition to the physical one, that is maintained well-separated. They include statistical information concerning the input dataset and the parameters used to build the model. Figure 2.2 depicts an example of PMML document storing information on frequently two or more authors published together among the dblp database.

### 2.4 Summary

This chapter deals with the XML markup language. On the one side, XML is another (interoperable) representation of data (in addition to flat files, archives, relational databases, etc.). On the other side, XML is a natural candidate for representation and exchange of extracted mining models and metadata. In this manuscript we would like to go further and conceive a language and system where XML is used both for model and data representation, while (a superset of) XQuery is used for defining data and model processing tasks. Hopefully, this will be a basis for a general-purpose inductive database out of XML data.

Summing up, there are as many reasons to mine XML data as there are reasons to use (an extension of) XQuery for doing it. As we will show in the remainder of this manuscript, some advantages of using XQuery to this purpose are the following.

• It is sufficiently expressive to represent the “interactive” aspect of the KDD process, with FLOWR, conditional expressions, variables, functions and modular declarations.

• It is based on the concept of sequence and it has the compositionality as a key paradigm; this may help in representing the “multi-step” nature of the KDD process, capturing the closure principle of inductive databases.

• It is a recent W3C standard. This will favour the availability of industrial optimizers. There is a fervent activity in the research community to this purpose.
<xml version="1.0" ?>
<PMML version="3.2" xmlns="http://www.dmg.org/PMML-3_2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Header copyright="Department of Computer Science - University of Pisa"/>
    <DataDictionary numberOfFields="2">
        <DataField name="paper" optype="categorical" dataType="string" />
        <DataField name="aut-name" optype="categorical" dataType="string" />
    </DataDictionary>
    <AssociationModel functionName="associationRules"
        numberOfTransactions="12980" numberOfItems="1290"
        minimumSupport="0.01" minimumConfidence="0"
        numberOfItemsets="2030" numberOfRules="0">
        <MiningSchema>
            <MiningField name="paper" usageType="group" />
            <MiningField name="aut-name" usageType="predicted" />
        </MiningSchema>
        <Item id="1" value="Albert Einstein" />
        <Item id="2" value="Enrico Fermi" />
        <Itemset id="1" support="0.09" numberOfItems="1">
            <ItemRef itemRef="1" />
        </Itemset>
        <Itemset id="2" support="0.05" numberOfItems="1">
            <ItemRef itemRef="2" />
        </Itemset>
        <Itemset id="3" support="0.01" numberOfItems="2">
            <ItemRef itemRef="1" />
            <ItemRef itemRef="2" />
        </Itemset>
        ...
    </AssociationModel>
</PMML>

Figure 2.2: A sample of PMML document containing a set of frequent itemsets.

- It offers full-text processing [112] to the purpose, for example, of text-mining.

One capability that XQuery 1.0 does not provide is updating\(^3\), which would be particularly useful in the case of XML data stored in databases. We will deal with the XQuery limitations (and we will propose solutions) in the Section 3.3.

\(^3\)At least, there is not a standard for update in XQuery, yet. However, the W3C XQuery working group is also expanding it to include insert, update, and delete operations. Moreover, many applications offer their proprietary implementations.
Chapter 3

Foundations of an XML-Based KDD Framework

Abstract

This thesis presents the idea of a new IDB in which raw data, mining models and domain knowledge are represented by way of XML documents, stored inside native XML databases. Our starting point is the XQuery/XPath Data Model described in Section 2.2.3, extended with mechanisms to model the relevant aspects of the KDD. Strengths of the model are an understandable representation of the background and induced knowledge and the capability of satisfying the closure principle, a crucial paradigm in IDBs.

On such a data model a query language for mining XML data is defined. This language, called XQuake, is composed by high-level operators for the easy specification of mining tasks as a natural extension of XQuery. An intuitive syntax, the expressiveness and the capability of dealing uniformly with raw data, induced and background knowledge are important features of our language. This chapter includes an overview of XQuake, while in the next parts of this thesis its concrete usage is presented.

We conclude by determining what extensions are required for turning XQuery into a language supportive of the implementation of DM primitives. Such extensions serve as a base for the practical implementation of the mining tasks.

3.1 The XQuery Data Model for Data Mining

Query Languages are designed to be applied to data corresponding to a particular data model. For example, SQL retrieves, creates, modifies, and deletes data represented in the relational model, whose core concept is the tuple that well represents highly regular (i.e. structured) data. Similarly, XQuery is used to locate and retrieve data that is expressed in the XQuery Data Model (XDM) that defines the
logical structure of an XML document rather than its surface syntax. XDM is based on the concept of sequence used to represent semi-structured data (Section 2.2.3).

To pursue our objective of a new mining query language, in this section we extend XDM with new data types, accessors and functions which are related to knowledge discovery analysis. We use the acronym X(DM)$^2$ to indicate the Xquery Data Model for Data Mining. These minimal extensions do not require any modifications to the algebra and formal semantics of XDM. As a result, the existing semantics of XQuery functions need not be modified and X(DM)$^2$ is still based on the concept of sequence. A sequence is used both as input and as output of mining primitives, permitting to accomplish the closure principle of IDBs. We are going to formalize these concepts in the remainder of this section.

### 3.1.1 Input data

A *dataset* can be often viewed as a collection of *data objects*, also known as *records*, *cases* or *instances*. In turn, data objects are described by a number of *attributes* that capture the basic characteristics of an object, such as the time at which an event occurred or the age of a person. In DM, attributes are also known as *fields*, *features* or *dimensions* [63] (Chapter 2).

In mining systems based on the relational model, often, a data set is a single table in which the objects are rows (or tuples) and each column corresponds to an attribute. For example, Table 3.2 shows a data set that consists of weather conditions information used for classification purposes. Each row correspond to a day and each column is an attribute that describes, for example, the wind condition or the weather outlook for that day.

While record-based data sets are common in relational databases systems, different is the situation in XML databases. A first essential question arises.

**Question.** *How to mine datasets that are in a semi-structured (i.e. XML) format?*

Relational-oriented techniques can be applied to non-relational data by extracting features from data objects and using these features to create a set of tuples. On such an input database, traditional record-oriented mining techniques can be adopted. In DM, this kind of solution brings to a loosely-coupled architecture in which the data to be mined (e.g. XML, object-oriented) is mapped to a different model (e.g. relational), on which well-established techniques are used for mining. This is for example the strategy adopted by the XMine operator to extract association rules from XML data [23] (see Section 1.4.1).

In some case, however, it is not so easy to represent the data in a record-based format, or this type of representation does not capture all the information contained in the data. In other cases, such a representation has an high cost, that makes it impracticable. An example is the Geography Markup Language (GML) [80]. It is an XML extension that allows developers to encode geographic information for both
transfer and storage. The GML schema was written to be flexible enough to include any complex object, such as points, lines or area vectors. It is an example in which data involved do not easily fit the relational data model, while they do fit the XML data model. Trying to map such an information into a relational model may require to create a large number of different tables. As you can imagine, mining such data may require an impractical number of joins.

Summing up, since our purpose is to deal with XML data avoiding any kind of transformation of input data to other kind of representations, we bring data mining techniques to XML data. In other words, we must be able of performing mining from any XML data source, such as for example those reported in Appendix A. This implies a great effort in the sense that mining techniques must be re-adapted to directly deal with XML sources, but it brings several advantages that will be highlighted in the remainder of this thesis. For the moment, we start with a first important definition in the X(DM)² data model.

**Definition 3.1 (XML dataset).** An **XML dataset** is an ordered collection of zero or more items. Items are defined as in the XQuery Data Model.

Basically, the definition of XML dataset coincides with the definition of a sequence in XDM and, intuitively, serves as a container of a set of records. Like sequences, an XML dataset cannot be a member of another sequence, and a single item appearing on its own is modeled as a dataset with only one item. Examples of XML datasets include the sequence (‘a’, ‘b’, 123), the integer 2, the empty sequence, the collection of `<keyword>` sub-elements of the `dblp` document root.

The Definition 3.1 above leads at data in an “record-oriented” format. In this thesis, we deal with another typical kind of dataset, that is the transaction dataset. Transaction data is a special type of record data where each record (transaction) involves a set of items. A typical example of transaction is the set of products purchased by a customer, whereas the individual products are the items. Table 3.4 shows an example of transaction dataset. Each row represents the purchases of a particular customer at a particular time.

**Notation.** In the remainder we denote with $F$, $G$ and $H$ generic XQuery 1.0/XPath 2.0 expressions and with $F(s_1, \ldots, s_n)$ an XQuery/XPath expression over items $s_1, \ldots, s_n$. We also designate with $|S|$ the cardinality of the sequence $S$.

**Definition 3.2 (XML transaction dataset).** Let $R = (r_1, \ldots, r_n)$ be a sequence of zero or more items. For each $i \in [1, n]$, let $T_i = (a_{i,1}, \ldots, a_{i,m_i})$ be a sequence of XML nodes such that $T_i = F(r_i)$, where $F$ is a generic XQuery expression and $m_i \geq 0 \ \forall i \in [1, n]$. The sequence $R^{<T_1, \ldots, T_n>}$ is said XML transaction dataset. Each $T_i$ is said XML transaction and each $a_{i,j}$ is said XML item.
3.1.2 Mining attributes

In this section we address the issue of describing datasets by considering what types of attributes are used in the mining task.

The type of an attribute

In data mining knowing the type of an attribute is important, because it tells us which properties of the measured values are consistent with the underlying properties of the attribute. Therefore, either it allows to avoid foolish actions or it permits physical optimizations. For example, multiplying the temperature values with a numeric constant may be useful to change the measurement scale, but multiplying the eye’s colors with that value has not much sense. Another example is in classification tasks. Ordinal attributes can produce binary or multiway splits in a decision tree induction, since values can be grouped as long as the grouping does not violate the order property of the attribute values [63] (paragraph 4.3.3).

The system type of XQuery is capable of covering a large number of types used in the KDD field, like for example positive integer values, double values, string values, calendar date values and so on. It defines 25 basic and derived types, but they are not sufficient for describing specific DM fields, e.g. for ordered or cyclical data. A richer set of types than XQuery’s one may be available by extending XDM. In our framework, we define two additional types, including discrete and ordinal, to support specific mining tasks.

A discrete type constrains the string type to a specified set of values. Enumeration does not impose an order relation on the value space it creates; the value of the ordered property of the derived data type remains that of the data type from which it is derived. On the contrary, the values of an ordinal attribute provide enough information to order objects. Examples of ordinal fields are hardness of minerals, street numbers, \{bad, sufficient, good, excellent\}.

Definition 3.3 (discrete type). A discrete type is an atomic type defined as a restriction of the atomic string type by constraining the string value to a set of specified values.

Definition 3.4 (ordinal type). An ordinal type is an atomic type defined as a restriction of the atomic discrete type by constraining an ordering among values.

Figure 3.1 is the hierarchy of data types of XDM which is extended with mining types. On discrete and ordinal, the typical operators over the string type are defined. In addition, Table 3.1 reports constructors and operators over such types in order that the mining engine is able to operate on their instances.

Example 3.1. The evaluation of the following XQuery expression over the X(DM)² data model returns the pair (-1,0).
let $v1$ as xs:ordinalString<('cool','mild','hot')> := 'cool'
let $v2$ as xs:ordinalString<('cool','mild','hot')> := 'hot'
let $v3$ as xs:ordinalString<('cool','mild','hot')> := 'hot'
return (fn:compare($v1, $v2), fn:compare($v2, $v3)).

The definition of attribute type follows.

**Definition 3.5** (attribute type). An **attribute type** is a primitive simple type or a type derived by a restriction of another atomic type, as stated in the XDM specification. Discrete and ordinal are valid attribute types in X(DM)$^2$.

**XML attribute**

Given these definitions and notations, we can define what an XML attribute is.

**Definition 3.6** (XML attribute). Let $S = (s_1, \ldots, s_n)$ be an XML dataset and let $F$ be an XQuery/XPath expression such that $\forall i \in [1, n]$, $F(s_i)$ returns an empty
Function Syntax | Meaning
--- | ---
xs:discreteString($values as xs:string*, $arg as xs:anyAtomicType?) as xs:discreteString? | Discrete type constructor. It checks if $arg$ is in $values$

| xs:ordinalString($values as xs:string*, $arg as xs:anyAtomicType?) as xs:ordinalString? | Ordinal type constructor. It checks if $arg$ is in $values$. Order of values coincides with the order of the items in the $values$ sequence

| fn:discrete-values($arg as xs:discreteString?) as xs:string* | Returns the values of the argument

| fn:compare($arg1 as xs:ordinalString?, $arg2 as ordinalString?) as xs:integer? | Returns -1, 0, or 1, depending on whether the value of $arg1$ is respectively less than, equal to, or greater than $arg2$. If $arg1$ and $arg2$ do not share the same values, it returns an empty sequence.

Table 3.1: Basic functions and constructors on extended mining types

sequence or an atomic value in the value space of an attribute type. An XML attribute over $S$ and w.r.t $F$ is an ordered collection $A_{S,F} = (a_1, \ldots, a_n)$ where each $a_i$ is obtained by evaluating $F$ over $s_i$, i.e. $a_i = F(s_i)$. Also, an XML attribute is always labelled with a name.

We emphasise that given such a definition it can happen that $F(s_i)$ returns an empty sequence for some $i$. In this case, $|A_{S,F}| < |S|$.

**Example 3.2.** Let’s consider the playTennis dataset of Table 3.2, containing a set of tuples that describe the weather conditions suitable for playing tennis. Two different (among infinite) XML representations of such a dataset are depicted in Figures 3.2 and 3.3.

Let $S’ = (s’_1, \ldots, s’_{14})$ be the XML dataset containing the collection of all the <instance> elements of the XML document of Figure 3.2. Similarly, let $S” = (s”_1, \ldots, s”_{14})$ be the XML dataset as the collection of all the XML tags <day> of Figure 3.3. Examples of XML data attributes over $S’$ and $S”$ are:

- $A_{S’,G’} = A_{S”,G”} = ()$ by placing $F’(s’_i) = G’(s”_i) = ()$
- $A_{S’,F”} = A_{S”,G”} = (“Hot”, “Mild”, “Hot”, “Cool”, \ldots)$ by placing
  - $F”(s’_i) = s’_i[/value[2]/text()]$
  - $G”(s”_i) = s”_i[/conditions/@temperature]
- $A_{S’,F”} = A_{S”,G”} = (“Cool”, “Mild”, “Hot”, “Mild”)$ by placing
  - $F”(s’_i) = s’_i[/./value[5] eq “No”]/value[2]/text()$
  - $G”(s”_i) = s”_i[/./@play eq “No”]/conditions/@temperature$
### Table 3.2: Training set of the Play Tennis example

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Wind</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overcast</td>
<td>Hot</td>
<td>null</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>Overcast</td>
<td>null</td>
<td>Low</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>Overcast</td>
<td>Mild</td>
<td>High</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>Overcast</td>
<td>Hot</td>
<td>Low</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Cool</td>
<td>Low</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>Rainy</td>
<td>Mild</td>
<td>null</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Cool</td>
<td>null</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Mild</td>
<td>Low</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>Sunny</td>
<td>null</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Mild</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Cool</td>
<td>Low</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>Sunny</td>
<td>Mild</td>
<td>Low</td>
<td>Strong</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 3.2: An XML representation of the dataset of Table 3.2.

A role may be defined for fields. For example, it may be important to distinguished between attributes used as input to a mining algorithm and attributes that
are not required to apply a model but only providing additional information for explanatory purpose. More specifically:

- `<active>` specifies that the field is used as input to the analysis;
- `<predicted>` specifies that it is a prediction attribute, e.g. in classification tasks or in supervised discretization;
- `<supplementary>` states that it holds additional descriptive information.

**Definition 3.7** (role of an XML attribute). *Each XML attribute has a role. A role may be active, predicted or supplementary.*

In addition to a role, a weight states the relative importance of a field in the mining task. This indicator is typically used in predictive models in order to rank fields by their predictive contribution. A value of 1.0 suggests that the target field is directly correlated to that field. A value of 0.0 indicates that the field is completely irrelevant, i.e. such a field would have a supplementary rather than an active usage. Note that the weight cannot be negative and it makes sense for active fields only.

**Definition 3.8** (weight of an active attribute). *An active XML attribute is labeled with a weight, a double value in [0, 1]. If not specified, active attributes have a weight of 1.*

**Dealing with missing values**

It is not unusual for an object to be missing one or more attribute values. In some cases, the information was not collected, e.g. some people decline to give their age or weight. In other cases, some attributes are not applicable to all objects. Regardless, missing values should be taken into account during the data analysis. The relational
query language SQL supports a special value called the null value to represent data that is missing, unknown, or inapplicable. For example, Table 3.2 contains some null values in the temperature and humidity columns.

The XQuery Data Model is already designed for less regular data. XQuery functions work well when applied to data in which many values are not represented at all and in which elements and attributes may be optional and omitted. XQuery syntax is optimized for building new XML documents from one or more inherently semi-structured XML documents, easily accommodating the complete absence of some data. Summing up, dealing with missing values does not require any extension over the XDM model.

3.1.3 Context knowledge

The background knowledge, or domain knowledge, is defined as the information provided by an expert about the domain to be mined. For instance, the most common use of domain knowledge is for driving the exploitation of the relevant sub-space within the space of all possible solutions (see also Section 5.2).

Generally, such an information can be represented by means of any kind of formalism, (e.g. via the relational model, object-oriented representation or ontologies), even if, to meet the closure property of IDBs, a common representation between data, models and background knowledge has to be adopted. Obviously, such a feature also favours an uniform mechanism to express mining queries, since user can refer data, model and domain knowledge via the same declarative language.

Nowadays, however, traditional data analysis systems do not consider the semantics aspect of the input data, but rather they are primarily oriented toward the extraction of “quantitative” statistics. This inherent limitation has been highlighted in the Chapter 1, explaining three different mining systems. Among the studied systems, only LDL++ [41] offers an in-depth analysis of background knowledge, considering its encapsulation inside the mining process. On the other hand, at the present time, academic and industrial well-established KDD environments offer a plethora of “quantitative” features but none of them is able to replay to a “semantics” mining query on the background knowledge. So, an important objective of our project is to capture what is known about the problem from whatever sources of information are available.

However, domain knowledge is strictly dependent from the task, and hence difficult to define and to integrate with data and mining models. To address this challenge, our DM system has to be equipped with a substantial amount of background knowledge in the form of XML sequences. The context knowledge is encapsulated in a way that makes it available in a useful form for the mining algorithm. In the remainder of this manuscript, we refer with the term of “metadata” at the domain knowledge of interest, intended as a descriptive information about the context.
Definition 3.9 (XML metadata attribute). Let $S = (s_1, \ldots, s_n)$ be an XML dataset and let $F$ be an XQuery/XPath expression such that $\forall i \in [1, n]$, $F(s_i)$ returns an XML item. An XML metadata attribute over $S$ and w.r.t $F$ is an ordered collection $A_{S,F} = (a_1, \ldots, a_n)$ where each $a_i$ is obtained by evaluating $F$ over $s_i$, i.e. $a_i = F(s_i)$. An XML metadata attribute is labelled with a name. □

Prior knowledge can be used in different ways to constrain the search space. As a case study, this thesis presents in Section 5.2 how the background knowledge can be encapsulated as deeply as possible into the extraction process of a frequent pattern task. From the one side this helps the analyst in the specification of the most significant patterns. From the other side this improves the performance of the mining.

3.1.4 Mining models

As mentioned in Section 1.2, in order to comply with the closure principle of IDBs, mining languages represent the extracted knowledge by means of an uniform paradigm. For example, SQL-based query languages like Mine Rule [27] or MSQL [57] represent source data and induced models as database relations.

Also X(DM)$^2$ proposes a model in which the mining models are represented as XML documents. In particular, to describe the high-level information, PMML models (Section 2.3) are used. This seems an appropriate choice due to the fact that PMML is becoming a primary standard for representing models as XML documents, adopted by major commercial suites. Figure 3.1 shows the PMML extension over the XDM system type, in which a PMML model is a single item in the value space of a PMML type.

Definition 3.10 (PMML document). A PMML type is a type defined as a restriction of a document type. A tree whose root node is a PMML node is referred to as a PMML document. □

Table 3.3 summarizes how the data mining entities are represented in X(DM)$^2$. In the next section, a mining language based on such a data model is presented.

3.2 The XQuake philosophy

XQuake - acronym of XQUery-based Applications for Knowledge Extraction - adopts an XQuery-like syntax to facilitate high level DM and in-deep integration within native XML databases. Essentially, XQuery expressions are used to identify XML datasets as well as mining fields and metadata, to express constraints on the domain knowledge, to specify user preferences and the format of the XML output. It is built over the XQuery Data Model for Data Mining earlier discussed.
3.2. THE XQUAKE PHILOSOPHY

<table>
<thead>
<tr>
<th>DM Entity</th>
<th>Relational Model</th>
<th>X(DFM)$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data set</td>
<td>Set of tuples</td>
<td>XML dataset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Definitions 3.1)</td>
</tr>
<tr>
<td>Field</td>
<td>Column</td>
<td>XML attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Definitions 3.6, 3.7 and 3.8)</td>
</tr>
<tr>
<td>Metadata</td>
<td>Relational table(s)</td>
<td>XML metadata attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Definition 3.9)</td>
</tr>
<tr>
<td>Mining model</td>
<td>Relational table(s)</td>
<td>PMML Document</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Definition 3.10)</td>
</tr>
</tbody>
</table>

Table 3.3: Data mining entities representation in X(DFM)$^2$.

A mining query begins with a collection of XQuery functions and variable declarations followed by an XQuake operator. The syntax of each operator includes four basic statements that specify four major activities in DM:

1. task and method specification;
2. domain entities identification;
3. exploitation of constraints and user preferences;
4. output construction.

The outline of a generic operator is explained below.

### 3.2.1 Task and method specification

Each XQuake operator starts specifying the kind of KDD activity.

- The `prepare` keyword indicates operators for data preprocessing;
- `mine` includes operators to extract a mining model;
- `apply` is used for modeling the application to the input data;
- `filter` is used to select a subset of the extracted knowledge according to a condition.

Several other kinds of constructs can be conceived to deal with model evaluation or model meta-reasoning. For example, the following XQuake fragment denotes a data sampling task:

```xquery
prepare sampling doc("my-out") using alg:my-sampling-alg(my-params ...) ...
```
The doc("my-out") expression directs the result of the mining task to a specific native XML database for further processing or analysis. The using statement indicates the kind of mining or preprocessing algorithm used, together with a set of basic parameters for the mining, such as the minimum support threshold for association rule mining algorithms or the confidence for pruning in classification tasks.

3.2.2 Domain entities identification

As stated in Section 3.1, any KDD task may need to specify the set of relevant entities that are input to the analysis. They include the physical data sources and the mined patterns, as well as logical elements such as the type of the attributes or the domain knowledge. The language offers a powerful and versatile way to locate the input objects of a mining task. The syntax is an adaptation of the standard XQuery FLWOR syntax, in which the result of the evaluation of an expression is linked to a variable in for and let clauses.

Data location, filtering and specification of mining attributes

We start with the data specification. Below is a simple fragment statement that can be used to locate training instances in a clustering or classification activity (we use a different font to highlight an XQuery expression). XQuake also offers an optional WHERE statement, used to specify data filtering constraints. The user specifies them through an XQuery condition, that is typically processed before the mining task.

\[\ldots \text{for data } \$my-tuple \text{ in } < \text{XQuery expression } > \]
\[\text{where } < \text{XQuery expression } > \ldots\]

As stated in the Definition 3.1, input data are typically a sequence of XML nodes. The FOR expression above binds the variable \$my-tuple to each item during the evaluation of the operator.

In the fragment of query below, we extend our definition by including an XML attribute over the \$my-tuple data.

\[\ldots \text{for data } \$my-tuple \text{ in } < \text{XQuery expression } > \]
\[\text{let active field } \$my-field := < \text{XQuery expression on } \$my-tuple > \ldots\]

In this case, the LET clause defines a data attribute whose name is \$my-field, whose values are obtained by means of an XQuery expression and whose type is omitted. We emphasise that, as required by Definition 3.6, values of the field are in the value space of an atomic mining type, e.g. numeric types, discrete types or ordinal types. The keyword after the LET refers to the role of such an attribute in the mining activity of interest (Definition 3.7). Other values are PREDICTED and SUPPLEMENTARY.
3.2. THE XQUAKE PHILOSOPHY

Example 3.3. Consider the playTennis Example 3.1 and in particular the dataset of Figure 3.3. The following query fragment specifies each <day> element as a record of the XML dataset, and each value of the @temperature attribute, for which we are not able of playing tennis, as XML attribute value.

\[
\text{... for data } \$\text{day in } \text{doc("weather")}/\text{day} \\
\text{let active field } \$\text{no-play := } \$\text{day}[./@\text{play eq "No"}]/\text{conditions/@temperature} \\
\text{...} \quad \triangleleft
\]

Example 3.4. In the dblp dataset of Figure A.1, we are interested in the specification of a mining attribute that indicates whether a journal paper published by the “ACM” focusses on the KDD field. More specifically, ACM journal papers are the input XML sequences and a boolean attribute, $\text{has-kdd-keyword}$, encapsulates the required information.

\[
\text{... for data } \$\text{paper in } \text{doc("dblp")//article[fn:contains(./journal, "ACM")]} \\
\text{let active field } \$\text{has-kdd-keyword := } \\
\quad \text{some } \$\text{keyword in } \$\text{paper/keywords/keyword satisfies } \$\text{keyword eq "KDD"} \\
\text{...} \quad \triangleleft
\]

XML transaction datasets (Definition 3.2) can be queried in a similar way. The problem is to bind a variable to each XML transaction and a new variable to each XML item of that transaction.

Example 3.5. Let us consider the transaction dataset of Table 3.4, and the correspondent XML representation of Figure 3.4. The following query fragment binds each XML transaction to the variable purchase and each XML item to the variable product. An additional attribute product-cost captures the descriptive information of the items.

\[
\text{... for data } \$\text{purchase in } \text{doc("purchases")/purchase} \\
\text{for item } \$\text{product in } \$\text{purchase/item} \\
\text{let active field } \$\text{product-name := } \$\text{product/text()} \\
\text{let supplementary field } \$\text{product-cost := } \$\text{product/@price } \times \$\text{product/@qty} \\
\text{...} \quad \triangleleft
\]

Notice the use of a special syntax to identify the items of each transaction, i.e. the products purchased by a customer.

Example 3.6. Notice the use of a special syntax to identify the items of each transaction, i.e. the products purchased by a customer.

\[
\text{Notice the use of a special syntax to identify the items of each transaction, i.e. the products purchased by a customer.} \quad \triangleleft
\]

Adding logical information to attributes

XQuake maintains the typing philosophy of XQuery by offering a method to optionally equip attributes with logical information. In the example below, an explicit type is provided by the user for the field $\text{has-kdd-keyword}$.
<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
<th>Prices</th>
<th>Quantities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a,b}</td>
<td>{10,20}</td>
<td>{5,10}</td>
</tr>
<tr>
<td>2</td>
<td>{b,c,d}</td>
<td>{20,30,40}</td>
<td>{1,2,5}</td>
</tr>
<tr>
<td>3</td>
<td>{a,c,d,e}</td>
<td>{10,30,40,50}</td>
<td>{4,2,10,2}</td>
</tr>
<tr>
<td>4</td>
<td>{a,d,e}</td>
<td>{10,40,50}</td>
<td>{1,10,5}</td>
</tr>
<tr>
<td>5</td>
<td>{a,b,c}</td>
<td>{10,20,30}</td>
<td>{4,8,1}</td>
</tr>
<tr>
<td>6</td>
<td>{a,b,c,d}</td>
<td>{10,20,30,40}</td>
<td>{5,10,15,20}</td>
</tr>
<tr>
<td>7</td>
<td>{a}</td>
<td>{10}</td>
<td>{5}</td>
</tr>
<tr>
<td>8</td>
<td>{a,b,c}</td>
<td>{10,20,30}</td>
<td>{10,4,2}</td>
</tr>
<tr>
<td>9</td>
<td>{a,b,d}</td>
<td>{10,20,40}</td>
<td>{8,6,10}</td>
</tr>
<tr>
<td>10</td>
<td>{b,c,e}</td>
<td>{20,30,50}</td>
<td>{5,10,1}</td>
</tr>
</tbody>
</table>

Table 3.4: A transaction dataset over the market basket data.

```xml
<purchases>
  <purchase>
    <item price="10" qty="5">a</item>
    <item price="20" qty="10">b</item>
  </purchase>
  <purchase>
    <item price="20" qty="1">b</item>
    <item price="30" qty="2">c</item>
    <item price="40" qty="5">d</item>
  </purchase>
  ...
  <purchase>
    <item price="20" qty="5">b</item>
    <item price="30" qty="10">c</item>
    <item price="50" qty="1">e</item>
  </purchase>
</purchases>
```

Figure 3.4: A possible XML representation of the dataset of Table 3.4.

... let active field $has-kdd-keyword as xs:boolean := ...

Either if an input field has an explicit or an implicit type, it is validated against a required type that depends on the context in which it appears. For instance, the target attribute of a classification task is required to be discrete. An error is raised whenever the type of an expression does not match the expected type. The example of query fragment below explicitly defines the type of the declared data attributes.

**Example 3.6.** We aim at using the **playTennis** dataset of Figure 3.3 in a tree classification task. Training instances are all the `<day>` XML elements. The target attribute is the attribute **@play**. Active data attributes are the **@outlook**, **@temperature**, **@humidity**, and **@wind**.
3.2. THE XQUAKE PHILOSOPHY

@temperature, @humidity, and @wind XML attributes located in the <conditions> sub-element.

... for data $day in doc("weather1")/day
    let active field $outlook as xs:discrete<"Sunny", "Overcast", "Rainy"> :=
        $day/conditions/@outlook
    let active field $temp as xs:ordinal<"Cool", "Mild", "Hot"> :=
        $day/conditions/@temperature
    let active field $hum as xs:discrete<"Low", "High"> := $day/conditions/@humidity
    let active field $wind as xs:discrete<"Weak", "Strong"> := $day/conditions/@wind
    let predicted field $play as xs:discrete<"Yes", "No"> := $day/@play

... \(\triangleright\)

In the example above, all the fields have a discrete type, except for the temperature attribute that admits an ordinal type with values ‘Cool’ < ‘Mid’ < ‘Hot’ in the predictive task. In general, as stated in the X(DM)\(^2\) specification, each item value having as type a discrete or ordinal type possesses all of the properties and operations of the string type above it. As mentioned, knowing the type of a field at this level is important, since we can adopt ad-hoc optimizations during the construction of a predictive tree based on ordinal or discrete attributes (see Section 4.3).

Another important aspect in defining logical attributes concerns missing values. In XQuake, sequences of length zero may be used to represent missing or unknown information (i.e. missing XML elements or attributes), in much the same way that null values are used in relational systems. The specific mining algorithm will take care of handling such missing information according to the targets of the analysis. For instance, in the Example 3.6 above, we have specified two discrete fields, $temp and $hum, that admit missing values if the homonym XML attribute is absent.

In the example below, we show how values of a discrete field are derived from a data dictionary associated to a dataset, rather than being defined in an explicit way.

**Example 3.7.** Our aim is to define a classification task similar to Example 3.6, but applied to the XML dataset of Figure 3.2. The user-defined XQuaery function below facilitates the task.

```xml
declare function local:values($a as node()) as xs:string{
    if not(($a/@type eq "nominal") or ($a/@type eq 'ordinal'))
        then ()
    else for $i in $a/labels/label
        return $i/text()
};
```

It extracts the values of the given discrete attribute. Such a function may be used in the specification of the mining task below, as any other XQuaery expression.
... for data $i in doc("weather2")/body/instance
let active field $outlook as
   xs:discrete<local:values(/attribute[@name eq "outlook"])> :=
   $i/value[1]/text()
...
let predicted field $play as
   xs:discrete<local:values(/attribute[@class eq "yes"])> :=
   $i/value[last()]/text()
...

The syntax of the language also allows the specification of other kinds of qualifiers
for an attribute, for example in order to assign a weight before the construction of
a model, as stated in the Definition 3.8.
... let active field $my-weighted-field weight 0.75 as $xs:decimal := ... 

Domain knowledge location

XQuake admits a special syntax to specify domain knowledge, which is particularly
useful for the definition of domain-based constraints. As stated in the Definition
3.9, in contrast to active and predicted mining fields, a metadata attribute may also
include non-atomic types, such as XML nodes or attributes. For example, below we
assign an hypothetical XML hierarchy to a table column as a metadata information.

... for data $country in doc("mondial")//country
let metadata field $cap-hier :=
   let $capital := $country//city[@is-capital="yes"]
   return doc("hierarchy")/root/city[.=|$capital|
...

For each distinct <country> element of the mondial dataset (Figure A.3), the META-
DATA keyword defines a special field used to bind domain knowledge (an XML tax-
onomy in this case) to the capital of that country.

Table 3.5 provides a summary of the mining fields as defined by the XQuake
language, including their type, usage and information whether that attribute admit
missing or weighted values.

Mining model location

From the mining models perspective, a similar syntax may be used to locate (parts
of) a (new or extracted) PMML model.
... for tree $my-tree-model in doc("my-tree-model")
let metadata field $my-mining-schema as xs:node() := $model/MiningSchema...
In the example above, the construct binds the mining schema XML element of the my-tree-model document to the variable $my-mining-schema$. The type of the variable and the location of such an element are explicit.

**XQuake facilities**

The toy dataset reported in Table 3.2 admits only a small number of attributes and instances. Consider instead a large collection of objects, each of which has a large number of attributes of several different sorts. This is a typical situation in data mining. To support this case, XQuake introduces a facility to group those attributes into a single field specification. This is achieved by means of the definition of a LET GROUP clause.

**Example 3.8.** The classification task of Example 3.7 can be more compactly expressed by means of the following query fragment.

...  
for data $i$ in doc("weather2")/body/instance  
let group<$4 > field $weather-attrs as xs:string* :=  
    for $j$ in (1 to 4) let $value := $i/value[$j]/text()  
    return if empty($value) then "?" else $value  
let predicted field $play$ as  
    xs:discrete<local:values(//attribute[@class eq 'yes'])> :=  
        $i/value[last()]/text()  
...  

The usage of the GROUP keyword in the LET clause avoids the specification of each single field, since it groups into a string sequence, $weather-attrs$, every value of interest (in this case the 4-tuple containing the values of the attributes outlook, temperature, humidity and wind). If one of such attributes contains a missing value for a given record, a special symbol is returned as output, to handle the missing information.

---

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Description</th>
<th>Type</th>
<th>Missing Values</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>used as input of the analysis</td>
<td>primitive</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Predicted</td>
<td>prediction field</td>
<td>primitive</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Supplementary</td>
<td>holding additional descriptive info</td>
<td>primitive</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Metadata</td>
<td>used to contain domain knowledge</td>
<td>any</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 3.5: Mining field types in XQuake.
The introduction of this facility does not require any modification to our data model since a group attribute of length $k$ and type $t^*$ is equivalent to a set of $k$ active fields of type $t$, if we constrain the group to contain the same number of elements for each data item. Otherwise, an error is raised at run-time. In the example, the special symbol “?” is adopted to accomplish with missing information cases. Such a special symbol is treated as an empty sequence by an XQuake system module before the data mining step. In Chapter 6 we will show how a group attribute is treated from the physical point of view.

3.2.3 Exploitation of constraints and user preferences

Generally speaking, the language must allow complex querying on the domain knowledge as well as an easy and elegant way to express user preferences. The specification of a mining constraint acts by means of an \texttt{HAVING} clause, whose syntax and meaning strictly depends on the mining task. As an example, for the frequent itemsets mining task, XQuake defines a special clause that allows the definition of domain-dependent constraints on the format of the input itemsets (Section 5.2).

The \texttt{HAVING} statement is also used to specify user preferences, for example the bins of a discretization technique (Section 4.1).

3.2.4 Output construction

A complete mining language needs a facility for constructing and customizing the output results, according to the focus of the task. XQuake offers a \texttt{RETURN} clause to accomplished this task with element and attribute constructors, as they are an extremely common way to produce XML output in XQuery.

As mentioned before, we permit a customized output only for operators returning XML data (e.g. preprocessing operators), since we fix the output of the other kinds of operator to be a PMML model. This seems an appropriate choice for an high number of models, because, typically, their nested structure is complex and hard to specify.

3.2.5 Summary

We presented an overview of the mining language XQuake and the data model on which it operates. A crucial issue is whether the XQuake language is powerful enough for expressing the most significant inductive schemas. In other words,

\textbf{Question.} \textit{“Given an inductive database formalized by $Th(\mathcal{L},r,q)$, an XQuake query can be used to define the predicate $q$ that populates $Th$ and that defines the format of any valid pattern $s \in \mathcal{L}$”}?
In this thesis, we will use XQuake operators as the means to introduce mining primitives and to realize the notion of inductive database. We will show how the statement above has a positive answer for some important data mining techniques and, more generally, how the notion of IDB fits naturally in XML-based languages. We chose the most basic KDD tasks and a couple of fundamental methodologies in each category: the equal-frequency and equal-width methods for discretization, the random and stratified sampling, the Apriori [1], FP-Growth [50] and ID3 [86] (Chapter 3) algorithms. Moreover, the extracted models can be applied to “new” instances to predict features or to select data according to the knowledge stored in the model.

Summing up, the syntax of the clauses shown in this chapter is reported in Figure 3.5, where ExprSingle is a generic XQuery expression. The definition of each operator will be specialized in the next two chapters, in which, XQuake operators for the frequent patterns discovery and the classification task are presented. We also report in Table 3.6 a set of XQuery functions (and their meaning) defined over the XQuake constructs discussed in this paragraph. How such XQuery functions are concretely implemented will be explained in Chapter 6.

The next section is devoted to determine what extensions are required for turning XQuery into a language supporting the implementation of mining primitives.

<table>
<thead>
<tr>
<th>Function Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>core:for-data() as node()*</td>
<td>Evaluates the FOR DATA clause</td>
</tr>
<tr>
<td>core:for-data-where() as node()*</td>
<td>Evaluates the FOR DATA clause including the WHERE statement</td>
</tr>
<tr>
<td>core:where($t as node()) as xs:boolean</td>
<td>Evaluates the WHERE expression for the given tuple</td>
</tr>
<tr>
<td>core:num-tuples() as xs:int</td>
<td>Returns the number of instances of this dataset</td>
</tr>
<tr>
<td>core:num-active() as xs:int</td>
<td>Returns the number of active fields in the statement</td>
</tr>
<tr>
<td>core:unroll-items($t as node()<em>) as xs:string</em></td>
<td>Returns the atomic items given an XML transaction</td>
</tr>
<tr>
<td>core:let-active($t as node()*, $i as xs:int) as xs:anyAtomicType?</td>
<td>Evaluates the LET ACTIVE clause of index $i over the XML transaction $t</td>
</tr>
</tbody>
</table>

Table 3.6: Some useful XQuery functions of the core package.

3.3 On the implementation of the mining tasks

In Section 1.2 we explored two orthogonal solutions to the design of a mining architecture. The more database-aware applications use loosely-coupled approaches to fetch data records as needed by the mining algorithm. The core of the algorithm is typically implemented in a host programming language. On the contrary,
CHAPTER 3. FOUNDATIONS OF AN XML-BASED KDD FRAMEWORK

XQuakeMain ::= VarDecl* FuntionDecl* XQuakeOp
XQuakeOp ::= (PreprocessingOp | MiningOp | ModelApplicationOp | ModelFilterOp)

PreprocessingOp ::= "prepare" (DiscrOp | SamplingOp)
MiningOp ::= "mine" (MineItemsetsOp | MineTreeOp)
ModelApplicationOp ::= "apply" ApplyItemsetsOp
ModelFilterOp ::= "filter" FilterItemsetsOp

DiscrOp ::= ...
SamplingOp ::= ...
MineItemsetsOp ::= ...
MineTreeOp ::= ...
ApplyItemsetsOp ::= ...
FilterItemsetsOp ::= ...

UsingClause ::= "using" "alg:" QName "(" (ExprSingle ("," ExprSingle)*)? ")"

ForDataClause ::= "for" "data" "$" VarName TypeDeclaration? "in" ExprSingle
WhereClause? LetFieldClause*

LetFieldClause ::= "let" ("active"|"predicted"|"metadata"|"supplementary"|GroupField)
                "field" "$" VarName TypeDeclaration? WeightDeclaration? ":=" ExprSingle
GroupField ::= "group" "<" {positive integer} “>"
WeightDeclaration ::= "weight" Number
WhereClause ::= "where" ExprSingle

ForTransactionDataClause ::= "for" "data" "$" VarName TypeDeclaration? "in" ExprSingle
 "for" "item" "$" VarName TypeDeclaration? "in" ExprSingle
WhereClause? LetFieldClause*

ForModelClause ::= "for" PMMLModel "$" VarName "in" XQuakeOp LetFieldClause*
PMMLModel ::= ("tree" | "association")

ReturnClause ::= "return" ExprSingle

Figure 3.5: General syntax of XQuake.

tightly-coupled techniques, instead of bringing the records of the database into the application program, selectively push parts of the application program that perform the mining computation into the database.

The XQuake system architecture is designed directly over native XML databases, and it uses a compromise between the two solutions: we define and implement the mining operations via (extended) XQuery programs, while maintaining the capability of implementing some complex functionalities by means of a procedural language. As shown below, this strategy requires some adaptations of the XQuery language
for DM purposes. On the other hand, it allows the exploitation of domain-specific constraints in the right point of the mining process, since constraints are expressed (and evaluated) directly within the “XQuery view”, that is the core of the mining operator. Moreover, complex operations over data structures are implemented in an imperative language and the input/output of such operators is integrated within the XQuery program by external functions. This favours the implementation of all the cases in which maintaining data structures directly encoded in XML is both inflexible and inefficient.

The purpose of this section is to expand XQuery to support a better implementation of mining primitives. The goal is to have a powerful extension that is appropriate for all use cases, including preprocessing and mining algorithms. The system architecture of XQuake will be presented in Chapter 6.

### 3.3.1 Motivation

The capability of iterating on data is the basis of KDD algorithms. Many basic algorithms operate in the same way: (i) some variables are initialized to contain a set of statistical indicators; (ii) for each single tuple the indicators are updated according to the purpose of the task; (iii) when no more tuples are available, a termination condition is evaluated on such variables to determine whether they compute all the patterns or no further statistics can be extracted. Typically we need to repeat this process more than once, evaluating the termination condition at each iteration and updating statistics for each input tuple. More specifically, this methodology can be summarized by means of an iterative program over an input dataset \( D \):

\[
S := \text{init}(); \\
\text{do } \{ \\
\quad \text{for each tuple } t \text{ in } D \{ \\
\quad \quad S := f(S, t); \quad \quad (\star) \\
\text{\}} \} \text{ while}(g(S)); \\
\text{return } h(S);
\]

where \( \text{init}, f, g \) and \( h \) are user-defined functions.

The above iterative schema is common to many algorithms for discovering frequent itemsets and sequential patterns, and for classification and clustering. For example, at each iteration of the Apriori algorithm, new candidates are computed by combining the frequent itemsets extracted at the previous iteration and removing the infrequent ones. The cycle terminates when the set of new candidate itemsets is empty. The variable \( S \) refers to a complex data structure that stores the support counts of the itemsets.

Several variants are possible. For example, we can repeat the \text{init} function at each iteration of the external loop, or we can replace the \text{for} loop with a windowing
computation, in which the overall dataset is split into data windows (i.e. subsets of contiguous tuples). For each window, an output item is returned as a result of the computation. This feature is particularly useful for preprocessing tasks (e.g. sampling, discretization, etc.), in which, typically, the output is a new tuple for each (window of) tuple(s).

The above considerations bring up an interesting question: is XQuery capable of implementing a computation like (⋆)? By exploiting recursion and the nesting property of XQuery, it is easy to define the external do-while cycle directly as an XQuery program. More complicated is the definition of the internal loop. In fact, the main deficiencies of the traditional XQuery FLWOR expression are twofold. On one side, it is difficult to define queries over windows of data and this complicates the implementation of data preparation tasks. On the other side, also the managing of local temporary variables in an iterative computation is not straightforward. Consider for example the following simple FLWOR expression.

\[
\text{let } i := 0 \\
\text{for } j \text{ in } (1,2,3) \\
\text{let } i := i + j \\
\text{return } i.
\]

Users new to XQuery sometimes expect that one evaluation of the body of a for expression affects later evaluations, as it would in imperative programming languages. So, they expect (1, 3, 6) as the result. The semantics of the for expression in XQuery makes each evaluation independent from every other evaluation, i.e. the environment at each iteration is the same (except for the binding loop variable $j$). As a consequence, the “surprising” result of the evaluation is (1, 2, 3). This has the implication that it becomes difficult to implement a computation like (⋆) in an intuitive way, since the value of the variable $S$ needs to be affected from previous iterations.

In order to implement data-oriented tasks, we add to XQuery a new iterative construct, namely wfor, as an alternative to standard FLWOR expression and capable of overcoming the limitations reported above. We integrated such an extension into a Java-based open source XQuery engine (see Section 6.1).

### 3.3.2 Run-through examples

The wfor statement iterates over an input sequence and it binds a variable with every iteration. In addition, the statement permits us to specify a declare-init-iterate-return computation like for data cursors. A complete syntax is given in Figure 3.6a.

The declare clause declares and initializes a local variable with a syntax similar to the homonym statement of XQuery. The effect is to introduce the variable named
3.3. ON THE IMPLEMENTATION OF THE MINING TASKS

by the declare - say state variable - and to initialize it with the value of the
given expression. The state variable is in the scope of all the rest of the wfor
expression, i.e. for the iterate, return as well as for the init and while clauses
when present. The state variable is updated at each iteration with the result of the
iterate clause, whose aim is to consume the sequence item by item. Finally, the
return clause returns an output value of the overall computation. Observe that
any XQuery expression can be used in the init, iterate and return statements,
including FLWOR expressions or nested wfor expressions. This feature allows one to
iterate over an input sequence several times.

Example 3.9 (average absolute deviation). In order to compute the average
absolute deviation of a sequence of numeric values, $seq$, the following query can be
used.

Figure 3.6: Syntax (a) and intuitive semantics (b) of the wfor iterator.
Two scans are performed over the input sequence. During the first iteration (external \texttt{wfor}), we calculate the mean value. The \$value1 variable is bound to each item of the input sequence. The state variable, \$m, stores the temporary result of the computation and contains the current sum and the number of visited elements. \texttt{declare} initializes the total sum and the count to 0. \texttt{iterate} updates \$m with the value resulting from the evaluation of the body expression and then consumes an item of the sequence until no further items are available. \texttt{return} terminates the first scan of data by computing the mean of the items. During the second iteration (internal \texttt{wfor}), for each item the value of the state variable is increased with the absolute difference between that item and the mean. \texttt{return} closes the computation returning the final result (a single value in this case) as answer.

In order to specify more complex computations, we permit the specification of simple tumbling window queries\footnote{In tumbling windows items do not overlap. There are other different kinds of windows, i.e. \textit{sliding windows} and \textit{landmark windows}. The state-of-the-art in extending XQuery with window functions is out of the scope of this thesis. We refer the interested reader to [15].} by adding an \texttt{while} clause and an \texttt{init} clause (see Figure 3.6a). Basically, the first one permits (i) to break the execution of \texttt{iterate} and (ii) to anticipate the evaluation of \texttt{return} by returning a new value as output. At this point, if additional items exist in the input sequence, the computation continues by opening a new data window and re-initializing the state variable with the result of the evaluation of the body of the \texttt{init} statement.

**Example 3.10** (on-line mean). The goal is to compute the on-line mean of a numeric sequence, \$seq, i.e. we define a query that returns the mean every 100 values count:

\begin{verbatim}
for $value in $seq
    declare variable $state as xs:int* := (0, 0)
    init (0, 0)
\end{verbatim}

In this example, a new window is opened at each evaluation of \texttt{init} by re-initializing the state variable every time. \texttt{iterate} updates \$state as usual and then consumes a new item. Whenever the \texttt{while} clause holds - i.e. the number of items read is 100 - \texttt{return} terminates the computation over the current window and it returns the
mean value. If additional items of the sequence $\text{seq}$ exist, the iteration continues by evaluating $\text{init}$ over a new window of values, until the items of the input sequence are entirely consumed.

Intuitively, the $\text{while}$ specifies when the $\text{return}$ clause should be evaluated and a new value returned as output. In the previous example, the $\text{while}$ clause partitions the input sequence into windows of exactly (if any) 100 elements, so one value is returned every 100 items read. If $\text{while}$ is true or absent then $\text{return}$ is invoked only one time and a single value is returned as output. The usage of the $\text{init}$ clause states that its body expression is evaluated every time a new window is created, thus re-initializing the state variable. When the $\text{init}$ is absent, the operator assumes a different semantics.

**Example 3.11** (cumulative mean). Given a numeric sequence, we aim at computing, for each item, the mean of all values in the current sequence up to and including the current item. For example, if $\text{seq}$ is the sequence (10, 20, 30, 40), the output sequence is (10, 15, 20, 25). The $\text{wfor}$ expression is as follows.

\[
\text{wfor } \$\text{value} \text{ in } \$\text{seq} \\
\quad \text{declare variable } \$\text{state} \text{ as } \text{xs:int*} := (0,0) \\
\quad \text{iterate } (\$\text{state}[1] + \$\text{value}, \$\text{state}[2] + 1) \text{ while } \text{fn:false()} \\
\quad \text{return } \$\text{state}[1] \text{ div } \$\text{state}[2].
\]

The intuitive semantics of the $\text{wfor}$ iterator can be better explained by means of the automaton depicted in Figure 3.6b. The formal semantics is reported in Section 3.3.5.

### 3.3.3 $\text{wfor}$ for data mining

In the following, we provide some example on how the $\text{wfor}$ operator can facilitate the definition of DM tasks.

**Example 3.12.** Consider from the *mondial* dataset the collection of the elements $<\text{country}>$. For each country, we know the name of the most pursued religion in that country, and the number of persons that pursue such a religion in that country. As stated in the Example 1.2, we aim at detecting the religions in the dataset pursued by an average number of persons greater than a given threshold.

This task can be easily implemented by first ordering the data according to the $<\text{religions}>$ element and then by using a $\text{wfor}$ expression to collect the mean for each distinct value of such an element.

\[
\text{let } \$\text{ordered-countries} := \text{for } \$i \text{ in doc("mondial")/country} \\
\quad \text{order by } \$i/\text{religions}[1] \text{ return } \$i
\]
The \texttt{wfor} command above creates a data window for each distinct item whose mean has to be computed. On such a data window, the \texttt{collection} variable stores a triplet of values: (item name, current sum, current count). Such a variable is updated in the \texttt{iterate} clause until a different item is found. When this happens, the \texttt{return} statement computes the required mean and returns the pair (item name, mean) as answer if the mean value is greater than the user-defined threshold.

\begin{example}
A different solution to the problem stated in the previous example makes use of an XML-encoded structure that stores the triplet of XML attributes (@name, @current-count, @current-sum) for each distinct religion.

\begin{verbatim}
<distinct-items>
  <item name="Roman Catholic" current-count="10" current-sum="7856400"/>
  <item name="Greek Orthodox" current-count="19" current-sum="6700000"/>
  ...
  <item name="Protestant" current-count="2" current-sum="550000"/>
</distinct-items>
\end{verbatim}

Such a data structure “simulates” an Hashtable in imperative languages, since it maps keys (the religion name) to values (the pair sum, count). It is extended with a new \texttt{<item>} element whenever a new item value is encountered. We can specify the following functions over such a data structure:

\begin{itemize}
  \item \texttt{local:append($ds as node(), $n as xs:string, $v as xs:int) as node()} that inserts a new \texttt{<item>} element in this data structure with given name, $\texttt{n}$, and sum, $\texttt{v}$, and setting the \texttt{@current-count} attribute to 1. The function returns the structure itself.
\end{itemize}
3.3. ON THE IMPLEMENTATION OF THE MINING TASKS

- `local:update($ds as node(), $n as xs:string, $v as xs:int) as node()` that updates the `<item>` element in $ds$ having the given name $n$ by increasing the sum with the value $v$ and by increasing the number of elements of 1. It returns $ds$ itself.

These simple functions illustrate the need for expressions that have a side-effect (the updating of the $ds$ structure) and also return a value (the structure itself) [39]. Given such operations, the implementation of the task becomes straightforward by using a `wfor` iterator.

```
wfor $c$ in doc("mondial")/country
  declare variable $ds$ as xs:node() := <distinct-items/></distinct-items>
  iterate let $name$ = $c$/religions[1]/text()
    let $v$ := $c$/religions[1]/@percentage * xs:int($c$/population)
    return if (some $j$ in $ds$/item/@name satisfies $j$ eq $name$)
      then local:update($ds$, $name$, $v$)
      else local:append($ds$, $name$, $v$)
  return for $item$ in $ds$/item
  let $mean$ := $item$/@current-sum div $item$/@current-count
  return if $mean$ > 100000 then ($item$/@name, $mean$) else ()
```

Data collections in KDD often contain a time series, that is a sequence of observations which are ordered in time. For example, measuring the height of the Piazza S. Marco in Venice above the sea level produces a time series. Inherent in the collection of data taken over time is some form of random variation. Methods do exist for reducing or eliminating the effect due to random variation. Widely used are smoothing techniques [85] (Chapter 9).

**Example 3.14** (median smoothing). Given an input series $X_1, \ldots, X_N$ and a positive integer $k$, the median smoothing strategy consists of replacing $X_i$ by:

$$X_i^* = \text{median}(X_{i-k}, \ldots, X_{i+k}), \text{ for } i = k + 1, \ldots, N - k.$$

```
wfor $v$ at $k$ in $X$
  declare variable $window$ as xs:double* := ()
  init if (empty($window$)) then () else fn:remove($window$, 1)
  iterate ($window$, $v$) while $pos < (2 * $k$) + 1
    return local:median($window$).
```

As $window$ moves through the sequence, the oldest item value is discarded (init clause) and a new one is added (iterate clause). The median smoothing strategy uses the median of the values in the window to replace the current value (return clause). The while condition ensures that the median value is computed on windows of size $2k + 1$. 

\[\]
Example 3.15 (linear scaling transformation). The linear scaling normalization \cite{Chapter 9} performs a linear transformation into the range \([0, 1]\) over a numeric attribute by computing \(v' = \frac{v - \min_A}{\max_A - \min_A}\) where \(\min_A\) and \(\max_A\) are the minimum and maximum values for the attributes \(A\), \(v\) is the instance value and \(v'\) is the normalized value.

\[
\text{wfor } i \text{ in } \$\text{seq} \\
\text{declare variable } \$\text{min-max} := (i, i) \\
\text{iterate let } \$\text{new-min} := \text{if } (\$\text{min-max}[1] < i) \text{ then } \$\text{min-max}[1] \text{ else } i \\
\text{let } \$\text{new-max} := \text{if } (i > \$\text{min-max}[2]) \text{ then } i \text{ else } \$\text{min-max}[2] \\
\text{return } (\$\text{new-min}, \$\text{new-max}) \\
\text{return for } j \text{ in } \$\text{seq} \\
\text{return } (j - \$\text{min-max}[1]) \div (\$\text{min-max}[2] - \$\text{min-max}[1]).
\]

Example 3.16 (sorted-neighborhood method). The query of Example 3.14 can also be adapted to the problem of merging multiple datasets, frequently encountered in data mining as a crucial first step of the KDD process. For example, the basic sorted-neighborhood method \cite{51} first sorts the records in the data list by using a key, and then moves a fixed size window through the sequential list of records limiting the comparisons for matching records to records in the window. The return statement can be easily rewritten to ensure this task.

3.3.4 External data structures

For efficiency purposes, we also design an XQuery implementation embedded in a Java environment. The environment can provide external variables and functions to XQuery. From the data mining point of view, this feature provides good opportunities for implementing and using complex data structures in an efficient way. The only extensions needed are the enrichment of the system type of XQuery with a new type, representing the reference to a Java object, and to provide a mapping among the primitive types of XQuery and Java. For example, the following code fragment defines a local variable that contains a reference to a Java Vector:

\[
\text{let } \$\text{my-vector as java := java.util.Vector:new(10).}
\]

We can also use instance methods in an intuitive way:

\[
\text{let } \$\text{value := java.util.Vector:get(\$\text{my-vector}, 5).}
\]

Example 3.17. To implement the problem of Example 3.13, we use an external Java-encoded Hashtable.

\[\text{wfor } c \text{ in doc("mondial")/country} \\
\text{declare variable } \$\text{ds as java := myPackage.MyHashTable:new()}
\]
iterate let $name = $c/religions[1]/text()
    let $v := $c/religions[1]/@percent * xs:decimal($c/population)
return myPackage.MyHashTable:put($ds, $name, $v)
return let $items as xs:string* := myPackage.MyHashTable:keys($ds)
let $avgs as xs:double* := myPackage.MyHashTable:values($ds)
for $p in (1 to count($items))
return if $avgs[$p] > 100000 then ($items[$p], $avgs[$p]) else ()

Here, put, keys and values are a re-implementation of the homonym Java methods over the myPackage.MyHashTable class, that extends the java.util.Hashtable class.

3.3.5 Core semantics of wfor

Giving the formal semantics of a query language is an important means in support of verifying the correctness of query transformations. In the following, we provide the formal semantics of wfor with the intent of using, in the near future, such definitions in the design and verification of query optimizations.

The definition of the core semantics of wfor is given by extending the semantics provided in [38], in which, the authors present a different semantics for XQuery with side effects, which is better suited for database compilation. As stated in [38], the dynamic semantics of the core is defined by the following judgment, where $\Sigma$ is the dynamic environment, $\sigma$ is the store, $e$ is a core expression and $v$ is a value:

$$\Sigma; \sigma \vdash e \Rightarrow v; \sigma'$$

Let $e_{win} \equiv \text{declare var } s := e_{decl} \text{ init } e_{iter} \text{ while } e_{w} \text{ return } e_{ret}$, the rules for providing the judgment of the wfor are:

$$\Sigma; \sigma \vdash e_{in} \Rightarrow (v_1, \ldots, v_n); \sigma_1 \quad n > 0 \quad \Sigma; \sigma_1 \vdash e_{decl} \Rightarrow v_{init}; \sigma_1^{init}
\begin{align*}
\Sigma; \sigma_1 & \vdash e_{init} \Rightarrow v_{open}^{init}; \sigma_1^{open} \\
\forall i \in 1, \ldots, n : (\Sigma, s \mapsto v_{init}); & \sigma_1^{open} \vdash e_{iter} \Rightarrow v_{next}^{open}; \sigma_i^{next}
\forall i \in 1, \ldots, n - 1 : (\Sigma, s \mapsto v_{next}^{open}); & \sigma_i^{next} \vdash e_{win} \Rightarrow (v_{next}^{open}, \sigma_i^{next})
\end{align*}
\Sigma; \sigma \vdash wfor \ s_x \text{ in } e_{in} \text{ e}_{win} \Rightarrow (v_{out}^{1}, \ldots, v_{out}^{n}); \sigma_{n+1}
\Rightarrow \text{win-true}
\Sigma; \sigma \vdash e_{w} \Rightarrow v; \sigma' \quad v \neq ()
\Sigma; \sigma \vdash e_{win} \Rightarrow (\Sigma(s),()); \sigma''
\Sigma; \sigma'' \vdash e_{init} \Rightarrow v'; \sigma''
\Sigma; \sigma \vdash wfor \ s_x \text{ in } e_{in} \text{ e}_{win} \Rightarrow ()
\Rightarrow \text{win-false}
\Sigma; \sigma \vdash e_{ret} \Rightarrow v; \sigma''
Chapter 4

XML preprocessing and classification

Abstract

The aim of a classifier is to create a model capable of assigning a class to instances according to the values of their attributes. Many methods have been proposed in the literature for classification, such as Bayesian classifiers, neural networks, decision trees and classification rules. In this chapter we deal with the decision trees, considering the problem of defining a new XQuake operator in order to extract a classifier.

Also, we are interested to preprocessing tasks that well fit with decision trees and make data more suitable for classification. In particular, the combined use of discretization and sampling techniques with classification is one of the most studied techniques in the research community. Hence, the definition of ad-hoc discretization and sampling operators out of XML data is a further contribution of this chapter. We will discuss both the semantics and the implementation aspects of such operators.

4.1 Data discretization

Several data mining algorithms, especially certain classification algorithms, require that the data be in the form of categorical attributes. Thus, it is often necessary to transform one or more continuous attributes into discrete ones. This process is called discretization.

4.1.1 Background

Discretization of continuous attributes has been extensively studied in literature. There is a wide variety of discretization methods starting with naive methods (i.e.
unsupervised methods), to more sophisticated methods, referred to as supervised methods, aiming at preserving the label homogeneity of a given attribute. For this reason, the latter are more suited for classification tasks.

Both supervised and unsupervised methods can be further distinguished into top-down and bottom-up methods, depending on whether intervals are formed by progressive cutting or merging. More specifically, a bottom-up method initially considers each data point as a separate interval and then it selects one or more adjacent data points merging them into a new interval. In contrast, top-down methods start with a single interval that includes all data attribute values and then generate a set of intervals by splitting the initial interval into two or more intervals. The Pearson $\chi^2$ algorithm [21] is an example of bottom up supervised method. At the opposite, the naive methods of equal-width and equal-frequency are examples of top-down unsupervised algorithms [68].

Another possible characterization depends on whether discretization is applied either independently (i.e. static discretization) or embedded in a data mining task, for example combined with classification algorithms (i.e. dynamic discretization).

We focus in the following on two classical approaches: the equal-width discretization and the equal-frequency discretization. We refer to the works proposed by Elomaa et. al. [3, 62] as a recent state-of-the-art in the area of discretization of numerical attributes.

**Equal-width discretization**

The equal-width (or natural binning) [68] is the simplest unsupervised method to discretize data into $k$ homogeneous intervals. Homogeneity here refers to computing the extreme values of a continuous attribute $A$ and to dividing the range of observed values into $k$ (approximately) equally width bins.

More specifically, suppose that there are $n$ training instances for which the values of $A$ are known, and suppose that the minimum and maximum value are $v_{A}^{\text{min}}$ and $v_{A}^{\text{max}}$ respectively. The algorithm divides the values between $v_{A}^{\text{min}}$ and $v_{A}^{\text{max}}$ into $k$ intervals of width

$$w = \frac{v_{A}^{\text{max}} - v_{A}^{\text{min}}}{k}.$$  \hspace{1cm} (4.1)

As a result, the cut points are at $v_{A}^{\text{min}} + w, v_{A}^{\text{min}} + 2w, \ldots, v_{A}^{\text{min}} + (k - 1)w$. The method is a special case of the top-down approach in which we start with a unique interval $[v_{\text{min}}, v_{\text{max}}]$ and we split it into equally sized bins.

**Example 4.1.** In the **mondial** dataset, the inflation and the population of the capital cities have continuous values. For the purpose of classification, we need a preprocessing phase in which, for each country, those values are discretized. For simplicity, we restrict our analysis to European countries having a Gross Domestic Product (GDP) greater than 150.000. The countries involved in the discretization
are listed in Table 4.1. Considering the attribute \textit{capital population}, an equal-width discretization into three intervals can be obtained by setting the interval width $w = 2.860.869$. In this way, we obtain the three intervals \textit{low} = [134.393, 2.995.262], \textit{med} = [2.995.262, 5.856.131] and \textit{high} = [5.856.131, 8.717.000]. The resulting values are transformed as shown in the column \textit{capital discr}. Similarly, the attribute \textit{inflation discr} shows the discretized values of the \textit{inflation} into 10 distinct bins, in which $w = 2.32$.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>France</td>
<td>Paris</td>
<td>1,7</td>
<td>2.152.423</td>
<td>I$_1$</td>
<td>low</td>
</tr>
<tr>
<td>Spain</td>
<td>Madrid</td>
<td>4,3</td>
<td>3.041.101</td>
<td>I$_1$</td>
<td>med</td>
</tr>
<tr>
<td>Austria</td>
<td>Vienna</td>
<td>2,3</td>
<td>1.583.000</td>
<td>I$_1$</td>
<td>low</td>
</tr>
<tr>
<td>Germany</td>
<td>Berlin</td>
<td>?</td>
<td>3.472.009</td>
<td>?</td>
<td>med</td>
</tr>
<tr>
<td>Italy</td>
<td>Rome</td>
<td>5,4</td>
<td>2.791.354</td>
<td>I$_2$</td>
<td>low</td>
</tr>
<tr>
<td>Switzerland</td>
<td>Bern</td>
<td>1,8</td>
<td>134.393</td>
<td>I$_1$</td>
<td>low</td>
</tr>
<tr>
<td>Poland</td>
<td>Warsaw</td>
<td>21,6</td>
<td>1.655.000</td>
<td>I$_{10}$</td>
<td>low</td>
</tr>
<tr>
<td>Ukraine</td>
<td>Kiev</td>
<td>9</td>
<td>2.616.000</td>
<td>I$_4$</td>
<td>low</td>
</tr>
<tr>
<td>Russia</td>
<td>Moscow</td>
<td>7</td>
<td>8.717.000</td>
<td>I$_3$</td>
<td>high</td>
</tr>
<tr>
<td>Belgium</td>
<td>Brussels</td>
<td>1,6</td>
<td>951.580</td>
<td>I$_1$</td>
<td>low</td>
</tr>
<tr>
<td>Netherlands</td>
<td>Amsterdam</td>
<td>2,25</td>
<td>1.101.407</td>
<td>I$_1$</td>
<td>low</td>
</tr>
<tr>
<td>Turkey</td>
<td>Ankara</td>
<td>9,4</td>
<td>2.782.200</td>
<td>I$_4$</td>
<td>low</td>
</tr>
<tr>
<td>Sweden</td>
<td>Stockholm</td>
<td>2,6</td>
<td>711.119</td>
<td>I$_1$</td>
<td>low</td>
</tr>
<tr>
<td>Un. Kingdom</td>
<td>London</td>
<td>3,1</td>
<td>6.967.500</td>
<td>I$_2$</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 4.1: Training set of the \textit{mondial} discretization problem. The last two attributes are obtained by means of a natural binning method.

\textbf{Equal-frequency discretization}

In Example 4.1, the distribution of the bins for the \textit{inflation} and \textit{population} attributes is highly unbalanced. As an instance, the intervals "\textit{med}" and "\textit{high}" of the population value contain only two elements, while the interval "\textit{low}" has 10 values.

A way to obtain more balanced intervals is the equal-frequency discretization that divides the range of a numeric attribute $A$ into $k$ intervals having the same frequency \cite{68}. It works as follows. Suppose there are $n$ training instances for which the values of $A$ are known. The algorithm sorts the observed values and then divides the sorted values into $k$ intervals so that each interval contains (approximately$^1$) the same number of training elements. In this way, each interval contains $w = \lceil n/k \rceil$

\textsuperscript{1}The number of instances can vary only for the last computed interval.
(possibly duplicated) adjacent values. The $j$-th elements $x_j$ belongs to the interval $i$ if $iw \leq j < (i + 1)w$.

**Example 4.2.** Let us consider again the Example 4.1. A balanced discretization of the attribute *population* may be obtained by means of the equal-frequency binning setting the interval frequency to 5. The computed bins now are low = [134.393, 1.619.000), med = [1.619.000, 2.916.227), high = [2.916.227, 8.717.000].

### 4.1.2 Task formulation

In the following we investigate on how an unsupervised discretization task can be formalized in the XML-based framework presented in Chapter 3. As a first step, we define the conditions in which an inductive database theory represents an unsupervised discretization task over the X(DM)$^2$ data model. The definition below accomplishes this task.

**Definition 4.1 (XML unsupervised discretization).** Suppose that an XML dataset $R = (r_1, \ldots, r_n)$ is given. For a certain XQuery expression $F$, let $A_{R,F} = (a_1, \ldots, a_n)$ be an active XML attribute over $R$ whose type is numeric. Also, let $D$ be a non-empty set of string values representing the required labels. An inductive database theory $Th(L, <R, A_{R,F}, D>, q)$ represents an XML unsupervised discretization task if:

- $L = \{ (G(r_1, a_1, d_1), \ldots, G(r_n, a_n, d_n)) \mid d_i \in \text{dom}(D) \text{ for } i \in [1, n] \}$, where $G$ is a generic XQuery expression;
- $q(<R, A_{R,F}, D>, (G(r_1, a_1, d_1), \ldots, G(r_n, a_n, d_n))) = \text{true}$ if and only if
  1. $\forall i \in [1, n], d_i$ is an empty sequence $\Leftrightarrow a_i$ is an empty sequence;
  2. there is an unsupervised discretization of the values of the attribute $A_{R,F}$ into $|D|$ intervals and $d_i \in \text{dom}(D)$ is the discretization value of $a_i$, for each $i \in [1, n]$.

The main idea is to bind the deductive part $<R, A_{R,F}, D>$ with the inductive virtual part $(G(r_1, a_1, d_1), \ldots, G(r_n, a_n, d_n))$. The deductive part consists of a triplet in which we specify the XML training dataset, $R$ having $n$ nodes, the discretization XML attribute over such a dataset, $A_{R,F}$, as well as the required labels, $D$. The XML attribute is required to be numeric.

The inductive part consists of a sequence of $n$ XML nodes. Each node is obtained on the basis of an XQuery expression $G$ computed over each node of the dataset, the corresponding numeric value of $A_{R,F}$ and the discretized value obtained by means of a discretization process. A valid discretization of the nodes of
4.1. DATA DISCRETIZATION

R, formalized as a theory $Th(\mathcal{L}, <R, A_{R,F}, D>, q)$, is obtained by suitably defining $q(<R, A_{R,F}, D>, G(r_1, a_1, d_1), \ldots, G(r_n, a_n, d_n))$ in a way that each $a_i$ is mapped to the interval $d_i$, according to some discretization objective. Missing values for $A_{R,F}$ are directly treated as unknown values by the XML discretization task. The above definition can be easily generalized to the case of a generic number $k > 1$ of discretization attributes $A_{R,F_1}, \ldots, A_{R,F_k}$ whose values must be discretized according to a common strategy.

We can provide a concrete model of the above inductive problem in terms of an XQuake operator. The syntax is defined as follows.

$$\text{DiscrOp ::= "prepare" "discretization" "doc" "{" QName "}" UsingClause}$$
$$\text{ForDataClause HavingUnsupDiscrClause+ ReturnClause}$$
$$\text{HavingUnsupDiscrClause ::= "having" BinExpr ("," BinExpr)*}$$
$$\text{BinExpr ::= "bin" "$" VarName "as" ExprSingle}$$

As one could notice, the XML dataset and the XML discretization attributes are expressed via the ForDataClause, as defined in the Figure 3.5.

The user preferences, i.e. the interval labels for the discretization, are specified in the WithDiscrClause through a nonempty set of BinExpr statements that encapsulate XQuery expressions in the body part. As stated in the Definition 4.1 their expected type is a non-empty sequence of string values, thus the operator checks for the type correctness before doing the mining. Also, it guarantees that the number of active discretization attributes declared in the ForDataClause clause is equal to the numbers of BinExpr expressions. The result of the unsupervised discretization is bound to user-defined variable(s) declared in the BinExpr statement(s). As required by the Definition 4.1, the returning type of such variables is string?, and missing input field values for a certain XML attribute will be labelled with missing discretized values.

Finally, the returnClause accomplishes the output construction task. The XQuery expression in the body of the statement is evaluated for each input node, to produce a collection of XML nodes. The variables defined in the clauses above are in the scope of the XQuery expression.

As a first example of application of the operator, we consider the problem of unsupervised discretization in the mondial dataset.

**Example 4.3.** The discretization problem stated in the Example 4.1 can be solved in XQuake by means of the following expression.

```xquery
prepare discretization doc("mondial-discr") using alg:equal-width()
for data $country in doc("mondial")/mondial/country
where some $j in $country/encompassed Continent satisfies $j eq 'europe'
and $country/gdp total > 150000
let active field $inf := xs:double($country/inflation)
let active field $pop := xs:double($country//@is-capital="yes")[pop]
having bin $inf-d as ('low', 'med', 'high'),
```


Specifically, we aim at computing three distinct bins for the inflation attribute and 10 distinct bins for the population attribute. Only in the first case nominal labels are explicit. In the latter case, labels for the intervals are automatically provided by the built-in XQuery function `format-bins` of the `discr` library, that takes as input the number of required intervals, \( k > 0 \), and returns a sequence of string items of length \( k \). The result of the discretization is bound to the variables `inf-d` and `pop-d` for the inflation and capital population values, respectively. Notice that the scope of such variables is the \texttt{return} clause below. An example of a fragment of the output produced by the above operator is the following.

\[
\text{\textless discr-country>}
\begin{align*}
\text{name} & \text{France}\text{/name>}
\text{population} & \text{10170241}</population>
\text{inflation @discr-as="low">1.6</inflation>
\ldots
\text{cap-pop @discr-as="[900000, 950000)">951580</cap-pop>
\text{\text/discr-country>
\end{align*}
\text{\texttt{\textbackslash{\text{\textbackslash{\textend{align*}}}}}}
\]

The next example illustrates the usage of the \texttt{GROUP} facility useful to avoid the specification of each single discretization attribute.

\textbf{Example 4.4.} We aim at discretizing, into two distinct bins, the amount of population per unit area of the top ten provinces in a country, ordered according to the number of cities.

\[
\text{\texttt{let $ord := for$i in$country/province order by count($i/city) descending return$i/population div$i/area
for$j in (1 to 10) return$ord[$j]}};
\]

\[
\text{\texttt{prepare discretization doc("human-density") using alg:equal-frequency()
for data$country in doc("mondial")//country[count(./province) >= 10])
let$group<10> field$r := top-10-density($country)
having bin$r-d in ("low-density", "high-density")
return$r @n="{$country/name}"} for$i in$r-d return$d}{$i}</d></r>\]
The method used here is the equal-frequency. In contrast to Example 4.3, the usage of the \texttt{GROUP} keyword in the \texttt{LET} clause avoids the specification of each single active field, since it groups into a numeric sequence, $\$r$, every value to be discretized. The XQuake system constrains the sequence to have the same length at each iteration. In this example, the xpath expression placed in the \texttt{FOR DATA} clause ensures that the countries having less than ten regions are eliminated from the discretization process.

We do not deal in this thesis with the problem of supervised discretization, rather we concentrate on providing an implementation of the above operator in the following.

### 4.1.3 Task implementation

#### Equal-width implementation

The implementation of the natural binning method is quite straightforward. For an XML discretization attribute, $A$, we compute the discretization interval width $w$ according to the Equation 4.1 and then we assign each value $r$ to the interval $j$ such that $r \in [v_{A}^{\text{min}} + jw, v_{A}^{\text{min}} + (j + 1)w]$.

The definition of two different iterations over the data accomplishes the implementation of the task. This is shown in the XQuery function of Figure 4.1, in which, for simplicity, only the implementation over a single discretization attribute is provided. The attribute index is given as a parameter. First, during the first data iteration (lines 2-6) we collect the minimum and maximum values of $A$ in a local variable, $\$\text{min\_max}$. Next, in the second iteration (lines 6-12) we compute the width of the interval and we assign each element to the corresponding bin (function \texttt{get-bin}). Notice that at line 9, the method checks whether the input attribute has a missing value. In this case, an empty sequence is returned as bin, according to the Definition 4.1.

#### Equal-frequency implementation

A solution to the equal frequency binning requires first to sort the input instances, and then to produce balanced intervals of size $w = \lceil n/k \rceil$ where $n$ is the number of cases and $k$ is the required number of bins. With ordered data, the $j$-th element belongs to the interval $i$ if $iw \leq j < (i + 1)w$.

A naive implementation of such a strategy can be easily provided by means of a \texttt{wfor} iterator. In a way similar to Example 3.12, we first order the data according to the discretization XML attribute. Then, the \texttt{wfor} command creates a data window of length $\lceil n/k \rceil$ whose elements share the same bin. For each element in the window, we return that interval. However, the presented solution has two main disadvantages. First of all, it cannot be applied to the case of $m > 1$ attributes. Second, it does
1. declare function local:natural-binning($a-index) as node* {

2. wfor $i in core:for-data-where()
3. declare variable $min-max as xs:int* := ()
4. iterate let $v := core:let-active($i, $a-index)
5. return (fn:min(($min-max[0], $v)), fn:max(($min-max[1], $v)))
6. return let $delta := ($s[2] - $s[1]) div count(discr:bins($a-index))
7. for $i2 in core:for-data-where()
8. let $v := core:let-active($i2, $a-index)
9. return if (empty($v)) then ()
10. else let $bins := discretization($a-index)
11. let $bin := local:get-bin($v2, $delta, $bins)
12. return discretization($i2, $v, $bin)

Figure 4.1: Equal-width implementation.

1. declare function local:equal-frequency($a-index) as node* {

2. wfor $i at $pos in core:for-data-where()
3. declare variable $tree-set := TreeSet:new($discr:bins($a-index))
4. iterate let $v := core:let-active($i, $a-index)
5. return TreeSet:insert($tree-set, $v, $pos)
6. return let $discr := TreeSet:discretize($tree-set)
7. for $i2 at $pos2 in core:for-data-where()
8. let $v2 := core:let-active($i2, $a-index)
9. let $bin := TreeSet:get-bin-at($tree-set, $pos2)
10. return discretization($i2, $v2, $bin)

Figure 4.2: Equal-frequency implementation.

not maintain the original ordering of input instances, and the output results in a “shuffled” dataset.

In order to improve the effectiveness of the implementation, we can make use of more complex structures, to be managed via external methods. The specification reported in Figure 4.2, for example, makes use of a tree set structure, tree-set, to sort the elements and compute the intervals. The structure stores the pairs \((v, p)\) where \(v\) is the numeric value to be discretized, and \(p\) represents its position in the dataset. The order is maintained on \(v\). The declare statement at line 3 initializes
The `insert` external function inserts a new pair $(v', p')$ into the tree set structure, maintaining the order of the elements. The `discretize` external method at line 6 acts as follows. It scans the `tree-set` in an ascending ordering and computes, for each element $v$, its discretized value. This is stored in an array, $A$, in the position indicated by $p$. Missing values of $v$ will be mapped into null values in $A$. Finally, we iterate a second time over the data (lines 7-10) in order to collect the parameters of the `return` function (line 10). In particular, at line 9, the `get-bin-at` Java method extracts the element at position $\text{pos2}$ from $A$.

As a result, the process requires 2 iterations over the data and $\log(n)$ comparisons to insert an element into the tree set structure. The overall complexity requires $O(n \log n)$ comparisons.

4.2 Data sampling

4.2.1 Background

The aim of data sampling in data mining is to draw a sample from the database that allows us to construct a model that reflects the structure of the data [48] (Chapter 4). Trivially, one reason for using just a sample rather than the entire dataset is efficiency. In fact, it may be infeasible, in terms of time requirements, to use the entire database, especially for mining algorithms that require several iterations over the data. By basing our computation solely on a sample, we make it quicker and easier. A fundamental requirement, however, is that such a sample must be drawn in a way that it reflects the structure of the complete dataset, i.e. that it is representative of the entire database. Two techniques for ensuring this property are the random sampling and the stratified sampling.

Simple random sampling

The simplest sampling method that avoids regularities over the data is the random sampling. It is based on the following assumption:

"each record in the database has an equal chance of being chosen."

As a consequence, in a simple random sampling, the $n$ records are selected from the $N$ records in the database in such a way that each set of $n$ records has an equal probability of being chosen.

A random sampling can be distinguished between sampling with replacement and sampling without replacement. In the former, a record selected for inclusion in the sample has a chance of being drawn again, but in the latter, once a record is drawn it cannot be drawn a second time. The differences between the two strategies increases for sample size near to the entire population size, but it is usually negligible for small data samples.
Stratified sampling

In *stratified sampling*, the entire population is split into non-overlapping sub-populations or *strata* and a sample is drawn separately from within each stratum. A stratum is a subset of the population that shares at least one common characteristic, for example males and females or full-time, part-time and unemployed. To select the subjects from each stratum, random sampling is often used. In this case, we deal with *stratified random sampling*. An important advantage of such a strategy is that if the strata are relatively homogeneous in terms of the variable of interest, the variance of the overall estimate may be smaller than that arising from a simple random sampling [48] (Chapter 4).

### 4.2.2 Task formulation

The specification of sampling random techniques in X(DM)$^2$ is quite straightforward. From a conceptual point of view, the theory $Th$ is represented by the set of XML nodes of an XML dataset that have been selected by the sampling strategy. The deductive part $L$ of the theory consists of a pair in which the XML training dataset and the required percentages are specified. According to the XQuake philosophy, the operator that accomplishes this task is also defined to permit a customized output.

**Example 4.5.** We aim at selecting two different samples, training and test set, from all the papers of the dblp database. The XML dataset includes both proceedings and journal papers. The training set (resp. test set) has 50% (resp. 25%) cases over the total number of papers and the strategy used is the simple random sampling with replacement. An example of the XML fragment to generate is the following:

```xml
<train @cases="2">
  <paper @key="SRC1997-018"/>
  ...
</paper>
</train>
<test @cases="1">
  <paper @key="Nayak02"/>
  ...
</paper>
</test>...
```

where the attribute `@cases` is the number of times that the paper has been included into the training or test set (with replacement strategy).

**Definition 4.2** (XML random sampling). Let $R = (r_1, \ldots, r_n)$ be an XML dataset. Let $P = (p_1, \ldots, p_m)$ be a sequence of numeric values with $m \geq 1$ and $0 \leq \sum_{i=1}^{m} p_i \leq 1$. An inductive database theory $Th(L,<R, P>, q)$ represents an XML random sampling task if:

- $L = \{(F(r_1, s_{i1}^1, \ldots, s_{jm}^1), \ldots, F(r_n, s_{i1}^n, \ldots, s_{jm}^n)) \mid s_{ij}^i \in \mathbb{N} \forall i \in [1, n], j \in [1, m]\}$

  where $F$ is a generic XQuery expression;

\end{definition}
4.2. DATA SAMPLING

- \( q(<R, P>, (F(r_1, s^1_1, \ldots, s^m_1), \ldots, F(r_n, s^1_n, \ldots, s^m_n)) = \text{true if and only if} \)
  
  - \( s^j_i \) is the number of times that the node \( r_i \) has been included into the sample \( j \) according to a simple random strategy;
  - \( \forall j \in [1..m], \sum_{i=1}^n s^j_i \simeq n \cdot p_j; \)
  - only in a “without replacement” strategy the task also ensures that: \( \forall i \in [1..n], \sum_{j=1}^m s^i_j \leq 1. \)

Intuitively, a valid simple random task out of XML data is formalized as an inductive database theory in which both the deductive and the inductive part are defined. The deductive part is a sequence \((r_1, \ldots, r_n)\) of length \( n \), where \( n \) is the number of XML nodes for which a sampling is required. The inductive part is a sequence of length \( n \) of XML nodes, \((s'_1, \ldots, s'_n)\). Each \( s'_i \) is computed by means of an XQuery expression, \( F \), defined over the node \( r_i \) and over a sequence of length \( m \) of natural numbers that encode the result of the sampling. In this case, \( m \) is the number of samples defined by the user. Notice that, as usual, the properties of the sampling are ensured by suitably defining the predicate \( q \).

**Example 4.6.** Below an XQuake query that resolves the sampling problem of Example 4.5 is presented.

```xquery
prepare sampling doc("s-proceedings") using alg:random-sampling(true())
for data $p$ in doc("dblp")//[Inproceedings|article]
  having samples $s$ in (0.50, 0.25)
return let $t1 := if ($s[1]>0) then <train @cases="$s[1]"> {$p} </train> else ()
                 let $t2 := if ($s[2]>0) then <test @cases="$s[2]"> {$p} </test> else ()
return ($t1, $t2).
```

The binary parameter of the algorithm indicates a “without replacement” policy, in which papers are equally likely to be extracted. Inside the HAVING expression, the SAMPLES clause binds to the variable $s$ a pair in which the first (resp. second) value is the number of times that the current paper has been included into the first (resp. second) sample, according to the sampling strategy.

**Example 4.7.** With reference to the \(<\text{country}>\) elements of the \texttt{mondial} dataset, the sampling below works with XML nodes splitted into two different categories: the countries that are members of the UNESCO organization and those that are not.
We aim at applying a random sampling at each stratum selecting ten equal-size samples. The problem can be solved via a stratified sampling, in which the selection attribute divides the countries according to the required property.

\[
\text{prepare sampling doc("s-mondial") using alg:stratified-sampling(false())}
\]

\[
\text{for data $c$ in doc("mondial"):country}
\]

\[
\text{let active field $is-unesco := string(fn:contains($c/memberships, "UNESCO"))}
\]

\[
\text{having samples $s$ in for $i$ in (1 to 10) return 0.1}
\]

\[
\text{return for $i$ at $pos$ in $s$}
\]

\[
\text{return if ($i = 1$) then <sample @index="$pos"> $c$ </sample> else ()}
\]

The syntax of the PREPARE SAMPLING operator is shown below.

\[
\text{SamplingOp ::= "prepare" "sampling" "doc" "{" QName "}" UsingClause}
\]

\[
\text{ForDataClause HavingSamplingClause ReturnClause}
\]

\[
\text{HavingSamplingClause ::= "having" SamplesExpr}
\]

\[
\text{SamplesExpr ::= "samples" "$" VarName "in" ExprSingle.}
\]

For the simple sampling, the ForDataClause statement only includes the specification of the input XML dataset and of the optional filtering condition. For the stratified sampling, the operator also requires the definition of an active XML attribute.

### 4.2.3 Task implementation

#### Simple random sampling

The problem can be easily formulated in the following way. Suppose we want to extract \( n \) samples \( S_1, \ldots, S_n \) of fixed cardinality with \(|S_1| + \ldots + |S_n| \leq m\) where \( m \) is the total number of XML items. The random sampling without replacement policy (in short RSWOR) selects \( n \) items from the input XML dataset where the probability of drawing any item is \( 1/m \). The random sampling with replacement method (RSWR) is similar to the previous one, except that each time an item is drawn from the dataset, it is recorded and then replaced. That is, after an XML node is drawn, it is placed back in the dataset, so that it may be drawn again.

Figure 4.3 depicts the main procedure for the random sampling in which the parameter indicates the kind of sampling. Essentially, the procedure demands the data iteration and the output construction to XQuery, whereas it uses external Java functions for the randomization process.

The body of the procedure is implemented as a FLWOR expression. At lines 2 and 3 respectively, the overall number of XML nodes, \( n \), and the required samples percentages are computed. At line 4, the procedure declares an external Java object, \$rand\, as an instance of the randomizer class. The object constructor operates as follows. It initializes an array of length \( n \) of random values according to the kind
4.2. DATA SAMPLING

1. declare function local:random-sampling($strategy as xs:boolean) {
2.   let $num-tuples as xs:int := core:num-tuples()
3.   let $ranges as xs:double+ := sampling:with-samples()
4.   let $rand as java := randomizer:new($num-tuple, $ranges, $strategy)
5.   for $i at $pos in core:for-data-where()
6.     let $evaluation as xs:int+ := randomizer:randomIndexes($rand, $pos)
7.     return sampling:eval-return($i, $evaluation)
8. }

Figure 4.3: Simple random sampling implementation.

of policy used. More specifically, for a RSWOR strategy, the array is initialized in such a way that each index $i$ of the XML tuple is in the position $i$, for $i \in [1, n]$. Then, the constructor shuffles the array by randomly exchanging its elements. This strategy allows that a particular index is not generated repeatedly. Conversely, in a RSWR technique, the constructor fills each cell of the array with a pseudorandom number, uniformly distributed into values between 1 and $n$, drawn from a random number generator’s sequence. At each extraction, the indexes of the XML tuples, from 1 to $m$ have the same probability to be placed in the array.

The iteration over the XML nodes starts at line 5. For each node that survives the where, we demand to the external function randomIndexes the generation of the values $s^1_i, \ldots, s^m_i$, where each $s^j_i$ is the number of times that the node instance $i$ has been included into the sample of index $j$. This is achieved by scanning the array of values generated in the constructor to find the position(s) of the input index $i$. Such positions indicate in which sample(s) the XML nodes fall into. For example, with reference to Example 4.7, suppose that the required samples are $S_1$ and $S_2$ having respectively 50% and 25% cases over the total number of cases $n = 100$. The array length is 100. The first 50 positions are reserved to $S_1$, the next 25 positions to $S_2$ and, finally, the last 25 positions stand for the cases having no sample. Suppose that by using a RSWR strategy the input XML node of index $i$ occur in the positions 12, 45 and 70 of the array. This means that the node $i$ has been selected one time to belong to the sample of index 1 and twice to the sample of index 2. The return value is the sequence (1,2).

Stratified random sampling

Given an XML dataset with $n$ nodes and a nominal XML attribute $A$, containing $m$ distinct values, that divides the dataset into $m$ mutually disjoint strata, the stratified sampling applies a simple random sampling at each stratum using a with
replacement or a without replacement policy.

The implementation of the iterator for the stratified sampling is similar to the previous one, but acts by means of two iterations over the data.

1. The first one in order to collect statistics - i.e. the frequencies of the outcomes - over the XML attribute that determines the strata, say \( A \). This computation can be easily implemented by using a Java Hashtable that stores the outcome name of \( A \) as the key and the outcome frequency as the value.

2. The second iteration uses the procedure of Figure 4.3 to randomly determine the cases for each stratum (i.e. outcome). The procedure can be rewritten to this purpose. Suppose \( m \) are the outcomes of \( A \), at line 4 we define \( m \) external objects \( \text{rand}_1, \ldots, \text{rand}_m \) as instances of the \texttt{randomizer} Java class. Now, the \texttt{num_tuples} parameter of \( \text{rand}_i \) refers to the frequency of the \( i \)-th outcome in \( A \). As we scan through the XML nodes (line 5), we invoke the \texttt{randomIndexes} external method over \( \text{rand}_i \), where \( i \) is the index of the outcome encountered for this XML node.

### 4.3 Decision tree classification

Classification is the task of assigning objects to one of several predefined categories according to the values of their attributes. The input data is a collection of records in which a special attribute is designed as the class attribute - or target attribute. The task is to build a model for predicting the value of the class attribute, from the values of the others. In tree classification, the model is usually called decision tree. This section, after introducing the basic concepts of classification, describes the issue of building a classification tree over XML data.

#### 4.3.1 Background

**The classification problem**

A classification model may serve as a descriptive model, i.e. as an explanatory tool for distinguishing between objects of different classes. However, its most frequent usage is as a predictive model, i.e. used to predict the class label of unknown records.

**Example 4.8** (play-tennis classification). Consider the sample dataset of table 3.2. The idea is to extract a classification model for answering to the question whether a day is suitable to play tennis or not according to the cases collected in the past. Training cases are the records of the table 3.2. The target attribute is the \texttt{Class} field, with two different values: \{ “yes”, “no”\}. The attribute set, i.e. the set of active fields as input of the analysis, is \texttt{Outlook}, \texttt{Temperature}, \texttt{Humidity} and \texttt{Wind}. The table contains some missing values (indicated by \texttt{null} values) for these attributes, that must be taken into consideration during the construction of the model.
Although the attributes presented in the above example are mostly discrete, the
attribute set can also contain continuous features. On the other hand, a fundamental
property of the classification task is that the class label must be a discrete attribute.
This characteristic distinguishes classification from regression, that is a predictive
mining task in which the target attribute assumes continuous values.

Among a plethora of techniques for the classification problem, such as Bayesian
classifiers [30, 122], neural networks [96], support vector machines [118] and rule-
based classifiers [63] (Chapter 5), we take decision trees into consideration [86, 76]
[63] (Chapter 4) for the following reasons:

- Decision trees do not require any prior assumption regarding the type of prob-
ability distributions among the class and the other attributes, unlike methods
such as Bayesian classification. This makes decision trees capable of dealing
with several different contexts.

- Decision trees are relatively easy to understand and interpret (at least for
smaller-sized trees), while maintaining an accuracy comparable with other
classification techniques.

- Decision trees are quite robust in the presence of noise or missing values, a
non-rare event in XML data.

**Decision trees**

A decision tree is a tree data structure that has three kinds of nodes [63] (Chapter
4):

- A *root node* has no incoming edges and zero or more outgoing edges. Each
tree has exactly one root node.

- An *internal node* has exactly one incoming edge and two or more outgoing
edges. Each internal node, together with the root node, contains an attribute
test condition, to separate records that have different characteristics.

- A *leaf node* or *terminal node* has exactly one incoming edge and no outgoing
edges. Each leaf node is labeled with one of the possible values of the target
attribute.

Unknown samples are classified by testing their attributes against the decision
nodes, and are assigned to the class corresponding to the reached leaf node. For
example, the root node shown in Figure 4.4 uses the attribute Outlook to separate
sunny and overcast days from rainy days. In a similar way, the internal node Wind is
used to separate strong winded days from no winded ones. Classifying a test record
is a process that starts from the root node applying the test condition to the record
and following the appropriate branch based on the outcome of the test. This will
generate a recursive process on the internal nodes, for which a new test condition is applied. The process terminates when a leaf node is reached, and the class label associated is assigned to the record. For example, with reference to the Figure 4.4, the sample \{Outlook=rainy, Wind=strong\} is classified as No. Starting from the root node, the first attribute selects the rightmost branch of the tree, and then, the second attribute selects the left branch of the Wind node.

**Building a decision tree**

Finding an optimal decision tree is a NP-complete problem [53]. Many algorithms employ a heuristic-based approach to guide their search in the hypothesis space. The general schema that we present in this section uses a greedy top-down recursive partitioning for growing a decision tree. In contrast, once a decision tree has been built, classifying a test record is fast, with a worst-case complexity of $O(n)$, where $n$ is the maximum depth of the tree.

A general schema for decision tree induction algorithm is given in Algorithm 1 [63] (Chapter 4). Basically, the algorithm works recursively from the root node, selecting the best attribute to split the data and expanding the leaf nodes until a stopping criterion is met. More specifically:

- The **find_best_split** function determines which attribute should be selected for splitting the training records. This depends on which measure is used.
- The **classify** function selects the class label to be assigned to a leaf node. Typically, the leaf node is assigned to the class that has the majority number of training records.
- The **stopping_cond** terminates the process of tree construction by testing whether all the records have either the same class label or the same attribute
label. Most of the times, the recursion terminates when the number of records have fallen below a minimum threshold.

Algorithm 1 treegrowth\((E,F)\)

<table>
<thead>
<tr>
<th>Input:</th>
<th>Training records (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Attribute set (F)</td>
</tr>
<tr>
<td>Output:</td>
<td>A decision tree</td>
</tr>
<tr>
<td>if</td>
<td>(\text{stopping_cond}(E,F)) then</td>
</tr>
<tr>
<td></td>
<td>leaf = \text{create_node}();</td>
</tr>
<tr>
<td></td>
<td>leaf.label = \text{classify}(E);</td>
</tr>
<tr>
<td></td>
<td>return leaf;</td>
</tr>
<tr>
<td>else</td>
<td>root = \text{create_node}();</td>
</tr>
<tr>
<td></td>
<td>root.test_cond = \text{find_best_split}(E,F);</td>
</tr>
<tr>
<td></td>
<td>let (V = {v \mid v\text{ is a possible outcome of root.test_cond}});</td>
</tr>
<tr>
<td></td>
<td>foreach (v \in V) do</td>
</tr>
<tr>
<td></td>
<td>(E_v = {e \mid \text{root.test_cond}(e)=v \text{ and } e \in E});</td>
</tr>
<tr>
<td></td>
<td>child = treegrowth((E_v,F));</td>
</tr>
<tr>
<td></td>
<td>add child to root with label the edge (root (\sim) child) as (v);</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>return root;</td>
</tr>
</tbody>
</table>

Example 4.9. The decision tree of the Example 4.8 is obtained by recursively dividing the training set of Table 3.2 until all the samples in each partition belong to the same class. At the root node, the attribute \textit{Outlook} is chosen as splitting attribute as stated by the method \text{find\_best\_split}. The attribute has three different values, so we have three branches from the node. Since all the samples in the \textit{overcast} partition belong to the same class, the branch corresponding to the value \textit{overcast} terminates in a leaf node. Differently, the partitions relative to the other two values do not have an uniform class value for all the samples. Thus, a new attribute is chosen for each partition and the sample is divided again. For example, the partition relative to the value \textit{sunny} is splitted according to the attribute \textit{Humidity}. ▷

After building the decision tree, a \textit{post-pruning} step may be achieved via a reduction of the size of the tree. In this phase, some branches are removed by replacing a split node with a leaf. This usually happens when the prediction error of the pruned branch is not worse than the error of the unpruned tree.

Many other algorithms prefer to operate by means of an \textit{early stopping} method in place of post-pruning methods. In this case, a tree is pruned by stopping its
construction by testing if the prediction attribute under consideration is below a user-defined level of confidence. If this is true, the current node becomes a leaf and the class label $l$ is associated to the node with a probability (i.e. a confidence), $c$, i.e. the instance has $c\%$ chances of being classified as $l$, w.r.t. the total number of cases of that branch.

Pruning is crucial for the quality of the final tree since decision trees that contain a large number of nodes are susceptible to the phenomenon known as overfitting.

**Selecting the best split: the ID3 algorithm**

As earlier mentioned, many decision tree learning algorithms differ in the definition of the splitting criterion. For instance, the CART [24] algorithm uses the Gini index, whereas the id3 [86] and C4.5 [87, 88] algorithms use the entropy. Generally speaking, a statistical criterion gives a measure of the impurity of the samples for each child node. The smaller the value of the impurity, the more skewed the class distribution. For example, a node with class distribution $(0,1)$ has zero impurity, whereas a node with uniform class distribution $(0.5,0.5)$ has the highest value of the impurity. As one can notice, the first class distribution is preferable.

The ID3 algorithm uses the information gain to quantify how well a test attribute divides the samples. More precisely, at each decision node, the algorithm chooses a possible attribute test with $n$ outcomes. The set $S$ of the training samples can be partitioned into subsets $S_1, \ldots, S_n$. If $S$ is a set of samples and $C_i$ is one of the possible classification values, we denote with $freq(C_i, S)$ the number of samples $S$ that belong to class $C_i$. The probability that a randomly picked sample in $S$ will belong to a class $C_i$ is given by:

$$\frac{freq(C_i, S)}{|S|}.$$  

Then, the information brought by this event is measured w.r.t. this probability:

$$- \log_2 \left( \frac{freq(C_i, S)}{|S|} \right).$$

To determine the expected information conveyed by a set of samples $S$, the classes information is summed in proportion to their probabilities:

$$\text{info}(S) = - \sum_{i=1}^{n} \frac{freq(C_i, S)}{|S|} \times \log_2 \left( \frac{freq(C_i, S)}{|S|} \right)$$

where $n$ is the number of the possible classifications. When the samples in $S$ are splitted according to a test attribute $T$ with $l$ outcomes, the expected information of the splitting is measured as the weighted sum of the information in each subset:

$$\text{info}_T(S) = \sum_{j=1}^{l} \frac{|S_i|}{|S|} \times \text{info}(S_i)$$
where $S_i$ is the $i$-th subset of the partition of $S$. Finally, the information gained by partitioning $S$ with the test attribute $T$ is given by

$$gain(T) = info(S) - info_T(S). \quad (4.2)$$

Intuitively, the information gain can be considered a measure of the confusion of the data. When the data is uniformly distributed among the classes then it will have an high value. Conversely, if the data has only one class the information gain will be zero. The ID3 method is based on this measure and it selects the attribute that maximize the quantity expressed by Equation 4.2 (procedure \texttt{find\_best\_split} of Algorithm 1). A limitation of the algorithm is that it cannot deal with training sets having numeric features. For this purpose, a discretization process must be adopted before applying the mining.

### 4.3.2 Task formulation

In the following, we are interested in specifying a new operator for the extraction of a decision tree from XML data. The starting point is the X(DM)$^2$ data model defined in Chapter 3. First, a user as to define the XML training dataset, i.e. the set of XML nodes used to build the model. Then, he/she has to specify the set of XML attributes that are active for the analysis. A special field is used as target attribute. For simplicity, we shall assume that all the attributes represent discrete features. This is not a major problem, since we can adopt discretization techniques (Section 4.1) over the training set before extracting the decision tree. The resulting output is a PMML decision tree.

**Example 4.10.** Our aim is to consider the mondial database as an XML table suitable as an input to a classification algorithm intended to build a model of the geographical, economic and political information of the countries. More specifically, the goal is to classify new countries in two different categories: the countries that are good candidates to become a new member of the UNESCO and those that are not. The classification scenario is as follows. Records of the training set are the countries located by means of the homonymous XML element. The set of the attributes includes country’s properties like:

1. the government type;
2. the discretized values of the capital population and the level of inflation (see example 4.3);
3. a binary attribute indicating whether the country is a member of the FAO;
4. a binary attribute indicating whether the capital of the country has a size greater than a fixed value.
At this point, we need to define when an inductive database theory $Th(\mathcal{L}, r, q)$ represents a valid XML decision tree extraction problem over X(DM), i.e. we need to specify the inductive part $\mathcal{L}$, the deductive part $r$ and when the predicate $q(r, p)$ is true.

**Definition 4.3** (discrete XML decision tree extraction). Let $S$ be a training set and let $A_{S,F_1}, \ldots, A_{S,F_k}$ be $k$ active string XML attributes over $S$ w.r.t $F_1, \ldots, F_k$ XQuery expressions. Also assume, over $S$, a discrete XML predicted attribute, $C_{S,G}$, where $G$ is an XQuery expression. We define:

- $\mathcal{L} = \{ p \mid p$ is a PMML tree document $\}$
- $q(<S, A_{S,F_1}, \ldots, A_{S,F_k}, C_{S,G}>, p) = \text{true}$ if and only if:
  - each internal node in $p$ is labeled with the name of an XML attribute $A_{S,F_j}$, with $j \in [1, k]$.
  - each branch of $p$ is labeled with a simple set predicate\footnote{A simple set predicate checks whether a field value is element of a set.} defined over a value $v \in \text{dom}(A_{S,F_j})$.
  - each leaf is labeled by a class $c \in \text{dom}(C_{S,G})$.

The resulting theory $Th(\mathcal{L}, <S, A_{S,F_1}, \ldots, A_{S,F_k}, C_{S,G}>, q)$ formalizes a discrete XML decision extraction task.

In XQuake, the issue of modelling the definition of an XML decision tree extraction task acts by means of the `mine tree` operator.

**Example 4.11.** Suppose we want to model the mining problem explained in the Example 4.10. We can specify it by means of the following XQuake expression.

```
mine tree doc("is-unesco-member") using alg:id3()
for data $c$ in doc("europe-discr")/country
let active field $government := string($c/government)
let active field $population := $c/population/@discr-as
let active field $inflation := $c/inflation/@discr-as
let active field $is-fao := string(fn:contains($c/memberships, "org-FAO"))
let active field $ext-cap := string($c//city[@is-capital = "yes"]/@gdp > 10000)
let predicted field $c$ as xs:discrete<"true","false"> :=
    string(fn:contains($c/memberships, "UNESCO"))
```

The statement can be read as follows. The `using` clause specifies the mining algorithm, ID3 in this case. The `for` clause identifies the XML nodes that denote the records of the training set. Each `let` element specifies the attribute in the source data set that is considered for mining, i.e. the mining schema. Since the ID3 algorithm is restricted to deal with discrete sets of values, the operator forces the type of each field to be `xs:string` or one of its subtypes.
The general schema for a decision tree algorithm (Algorithm 1) can be easily fitted into a recursive XQuery function. The algorithm can be decomposed into two main phases: (i) for each node of the decision tree, the computation of a table, that is a sub-set of the original dataset and (ii) the use of such a table to compute the statistical criterion at each node.

For example, consider again the decision tree of Figure 4.4. The root of the tree is assigned to the outlook attribute. Over such an attribute we have three possible partitions of the data: the records having the value of the outlook attribute equal to sunny, those having rainy and the partition with overcast values. On the three partitions, the computation continues in a recursive way. In the ID3 algorithm, the selection of the splitting attribute at each node acts by means of the information gain measure (Equation 4.2).

While the second point is quite simple, the database reduction at each recursion is hard to achieve for non main-memory strategies. A naive solution may adopt a single iteration over the data that is stored in the XML DB. On such an iteration, we evaluate each XQuery expressions to get the atomic values of the XML attributes. Then, we can use an internal in-memory table to copy the relevant portions of the entire dataset and, on such a table, we can recursively grow the tree via Java methods. The output tree is then encoded in an XML format and returned as a result. In this way, we mainly demand the entire tree construction to external functions.

Here, we prefer give a different solution that exploits path expressions to consider only the interesting tuples at each evaluation.

**Data Structures.**

For the implementation of the ID3 algorithm, we use the following data structures:

1. **gain-r**: a Java-based implementation of the gain ratio measure. The class encodes statistical information for each attribute of the reduced dataset associated to each node, also including the outcome frequencies of the target attribute. More specifically, it defines the following methods:
   (a) the constructor `new` that defines a new `gain-r` object given the number of XML splitting attributes. It initializes the data structures used to contain statistical information;
   (b) the `update` method that uses the values of the splitted attributes and of the predicted one to update data statistics;
(c) the `findBestSplit`, containing an implementation of the homonymous procedure of Algorithm 1. Trivially, the function determines the values of the gain ratio measures of each splitting attribute (Equation 4.2) according to the computed statistics. It returns the index of the attribute selected for the splitting;

(d) similarly, the `classify` and `stoppingCond` functions have the behaviour of the homonymous procedures of Algorithm 1.

(e) the `findOutcomes` method returns the outcomes of a given attribute index;

(f) finally, the `getId` procedure returns the unique identifier for any given node within the tree model and the `getRC` method returns the number of records which have been used to train this (fragment of the) tree model. They are used during the construction of the resulting PMML tree document.

2. history: an XML element to keep track of the past splitting attributes selected during the decision tree construction for a certain tree path. For each sub-element of the root tag, the index of the splitting attribute and the outcome used for the splitting are stored. For example, considering the Figure 4.4, the XML `history` element below stores the splitting information of the leftmost path (`Outlook = sunny` and `Humidity = normal`).

```xml
<history>
  <split @value="sunny" @attr-index="1">
    <split @value="normal" @attr-index="2">
  </split>
</history>
```

The `history` element is used to implement the data reduction step.

**Main procedure.**

Figure 4.5 depicts the general schema of the ID3 implementation. The procedure can be elegantly implemented by means of a `wfor` expression. The input of the `id3` function is the `history` element, initially empty.

The `wfor` iterator (line 2) scans the input transactions according to the data reduction property. This is achieved by means of an XPath expression that uses the local function `db-reduction()` to test whether the record belongs to the training set for the current node of the tree. Its trivial XQuery implementation follows:

```xml
declare function local:db-reduction($tuple, $history) as xs:boolean {
  every $i in $history/split
    satisfies (core:let-active($tuple, $i/@attr-index) eq $i/@value)
};
```
1. declare function local:id3($history as node()) as node() {

2.   wfor $t in core:for-data() [local:db-reduction(. , $history)]
3.   declare variable $gain-r := Stat:new(core:num-active())
4.   iterate Stat:update($gain-r, core:let-active($t), core:let-predicted($t))
5.   return if (Stat:stoppingCond($gain-r)) then ()
6.   else
7.     let $split := Stat:findBestSplit($gain-r) (: index of split :)
8.     return for $i in Stat:findOutcomes($gain-r, $split)
9.         return <Node id="{Stat:getId($gain-r)}"
10.            score="{Stat:classify($gain-r, $split, $i)}"
11.            recordCount="{Stat:getRC($gain-r, $split, $i)}">
12.                <SimplePredicate field="{tree:get-names($split)}"
13.                    operator="equal" value="{$i}"/>
14.                 </Node>
15.             </history>
16.             return if (stat:stop($r-gain, $i)) then ()
17.     else local:id3($new-h)}

Figure 4.5: The main procedure of the ID3 algorithm in XQuake.

At line 3 of the algorithm, the Java-encoded statistics are initialized to prepare the iteration over the selected data. This starts at line 4 in which, for each XML instance, the statistic values are updated with the values of the attributes of that instance (core:let-active) and of the target field (core:let-predicted).

The return routine (line 5) is responsible to create a new node of the PMML tree according to the best splitting attribute. First of all, the procedure determines whether all the records either belong to the same class or they have identical attribute values. In this case, it is not possible to split these records any further and the recursion for this tree path stops. Otherwise, the function determines the index of the splitting attribute that maximizes the information gain measure (line 7).

At this point, a child node is created for each outcome of such an attribute, and the records are distributed to the children based on the outcomes (lines 8-17). This is implicitly achieved by augmenting the history element with a new XML tag, <split>, that is used to remember the splitting attribute and the value of the outcome for this tree path (line 14). The algorithm is then recursively applied to each child node whenever the stopping condition does not hold (line 15).
CHAPTER 4. XML PREPROCESSING AND CLASSIFICATION

4.4 Summary

In this chapter we consider the problem of tuning the XML-based framework defined in the previous chapter in order to deal with a decision tree classifier, which is a simple yet widely used classification technique. The focus is on the construction of the model rather than on its use. Also, ad-hoc discretization and sampling techniques have been considered. We face each problem from a formulation perspective and from an implementation one.

The main features of the constructs presented here are a good flexibility to specify a variety of different mining tasks and an XQuery-like syntax that accepts a certain degree of extensibility in order to introduce user-defined functions in the statements. The preprocessing examples reported in this chapter also highlight the importance of customizing the output (at least for operators returning data) according to the user necessities, a feature typically lacking in mining languages.

The ID3 algorithm reported in Figure 4.5 is an example in which a typical divide-and-conquer approach fits into the XQuery schema of the algorithm. The general approach follows the same idea of the \( \textit{LDL}++ \) system (see Section 1.4.2). First of all, one defines the external structure, he/she identifies time-consuming operations and he/she implements them via Java. Then, he/she writes the main structure of the algorithm via wfor or via a FLWOR expression exploiting recursion when required. Data iteration, fields location and constraint evaluation are operations demanded to the “XQuery view” of the program. Updates of the data structure at each iteration are demanded to those external Java functions. Clearly, the price to pay is a certain overhead due to the Java binding mechanism for encoding input and output into the XQuery program. However, this solution has the advantage of evaluating constraints (expressed via XQuery functions) directly in the general schema of the algorithm. The Apriori implementation reported in the next chapter highlights this aspect.
Chapter 5

XML Frequent Itemsets Mining

Abstract

This chapter shows how the concepts defined in the Chapter 3 offer a viable tool for both specifying and implementing the important task of Frequent Itemsets Mining (FIM) out of XML data. There are two key issues that need to be addressed when applying frequent itemsets analysis to transaction data. First, discovering patterns from a large dataset can be computationally expensive. Second, due to the combinatorial nature of the problem, several of the discovered patterns are potentially “spurious”, making the analysis hard to interpret or simply uninteresting.

This part of the thesis is organized around these two issues. The first paragraph provides the theory and the basic implementation of the FIM task. The second paragraph deals with the issue of defining a constraint-based task for pattern discovery to prevent the generation of uninteresting result. This offers a practical example in which the domain knowledge assumes an important role to provide the domain expert with a powerful and expressive mean for driving the mining toward the required output. Concretely, we will show how XML frequent patterns can be mined by means of an XQuake operator, named MINE ITEMSETS, allowing the specification of complex objectives around the domain knowledge. Particular attention is given to the evaluation of the capability of our framework to accept optimizations proposed in the literature, thus maintaining the performance of similar works.

The chapter concludes by introducing two post-processing operators. The first one applies extracted itemsets to new data, and the second one filters the patterns according to a condition.
5.1 Frequent Itemsets Mining

5.1.1 Background

The frequent itemsets mining problem is a classical example of data summarization in which the model is simply a concise representation of the data, in the hope that it will reveal some meaningful structure.

Problem definition

The problem of mining a collection of frequent itemsets can be formalized as follows.

Definition 5.1 (Frequent Itemsets Mining problem). Let $I = \{i_1, \ldots, i_n\}$ be a set of distinct literals, called items. An itemset $X$ is a subset of $I$. If $|X| = k$ then $X$ is called a $k$-itemset. A transactional database is a bag of itemsets $D = \{t_1, \ldots, t_m\}$ with $t_i \subseteq I$. Each $t_i, i \in [1, m]$ is said transaction. A transaction $T \subseteq I$ supports an itemset $X \subseteq I$ if $X \subseteq T$. An itemset $X$ has a support $s = \sigma_D(X)$ if $s\%$ of the transactions support $X$. We indicate simply the support as $\sigma(X)$ when $D$ is clear from the context. Given a user-defined minimum support $\sigma$, an itemset $X$ such that $\sigma_D(X) \geq \sigma$ is called frequent itemset. The problem of Frequent Itemsets Mining (FIM) can be finally stated as follows: given a transaction database $D$ and a minimum support threshold $\sigma$, find all the frequent itemsets from the set of transactions w.r.t. $\sigma$. We denote by $C_{\sigma-freq}$ the minimum support constraint w.r.t the support threshold $\sigma$.

Frequent itemsets and their derivation, the association rules, find their most natural application in market basket analysis, where each transaction corresponds to a purchase, and a frequent itemset models items that are usually sold together. From the marketing perspective, such an analysis is particularly interesting for verifying how a seller’s collection of products meets the preferences of a customer. A simple example of transaction dataset over market basket analysis has been illustrated in the Table 3.4.

Beside market basket data, pattern analysis is fundamental in many application domains, including medicine, bioinformatics, web mining and so on. As an example, in the analysis of the access logs to an internet website, the association patterns may reveal interesting connections among the kind of pages visited (e.g. finance, sport, shopping, etc.) by the users. Models may reveal useful knowledge, such as: “users that frequently tend to visit the shopping and sport pages, they also tend to visit the film pages.”

Example 5.1. As a simple example, in the mondial dataset, we are interested in finding correlations among the countries that appear frequently as members of the mondial organizations. We show a fragment of the transaction database in Table 5.1. On such a dataset, the following itemset can be extracted:
Table 5.1: A sample transactional table over the mondial dataset.

<table>
<thead>
<tr>
<th>TID</th>
<th>Organiz. Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G8 {France, USA, Un. Kingdom, Russia, Germany, Japan, Italy, Canada}</td>
</tr>
<tr>
<td>2</td>
<td>UNESCO {Afghanistan, Albania, Algeria, Andorra, ...}</td>
</tr>
<tr>
<td>3</td>
<td>UNDOF {Austria, Poland, Canada}</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
</tr>
</tbody>
</table>

\{Spain, France, USA\}, \(\sigma = 0.15\).

The itemset suggests that the countries \textbf{Spain}, \textbf{France} and \textbf{USA} occurred about 15% of the times together as members of some organization.

Generally, the support is an important measure because an itemset that has very low support may occur simply by chance. As a result, typically, support is used to remove uninteresting patterns.

From the IDBs perspective, the classical FIM problem turns to be the computation of the extended theory \(Th(D, 2^{1}\mid I \mid, C_{\pi_{-freq}}, \sigma)\) [17]. As one could expect, brute-force approaches that compute the theory by enumerating every pattern and then test the support constraint \(C_{\pi_{-freq}}\) can not work because of the size of the search space. [47] shows that the problem of finding the number of frequent itemsets is \#P-hard, where the class \#P-hard is the counting counterpart of decision problems. Intuitively, this concept is clarified in Figure 5.5, where a lattice of itemsets is used to illustrate the search space: for only 4 distinct items, \(I = \{A, B, C, D\}\), we have 2\(^4\) potentially frequent itemsets.

As a consequence, a plethora of different algorithms for frequent itemsets generation have been proposed in the literature [1, 44, 50, 98, 116, 126].

The Apriori algorithm

The most popular algorithm performing the frequent itemsets generation is Apriori [1]. It is based on a level-wise search process that makes multiple passes over the data. During the pass \(k\), the algorithm finds the set of frequent itemsets \(L_k\) of length \(k\) that satisfy the minimum support requirement.

Algorithm 2 summarizes the general schema of the method. Initially, it computes the frequent itemsets of size 1 (\texttt{frequent 1 itemsets}). The core of the algorithm is then a cycle of passes each of them composed by two main phases: the candidate generation and the support counting.

1. \textit{Candidate generation}. Let \(F_k\) represent the set of frequent \(k\)-itemsets, and \(C_k\) be the set of candidate \(k\)-itemsets (potentially frequent itemsets) stored in a specialized data structure. In the candidate generation phase (\texttt{candidate generation}), the set of all frequent \(k\)-itemsets, \(L_k\), found in the pass \(k\), is used
to generate the candidate itemsets $C_{k+1}$. The candidate generation procedure ensures that $C_{k+1}$ is a superset of the set of all frequent $(k + 1)$-itemsets.

2. **Support counting.** Such a large collection of itemsets have to pass the support counting phase (support_counting) in which data is scanned. For each transaction, the candidates in $C_k$ contained in it are determined and their support count is incremented. Remember that candidate itemsets are stored in a special data structure. After the support counting, unfrequent itemsets are then dropped according to the min_supp_constraint and the algorithm terminates when $L_k$ is empty for a certain $k$.

---

**Algorithm 2** apriori$(D, \sigma)$

- **Input:** A set of transactions $D$
  
  A minimum support threshold $\sigma$

- **Output:** A set of frequent itemsets $L_1 = \text{frequent}_1\text{_itemsets}(D, \overline{\sigma})$;

  for $(k=1; L_k \neq \emptyset; k++)$ do
  
    $C_{k+1} = \text{candidate\_generation}(L_k)$;
    
    foreach $t \in D$ do
    
      support_counting($C_{k+1}, t$);
    
    end

  end

  $L_{k+1} = \text{min\_supp\_constraint}(C_{k+1}, \overline{\sigma})$;

  end

  return $\cup L_k$;

---

The Apriori algorithm uses a simple property in order to reduce the number of candidates at each iteration:

"given an itemset $I$, if $\sigma(I) > \overline{\sigma}$ then $\sigma(J) > \overline{\sigma}$ for each $J \subseteq I$".

Basically, the Apriori property means that all the subsets of a frequent itemset are frequent as well. The algorithm Apriori exploits its reverse implication to prune the search space:

"if $I \in C_k$ has a subset of length $k - 1$ that does not belong to $L_{k-1}$, i.e. this subset is not frequent, then $I$ cannot be frequent and it can be removed from $C_k$".

Note that when removing an itemset, we can also discard all its supersets.

As one could notice, the number of candidate itemsets, i.e. the size of the search space, has a critical role in determining the performance of the algorithm. Reducing the number of candidate itemsets is very difficult, basically because it is hard to predict the support of an itemset. As a consequence, an efficient and compact data structure is needed to store, update and retrieve the collection of candidates.
5.1. FREQUENT ITEMSETS MINING

On the other hand, another computational cost of Apriori is the number of transactions present in the dataset, that affects the support counting of each itemset. This cost also increases with the transaction length. The Apriori algorithm presented performs a number of iterations over the dataset that coincides with the size of the largest itemset found. At each iteration, the dataset has the same size, in terms of the number of transactions and the length of each transaction. Since in FIM problems the dataset is supposed to be stored on disk or on a database, reducing the number of iterations may be an important optimization objective. The FP-Growth algorithm below allows to reduce the number of transactions to be processed at each iteration via a divide et impera strategy.

The FP-Growth algorithm

FP-Growth [50] has the idea of dividing the search space on a prefix base. After the first scan, the original problem can be divided into \(|I|\) sub-problems, where \(I\) is the set of frequent singletons.

The algorithm acts as follows [63] (Chapter 6).

1. The dataset is scanned once to determine the support count of each item. Infrequent items are discarded, while the frequent ones are sorted in decreasing support counts.

2. The algorithm makes a second pass over the data to construct the FP-tree, that is a compressed representation of the transactions. It is constructed by mapping each transaction onto a path in the tree, following the idea that different transactions have several items in common and their paths may overlap. Clearly, the more the paths overlap, the more compression we can achieve. Figure 5.1 shows an example of FP-tree after reading all the transaction dataset.

![FP-Tree Example](image)

Figure 5.1: An example of FP-Tree (from [63]).
3. The FP-tree generated at the previous step is explored in a bottom-up fashion to generate the frequent itemsets. The key idea is that since every transaction is mapped onto a path in the FP-tree, we can derive the frequent itemsets ending with a particular item, say $e$, by examining only the paths containing $e$. These paths, called prefix paths, can be accessed quickly by using some pointers associated with the node $e$ (dashed arrows in the Figure 5.1). The extracted tree containing the prefix paths is called prefix tree for the item $e$.

4. From the prefix tree for $e$ generated at the previous step, the algorithm has to solve the subproblems of finding frequent itemsets ending with $de$, $ce$, $be$ and $ae$. To solve these subproblems, it first converts the prefix paths into a conditional FP-tree that is similar to a FP-tree, but it is used to find frequent itemsets ending with a particular suffix. In the example, the algorithm computes from the original prefix tree for $e$, the conditional FP-tree for that item (for details see [63] (Chapter 6)).

5. Finally, the FP-growth uses the conditional FP-tree for $e$ to solve each subproblem until we have no items to append and a base case is reached.

The steps from 3 to 5 are repeated in a recursive way, decomposing, each time, the problem of generating frequent itemsets into subproblems. At each recursive step, a conditional FP-tree is constructed by updating the frequency counts along the prefix paths and removing all infrequent items. Because the subproblems are disjoint, FP-growth will not generate any duplicate itemsets.

For certain transaction datasets, FP-growth outperforms Apriori by several orders of magnitude since it acts only two iterations over the dataset avoiding the candidate generation of Apriori. However, the run-time performance of FP-growth depends on the compactness factor of the dataset. If this value is low for a certain dataset, the resulting conditional FP-tree is very large and it becomes difficult to fit it directly into main-memory.

Other algorithms

The eclat algorithm [126] shares the same idea of FP-growth in dividing the search space on the basis of a prefix and, successively, in using a divide-and-conquer strategy to exploit frequent itemsets. The main difference w.r.t. FP-growth is in the data structure used to store the transaction dataset. While the latter uses the FP-tree as a compact transaction representation, eclat adopts an inverted list called tid-list in which, for each item, it stores the list of transaction identifiers containing that item. This yields a vertical data format of transactions, in contrast to an horizontal data format of FP-growth.
Other FIM algorithms, based on the splitting of the input data into smaller datasets, are partition [98], sampling [116] and disk mine [44].

5.1.2 Task formulation

In the previous section we have described the data mining task as a theory $T_h$ obtained via a selection predicate $C_{\sigma_{freq}}$ in which we are interested in itemsets having a certain minimum frequency $\sigma$. We can now adapt such a definition to our purpose, formally presenting an inductive database schema over the $X$(DM)$^2$ data model. Such a schema will serve as the basis for the specification of an XQuake operator, named MINE ITEMSET, used to extract constrained frequent itemset out of XML data.

However, in order to be as concrete as possible, throughout this section we prefer to give a slightly different presentation. We first show examples of queries within the resulting XQuake language and then we formalize the notion of the XML frequent itemsets mining task. We focus on the versatility offered by the operator here, whereas in Section 5.2 we discuss how to speed-up the computation by incorporating constraints directly into an Apriori implementation.

As a first example, we wish to discover correlations among authors in the dblp database.

Example 5.2 (co-relations among authors). In order to discover how frequently two or more authors published together, we aim at extracting frequent itemsets from the dblp dataset. In our analysis, we consider only the patterns with a minimum support of 10%. The resulting XQuake expression is the following.

```
mine itemsets doc("co-authors") using alg:apriori(0.1)
for data $paper in doc("dblp")(inproceedings|article)
for item $aut in $paper/author
let active field $aut-name := fn:concat($aut/FirstName, " ", $aut/LastName).
```

The algorithm used is the Apriori, with the relative minimum support expressed as a parameter (using clause). The set of involved XML transactions, i.e. both proceeding and journal papers, is specified through the FOR DATA clause. The XML items, i.e. `<inproceeding>` and `<article>` tags, resulting from the evaluation of the XPath expression represent the input transaction database. The next statement binds each XML node `<author>` to the variable $aut$. The LET ACTIVE clause uses a built-in function to format the author name, i.e. the atomic values in the itemset. The evaluation of the above query results in a PMML document, co-authors, encoding the required information. A fragment of such a PMML model is shown in Figure 2.2.

Intuitively, the statement allows to easily locate XML transactions and, among each transaction, the XML items. Its versatility permits easy compliance with the domain and the interpretation of the items.
Example 5.3 (co-relations among topics). Another interesting pattern analysis in the dblp dataset is to find the correlations among the research topics that have been investigated in recent years. In other words, the output contains patterns like \{data mining, databases\} supp=0.02, which states that, in the dblp database, the papers that focus on “data mining” and also on “databases” are 1% of the total.

\[
\text{mine itemsets doc(”topics”) using alg:apriori(0.01)}
\]
\[
\text{for data $paper$ in doc(”dblp”)/(inproceedings|article)}
\]
\[
\text{let active field $item := string($keyword).}
\]

Example 5.4. In the mondial dataset, we are interested in finding correlations among the religions and the government type of a country. In other words, we are interested in itemsets such as: \{Orthodox = med, Catholic = high, government = monarchy\} supp=0.35.

\[
\text{mine itemsets doc(”co-religions”) using alg:apriori(0.25)}
\]
\[
\text{for data $country$ in doc(”mondial”)/Country}
\]
\[
\text{let active field $item := local:format($religion).}
\]

Each country denotes a transaction, while an item is either the government type (sub-tag <government>) or the religion followed by a certain population percentage (sub-tags <religions>). Inside the LET clause, a user-defined XQuery function transforms the input XML element into a “categorical version” of the item.

\[
\text{declare function local:format($i as node()?) as xs:string?}
\]
\[
\text{if (empty($i)) then ()}
\]
\[
\text{else if ($i/name() eq ”government”)}
\]
\[
\text{then fn:concat(”government=”, string($i))}
\]
\[
\text{else if ($i/@percentage > 75)}
\]
\[
\text{then fn:concat(string($i), ”=”, ”high”)}
\]
\[
\text{else if ($i/@percentage < 25)}
\]
\[
\text{then fn:concat(string($i), ”=”, ”low”)}
\]
\[
\text{else fn:concat(string($i), ”=”, ”med”)}
\]

According to the definitions provided in the central part of this thesis (Chapter 3), we formalize now the notion of inductive database pattern analysis over the X(DM)\(^2\) data model.

**Definition 5.2 (XML frequent itemsets mining).** Assume an instance of an XML transaction dataset \(R_F^{<T_1,\ldots,T_n>}\) and the minimum support threshold \(\sigma \in (0,1]\). Let \(A_{R_F^{<T_1,\ldots,T_n>},G}\) be a string XML attribute over \(R_F^{<T_1,\ldots,T_n>}\) and w.r.t. an XQuery expression \(G\). The role of such an attribute is active. We define:
5.1. FREQUENT ITEMSETS MINING

- \( \mathcal{L} = \{ p \mid p \text{ is a PMML association model} \} \).
- \( q(< R_F^{<T_1,\ldots,T_n>}, A_{R_F^{<T_1,\ldots,T_n>},G}, \sigma>, p) = \text{true if and only if:} \)
  1. \( I = \text{dom}(A_{R_F^{<T_1,\ldots,T_n>},G}) \) is the set of the distinct items in \( p \).
  2. \( S = \{ i \in 2^I \mid C_{\sigma-freq}(i) = \text{true} \} \) is the set of itemsets of \( p \).
  3. \( R = \emptyset \) is the set of association rules of \( p \).

An XML inductive frequent itemsets mining task is defined by \( \text{Th}(\mathcal{L}, R, q) \).

It is quite straightforward to show that the MINE ITEMSETS XQuake operator represents a concrete model of such a schema. For instance, consider the query of the Example 5.2. The core of the operator specifies the input transaction database, \( R_F^{<T_1,\ldots,T_n>} \), with the XML transactions and items as input to the mining algorithm. This happens throughout the FOR DATA and FOR ITEM clauses that locate, respectively, the \(<\text{journal}>\) and \(<\text{inproceedings}>\) elements as transactions and, inside each transaction, the \(<\text{author}>\) sub-elements. Similarly, atomic values of the items are located by means of the definition of an active XML attribute \( A_{R_F^{<T_1,\ldots,T_n>},G} \) over the transaction dataset. In the example, items correspond to the names of the authors. Output itemsets are represented by way of a PMML document whose items are in the domain of \( A_{R_F^{<T_1,\ldots,T_n>},G} \) and whose itemsets satisfy the required minimum support constraint.

5.1.3 Task implementation

In the following we discuss the Apriori and FP-Growth implementation of the aforementioned MINE ITEMSETS operator.

Apriori implementation

In providing an efficient implementation of the MINE ITEMSETS operator via the Apriori algorithm [1], we take two critical aspects into consideration. First, we adopt optimized data structures, to speed-up the computation especially during the candidate generation and the support counting phases. Second, we integrate domain specific optimizations by encapsulating pruning strategies into the right point of the algorithm. More importantly, the pruning is automatically driven from the constraints expressed by the user. In both the cases, our goal is to exploit the best solutions available in the literature. We deal with the first issue in the remainder of this section, whereas we discuss how to speed-up the computation by means of constraints in the Section 5.2.

Data structures

In order to support complex mining operations, efficient data structures can be adopted. For the implementation of the Apriori, we use the following:
• **t-map**: a Java-encoded tree map structure used to store items. The key is the item name found in the transactions. For each distinct item encountered during the first data iteration we store: (i) the support count and (ii) for each atomic predicate of the mining constraints (HAVING clause), a boolean value indicating whether it holds on such an item (see also Section 5.2).

• **t-tree**: an external total support tree [28] used to hold the total support counts for itemsets. Basically, the t-tree imposes an ordering on items and then it enumerates the itemsets according to this ordering. The implementation can be optimized by storing levels in the tree in the form of arrays, thus reducing the number of links needed and providing direct indexing. The t-tree is implemented as a Java class.

• **t-red**: a list of integer values indicating the indexes of the transactions in the database that have been selected as “reduced” (i.e. they have been pruned from the input database). Since this data structure is very simple, we implement it directly as an XQuery sequence.

**Preparing the mining**

During the first scan of data we find frequent 1-itemsets and insert them into the t-tree. Transactions and items needed for the mining are located by means of the FOR DATA and FOR ITEM clauses, respectively. Moreover, also data constraints (expressed in the WHERE statement) and mining constraints (expressed in the HAVING statement) can be (partially) evaluated before invoking the Apriori.

We first create an instance of t-map and t-tree. Then we scan every transaction for which the WHERE clause (if any) holds, storing in t-red the indexes of pruned ones. As we scan through the items of each transaction, if the item encountered is already contained in t-map, its support count is increased by 1. Otherwise, for each new distinct item encountered, we insert it into t-map, setting the count to 1 and joining that item value with the metadata located via the let metadata clause (if any). Using such information, we are able to evaluate, during the first data iteration, the atomic predicates of the mining constraints (i.e. HAVING condition) that hold for the current item. The result of such evaluations is stored in t-map. At the end of the first iteration, each item that does not satisfy the minimum support constraint is marked as pruned. The remaining ones are used to initialize the t-tree structure.

Notice that with this kind of solution we decided to evaluate the join of the item value with the related metadata only at data preparation time, once for each distinct item. We will use this information during the next scans of the data. This is an important aspect, since the join may be expressed by means of a complex user-defined XQuery expression. The computation just described can be easily implemented through a FLWOR expression that incorporates external functions over t-tree and t-map.
1. declare function local:apriori($t-tree as java,
2. $t-map as java,
3. $t-red as xs:integer*,
4. $k as xs:integer) as node* {
5. wfor $t at $pos in core:for-data()
6. declare variable $s as empty-sequence := TTree:candidateGen($t-tree, $k)
7. iterate if (functx:is-value-in-sequence($t-red, $pos))
8. then ()
9. else let $items as xs:string* := core:unroll-items($t)
10. return TTree:suppCounting($t-tree, $items, $k + 1, $t-map)
11. return if (TTree:AM-Pruning($t-tree, $t-map, $k, $mi: min-sup) <= 1)
12. then TTree:M-Pruning($t-tree, $t-map)
13. else local:apriori($t-tree, $t-map, $t-red, $k + 1)
14. };

Figure 5.2: The main procedure of the Apriori algorithm.

Data mining
The Apriori [1] moves bottom-up in the itemset lattice, from small to large itemsets. It requires several iterations on the data, according to the size of the largest itemset. Its implementation is shown in fig. 5.2. The function recursively extends the t-tree level by level, until no more frequent itemsets can be found. It takes the (references to) t-tree and t-map as well as t-red and an integer representing the current level of the computation (initially 1). It returns the set of PMML elements representing the frequent patterns computed.

The wfor routine at line 5 scans the input transactions. The external function candidateGen($t-tree, $k) combines the frequent k-itemsets in t-tree to generate the candidate itemsets of size $k + 1, extending the total tree to a new level.

The iterate clause iterates over the input transactions for counting the occurrences of each candidate. First of all, we check whether the transaction has been previously marked as reduced (line 7). If this is the case, we skip the support counting phase for this transaction (line 8). Otherwise, we scan through each item in the transaction (line 9), traversing the total tree and recursively finding all itemsets that are supported by that item at the given level. Atomic values of the items are obtained by means of the function unrollItems of the core package (Table 3.6). The suppCounting($t-tree, $items, $k, $t-map) (line 10) external function updates, at a given level $k, the frequencies of each itemset in $t-tree according to the items $items contained in the transaction. The $t-map tree map is used to skip the support counting for infrequent items in $items.

The return routine prunes the results and tests whether no more frequent itemsets can be found. This is achieved by means of the AM-Pruning($t-tree, $t-map,
$k, ms)$ external function (line 11), that uses the anti-monotone property to filter out the candidates from $t$-tree, at the given level $k$, according to the minimum support $ms$. Observe that at this level, we can evaluate any kind of anti-monotone constraint in conjunction with the minimum support constraint (see Section 5.2.3). $t$-map can be used to check whether such a constraint holds. The function returns the number of itemsets, at the given level $k$, that survive the pruning. If there are surviving itemsets, then the Apriori algorithm is invoked recursively (line 13) to extending the total tree to a new level $k+1$ and the process is repeated until no more candidates can be found. At this point, the function $M$-Pruning($t$-tree, $t$-map) (line 12) prunes the frequent itemsets that do not satisfy a monotone constraint on $t$-tree. Again, $t$-map is used to this purpose.

**FP-Growth implementation**

**Data structures**

The algorithm exploits the following data structures:

- $t$-map: a Java-encoded structure that stores information on items during the first iteration over the data. As for Apriori, such a data structure contains the pairs ($name$, $frequency$) for each distinct item as well as the result of the evaluation of each mining constraint for that item.

- fp-tree: an external data structure to store the FP-tree. The structure is implemented as a Java class and offers methods to build a prefix tree according to the value of a given prefix and to extract a set of frequent itemsets from that prefix tree.

**Preparing the mining**

The dataset is scanned once, to determine the frequent 1-itemsets and their frequencies. At the end of the iteration, the infrequent items are discarded, and the survived ones are sorted in descending order w.r.t. their frequency. The implementation is similar to the one described for the Apriori.

**Data Mining**

The data mining step uses the fp-tree to hold a compressed main-memory representation of the input dataset. The construction of the tree acts by means of a single scan over the data. At each pass, the tree is updated according to the ordered items of the transaction. This procedure continues until all the transactions are mapped to one path of the tree, as described in Section 5.2.1. In addition, all nodes referring to the same item are linked together in a list, so that all transactions containing a specific item can easily be found and counted by traversing this list. This list is accessed through an head element, which also states the total number of occurrences of the item in the database.
The core of the FP-growth algorithm is to compute an FP-tree of a projected DB (say prefix tree), i.e. a database of the transactions containing a specific item, with this item removed. This can be implemented via a Java method. What is interesting in our context is the generation of the frequent itemsets directly from such an fp-tree by using a divide-and-conquer approach that recursively decomposes the problem into smaller sub-problems. The XQuery procedure reported in Figure 5.3 assumes to have as a parameter the prefix fp-tree, $prefix-tree$, for a certain prefix, $item$. It returns the set of frequent itemsets ending with that prefix. The function takes also the list of items, $remaining$, whose values must be recursively appended to $item$ to find the overall patterns.

**fp-growth** acts in a recursive way, finding the frequent itemsets ending with a particular suffix at each invocation. For example, suppose we are interested in extracting the frequent itemsets ending with the frequent item $e$. To do this, we must consider the subproblem of finding frequent itemsets ending with $de$, then those ending with $ce$ and so on. In turn, each of these subproblems are further decomposed into smaller subproblems and the overall solution is obtained by merging the single results of each subproblem. This divide-and-conquer approach is clear from the code of Figure 5.3 (lines 6-11). Before recursively invoking a new instance of the **fp-growth** method (line 11), we calculate the prefix trees for each subproblem $de$, $ce$, etc. (line 6) and the new list of prefixes to append in the next steps (lines 9-11).

The base case of the recursion is at line 6. The external method **freq-itemsets** builds the conditional FP-tree for a certain item from the input prefix tree. On such a conditional tree, the method extracts the itemsets ending with that item and satisfying the minimum support threshold. Moreover, at line 7, a pruning step removes those itemsets that do not satisfy a constraint. As stated before, such an
information is encapsulated in the freq-items XML structure. Notice that with this solution, every kind of constraint is evaluated at post-processing time, before merging the result.

5.2 Constraints-based FIM

Over the last few years, mining with constraints has emerged as an important research area as a support to the classical data mining techniques. Its aim is of providing a domain-dependent tool for driving the discovery process directly towards more interesting patterns. Inductive databases are therefore closely related to constraint-based mining.

5.2.1 Background

Problem definition

The fundamental idea inspiring constrained FIM mining is to introduce a set of constraints that frequent itemsets should satisfy. The analyst can use a conjunction of constraints for specifying the properties of the patterns of interest. As a trivial example, let us focus on the market basket analysis. Let’s suppose that each item is associated to a fixed price, and we are interested only on those co-occurring set of items such that their average price is higher than a given threshold. KDD systems should be able to exploit such constraints both to speed-up the knowledge process and to make the analysis more precise.

Formally, we can define the problem as follows.

**Definition 5.3** (Constraint-based FIM). A constraint is a function $C : 2^{[I]} \to \{\text{true, false}\}$. An itemset $X$ satisfies a constraint if and only if $C(I) = \text{true}$. Given a set of constraints $C = \{C_1, \ldots, C_k\}$, the constraint-based FIM problem turns to the computation of all itemsets for which $C_1(X) \land \ldots \land C_k(X) = \text{true}$.

Several classes of constraints and their properties have been identified in literature. The first interesting class is the anti-monotone [2]. It simply states that, if a predicate holds for a set $S$, then it holds for any subset of $S$. The first interesting example of anti-monotone constraint is the Apriori property, introduced in Section 5.1.1. At the opposite, the monotone one states that, if a predicate holds for a set $S$, it holds also for any superset of $S$ [25]. We can define more formally these properties.

**Definition 5.4** (Anti-monotone constraint). Given an itemset $X$, a constraint $C_{AM}$ is anti-monotone if $\forall Y \subseteq X : C_{AM}(X) = \text{true} \Rightarrow C_{AM}(Y) = \text{true}$.

**Definition 5.5** (Monotone constraint). Given an itemset $X$, a constraint $C_M$ is monotone if $\forall Y \supseteq X : C_M(X) = \text{true} \Rightarrow C_M(Y) = \text{true}$.
Example 5.5. Consider the Example 5.1 aiming at extracting correlations among the mondial countries. Each item of the mined itemsets is a country. Let us suppose that we are be able to associate to each country a fixed numeric attribute, \( \text{gdp} \), representing the gross domestic product for that country. Given an itemset \( S \), the constraint \( \max(S.\text{gdp}) > 100 \) is monotone, where \( \max(S.\text{gdp}) \) means the maximum value of the gross domestic product among the items in \( S \). The constraint \( \max(S.\text{gdp}) \leq 100 \) is anti-monotone. \( \triangleright \)

The \textit{succinct} constraint introduced in \cite{64} states that the decision concerning the satisfaction of a predicate can be determined on the basis of the single elements of the itemset.

\textbf{Definition 5.6} (Succinct constraint (from \cite{64})). An itemset \( I_S \subseteq S \) is a succinct set if it can be expressed as \( \sigma_p(I) \) for some selection predicate \( \sigma \). \( SP \subseteq 2^{|I|} \) is a succinct power-set if there is a fixed number of succinct sets \( I_1, \ldots, I_k \subseteq I \), such that \( SP \) can be expressed in terms of the strict power-sets of \( I_1, \ldots, I_k \) using union and minus. Finally, a constraint \( C_S \) is succinct provided that the set of itemsets satisfying the constraint is a succinct power-set. \( \square \)

Example 5.6 (succinct constraint). The two kinds of constraints defined in the Example 5.5 are both succinct, since a single country of \( S \) can be used as a comparator for the constraint. At the opposite, the \( \text{avg}(S.\text{gdp}) > 100 \) constraint is not succinct, where \( \text{avg}(S.\text{gdp}) \) stands for the average value of the gross domestic product among the countries in \( S \). \( \triangleright \)

There are other interesting properties for constraints in FIM, such as the class of convertible anti-monotone, \( C_{CAM} \) (resp. convertible monotone, \( C_{CCM} \)) \cite{84} and the class of loose anti-monotone constraints \cite{14}. The most important classes of constraints and their relationships are summarized in Figure 5.4.

---

Figure 5.4: Characterization of the classes of constraints in FIM (from \cite{14}).
Constraint-driven algorithms

Techniques for constraint-driven pattern discovery can be roughly classified on the basis of the KDD phase in which they are evaluated:

- pre-processing constraints are evaluated during the pre-processing phase, restricting the source data to the instances that can generate only patterns satisfying them;

- mining constraints are directly integrated into the mining algorithm (e.g. Apriori) used for extracting the rules;

- post-processing constraints are evaluated for filtering out patterns after the mining algorithm. These ones are not very interesting since, typically, the evaluation of monotone constraints entirely at post-processing time may yield an inefficient and sometimes expensive mining.

An example of pre-processing constraint is [124], that focus on improving the efficiency by using dataset filtering techniques that conceptually transform a given data mining task into an equivalent one operating on a smaller dataset. A first preliminary effort to propose a general framework for constrained frequent itemsets mining is [14], in which the authors present the design of ExAMinerGEN, a general Apriori-like algorithm, which is able to exploit all the possible kinds of constraints presented above. In particular, the authors report a data reduction algorithm, named ExAnte, for taking advantages from the conjunction of monotone and anti-monotone properties. The problem $C_{AM} \cap C_M$ has been extensively studied in the literature [58, 25, 13, 14]. For example, [25] implements a new algorithm for finding frequent itemsets, Dual Miner, that efficiently explores the search space from the top and from the bottom of the lattice, in order to exploit monotonicity and anti-monotonicity.

Other remarkable constraint-based optimizations can be found in [37] and [31]. The former proposes an “incremental” approach exploiting the results of previous queries in order to reduce the response time to new queries. In this context the term incremental describes the fact that the outcome of a new query starts from a previous result. The proposed algorithm is able to deal with context-dependent constraints. A similar idea is used in [31] in which a new query is rewritten in terms of union and intersection of the result sets of other previously executed and materialized queries. The authors propose a two-step approach. First, they find conditions to apply various types of constraints (essentially on item properties), then, by using an optimizer, they recognize equivalent queries avoiding repeated heavy computations.

Other kind of works exploit constraints to improve the comprehension of the extracted patterns with the aid of a concept hierarchy, i.e. a multilevel organization of the various entities or concepts defined in a particular domain. This line of research has been pursued in the past by [49, 103].
The ExAnte algorithm

The ExAnte algorithm [13, 14] considers the computation of the extended theory for \( C_{AM} \cup C_M \). The key idea is that a transaction that does not satisfy the monotone constraint \( C_M \), can be removed since none of its subsets will satisfy \( C_M \) either, and therefore those transaction cannot support any valid itemset. This data reduction, in turn, lowers the support of other frequent but invalid itemsets, thus reducing the search space and improving anti-monotone pruning techniques. This virtuous circle is encoded in the pseudo-code of Algorithm 3.

Algorithm 3 ExAnte\((D, C_M, \sigma)\)

**Input:** A set of transactions \( D \)
- A monotone constraint \( C_M \)
- A minimum support threshold \( \sigma \)

**Output:** A set of frequent itemsets

\( I = \emptyset \);

\( \text{foreach transaction } t \in D \text{ do} \)

\( \text{if } C_M(t) \text{ then} \)

\( \text{foreach item } i \in t \text{ do} \)

\( \sigma(i)++; \)

\( \text{if } \sigma(i) = \sigma \text{ then} \)

\( I = I \cup \{i\}; \)

\( \text{end} \)

\( \text{end} \)

\( \text{end} \)

\( \text{end} \)

old_number_interesting_items = |Items|;

\( \text{while } |\text{Items}| < \text{old_number_interesting_items} \text{ do} \)

\( \alpha\text{-reduce}(D); \)

\( \mu\text{-reduce}(D); \)

old_number_interesting_items = |I|;

\( I = \emptyset; \)

\( \text{foreach transaction } t \in D \text{ do} \)

\( \text{foreach item } i \in t \text{ do} \)

\( \sigma(i)++; \)

\( \text{if } \sigma(i) = \sigma \text{ then} \)

\( I = I \cup \{i\}; \)

\( \text{end} \)

\( \text{end} \)

\( \text{end} \)

The ExAnte procedure takes a set of transactions, \( D \), any conjunction of mono-
tine constraints, $C_M$, and the minimum support threshold, $\sigma$, i.e. the anti-monotone constraint. It starts counting the support of singleton items. At the end of the first iteration, items that are not frequent are pruned. During the first count, transactions that satisfy $C_M$ are considered. The other transactions are signed to be deleted from the database (procedure $\mu$-reduce). This implies that also the number of frequent singleton items may be reduced also. Thus, ExAnte deletes from alive transactions all infrequent items (procedure $\alpha$-reduce). This pruning can reduce the monotone value of some alive transactions, possibly inducing a violation of the monotone constraint. The while loop of Algorithm 3 captures this virtuous cycle among $\mu$-reduction and $\alpha$-reduction. The fix point is reached when no more opportunities of reducing the database nor the search space are possible.

Basically, the ExAnte algorithm performs a data reduction step typically returning a much smaller database containing only the necessary data to compute the patterns that satisfy the given constraints. After the ExAnte preprocessing, any frequent pattern mining algorithm can be run, such as Apriori or FP-Growth.

### 5.2.2 Task formulation

At this point of our work, we would like to come back to the idea of knowledge discovery as an interactive process with the user. In fact, the goal of Section 5.1.2 has been limited to discover all the patterns whose frequency in the XML transaction dataset exceeds a user-specified threshold. However, very often users want to restrict the set of patterns by adding domain-dependent extra constraints on their structure. Data mining systems exploit such constraints both to speedup the mining process and to return to the analyst a more precise mining model. To this purpose, in the following we extend the `mine itemsets` construct to handle constrained itemset mining.

**Example 5.7.** Let us consider the Example 5.2 in which co-author correlations from the dblp database are looked for. We suppose now that a user asks for a different piece of information about the extracted itemsets. In the analysis, the user considers interesting only the patterns with a minimum support of 10% in which some “leader” author occurs and in which at most two authors received a PhD after 2002. He/she considers as “leaders” those authors that have awards from the “IEEE” society or, alternatively, are prime investigators of an active project. The resulting XQuake expression is the following.

```xml
mine itemsets doc("leader-co-authors") using alg:apriori(0.1)
for data $paper in doc("dblp")/(inproceedings|article)
for item $aut in $paper/Author
let active field $aut-name := fn:concat($aut/FirstName, " ", $aut/LastName)
let metadata field $employee :=
  let $dept := collection("Dep")/Department[@id=$aut/@dep]
```


return $dept//Employee[//Name=$aut-name]

having at least 1 item satisfies
(some $award in $employee/Awards/* satisfies $award="IEEE" or
some $p in $employee/Projects/* satisfies
$p/@active="yes" and $p/@is-director="yes"),
at most 2 item satisfies $employee/PhD/@year > 2002.

The first four rows are used to locate transactions and items as in the query of Example 5.2. In addition, the user binds an XML element encoding the domain knowledge (i.e., the employee information of the department of interest to each distinct author - see also Figure A.2) to the variable $employee. This is achieved through the let metadata statement containing, in the body, an XQuery expression that first looks for the department of interest in the collection of the various departments, and then looks for the author name among its employees. Notice the use of both the $aut and $aut-name variables in the body expression. The set of itemset constraints occurs in the having clause by means of an “and” combination of the at least and at most operators.

Specifically, they constrain the number of the items of an itemset that satisfies a particular condition to have a certain threshold. They have the following format.

having at least < positive integer > item satisfies < XQuery predicate >

The operator AT LEAST (similar operators are AT MOST and EXACTLY) is true for all itemsets which have at least a specified number of items that satisfy the XQuery predicate. The latter one can be expressed on the variables previously defined. In the example 5.7, they denote both the author’s name and the employee metadata. The examples below highlight their usage.

exactly 1 item satisfies $aut-name eq "William Shakespeare"
at least 4 item satisfies fn:true()
at most fn:round($ALL div 2) item satisfies
    count($employee/Projects/Project[@active]="yes") > 1
exactly $ALL item satisfies $aut/@dep eq "Cambridge"

The first condition above finds out who publishes frequently together with “William Shakespeare”. The next clause looks for itemsets of length at least 4. The third condition imposes that at most half of the authors in the itemset are involved in at least two active projects. The special variable $ALL stands for the length of the current itemset that should be validated against the constraint. Finally, the last predicate finds correlations among the publications of the authors at “Cambridge University”.

In the above queries we have supposed the availability of metadata. In several cases, XQuery can also be used to build metadata from scratch. The variable below
stores an XML element containing the number of publications for each distinct 
author in the dblp database.

```xml
declare variable local:np as node()* :=
    for $author in distinct-values(doc("dblp")//author)
    return element publication {
        <name>{$author}</name>,
        <number>{count(/dblp/*/aut=$author)}</number>
    };
```

The fragment of the XQuake query below returns all itemsets which have “Albert 
Einstein” as an author, and at least one co-author with a number of publications 
greater than 30.

```xml
let metadata field $n as xs:decimal := local:np/number[./name=$aut-name]
    having exactly 1 item satisfies $aut-name eq "Albert Einstein",
        at least 1 item satisfies $aut-name ne "Albert Einstein" and $n > 30.
```

The predicate below aims at extracting frequent itemsets in which there does not 
either one author that is also editor.

```xml
let metadata field $is-editor := not(empty(doc("dblp")//editor[.=#$aut]))
    having exactly $ALL item satisfies not($is-editor).
```

In the next query, we adopt XQuake to solve the problem stated in the Example 
5.1, by considering the mondial dataset as a source for constraint-based frequent 
itemset mining.

**Example 5.8.** In the mondial dataset, a user is interested in finding correlations 
among the countries that appear frequently as members of the mondial organizations. 
He/she wants to restrict the output itemsets requiring that in each itemset 
least one item is an European country and that every country in the itemset 
is a republic. Also, he/she want to consider for the mining only those mondial 
organizations having at most ten countries.

```xml
mine itemsets doc("co-countries") using alg:apriori(0.35)
for data $d in doc("mondial")//organization/members[@num-members < 10]
for item $i in for $j in fn:tokenize(string($d/country)," ") return $j
let active field $f := fn:concat("country=", $i)
let metadata field $country := doc("mondial")//country[./car_code eq $i]
    having at least 1 item satisfies some $j in $country//@continent satisfies $j eq "eu",
        exactly $ALL item satisfies $country/government eq "republic".
```

Now, the metadata information is the set of <country> elements of the mondial 
dataset located by comparing the <car_code> sub-elements with the atomic item 
name stored in the variable $i.
Mining constraints can be more rigorously formalized as follows.

**Definition 5.7 (C\textsubscript{at-least} constraint).** Let \( I = \{i_1, \ldots, i_h\} \) be the set of distinct items of the transaction database \( R_F^{T_1, \ldots, T_n}\) where each \( i_j \in \text{dom}(A_{R_F^{T_1, \ldots, T_n}}, G) \) for \( j \in [1,h] \). Let \( I' = \{i'_1, \ldots, i'_h\} \) and \( I'' = \{i''_1, \ldots, i''_h\} \) be respectively the XML items and metadata elements associated to the items of \( I \). Also, let \( H \) be an XQuery predicate over \( i_j, i'_j \) and \( i''_j \), \( j \in [1,h] \) and let \( \gamma \) be a positive integer value. Given a \( k \)-itemset \( X = \{i_1, \ldots, i_k\} \), the constraint \( C^H_{at-least} \) is valid for \( X \) w.r.t. \( H \) and \( \gamma \) if at least \( \gamma \) items of \( X \) satisfy the XQuery predicate \( H \). In formula:

\[
C^H_{at-least}(X) = \begin{cases} 
\text{true} & \text{if } |\{i_j \mid H(i_j, i'_j, i''_j) = \text{true}, j \in [1,k]\}| \geq \gamma \\
\text{false} & \text{otherwise}
\end{cases}
\]

The above definition can be extended to the case of a generic number of XQuery predicates \( H_1, \ldots, H_l \) and constraint thresholds \( \gamma_1 \ldots \gamma_l \). Basically, the constraint \( C^{H_1, \ldots, H_l, \gamma_1 \ldots \gamma_l}_{at-least}(S) \) is valid for \( S \) if and only if each constraint \( C^{H_i, \gamma_i}_{at-least} \) is valid for \( i \in [1,l] \). The definition of \( C_{at-most} \) is analogous. Finally, \( C^{H_1, \ldots, H_l, \gamma_1 \ldots \gamma_l}_{exactly}(S) \) is a conjunction of the \( C^{H_1, \ldots, H_l, \gamma_1 \ldots \gamma_l}_{at-least}(S) \) and \( C^{H_1, \ldots, H_l, \gamma_1 \ldots \gamma_l}_{at-most}(S) \) constraints.

The Definition 5.2 can be easily extended to deal with these kinds of properties. Basically, the constraint-based FIM over XML data becomes the computation of the theory \( Th(D, 2^{|I|}, C_{\sigma - freq} \cup C^{H_1, \ldots, H_l, \gamma_1 \ldots \gamma_l}_{at-least} \cup C^{H_1, \ldots, H_l, \gamma_1 \ldots \gamma_l}_{at-most} \cup C^{H_1, \ldots, H_l, \gamma_1 \ldots \gamma_l}_{exactly}, \sigma) \).

The syntax of the MINE ITEMSETS operator is shown below.

\[
\text{MineItemsetsOp ::= "mine" "itemsets" "doc" "{" QName "}" ForTransactionDataClause (HavingMineItemsetsClause)\?} \\
\text{Having Mine Itemsets Clause ::= CardinalityExpr ("," CardinalityExpr)*} \\
\text{CardinalityExpr ::= ("at least"|"at most"|"exactly") ExprSingle "item" "satisfies" ExprSingle}
\]

**5.2.3 Task implementation**

The implementation of the MINE ITEMSETS operator is not straightforward and it requires appropriate mechanisms integrated in the system architecture in order both to use data structures in an efficient way and to embed domain-dependent constraints in the right point of the mining process. Constraints on itemsets can be classified according to the peculiar property they satisfy (e.g. monotone, anti-monotone, succinct, convertible, etc.). For each of these classes, there exists a specialized algorithm which is able to take advantage of such a property.

**Constraints evaluation**

In the definition of the Apriori schema (Figure 5.2) we considered the encapsulation of two important kinds of constraints: anti-monotone and monotone ones.
CHAPTER 5. XML FREQUENT ITEMSETS MINING

Figure 5.5: Monotonicity and anti-monotonicity in the MINE ITEMSETS operator

Since any conjunction of anti-monotone constraints is anti-monotone as well, this kind of constraint is exploited in conjunction with the frequency constraint, by avoiding the generation of candidate itemsets that have an infrequent subset. As mentioned before, we encapsulated such exploitation level by level in the external function \texttt{AM-Pruning} of the Apriori schema. The monotone constraints are exploited at post-processing time on the total tree \texttt{t-tree} before encoding the output itemsets in the PMML model (external function \texttt{M-pruning}).

Two essential questions arise: (i) “Are the constraints defined in the \texttt{having} clause monotone or anti-monotone”? (ii) “How can the MINE ITEMSETS operator exploits monotone constraints in conjunction with the anti-monotone constraints”? In other words, “how is it possible to use a more efficient synergy among these two kinds of constraints”?

In order to answer the first question, Figure 5.5 illustrates an example on how the various kinds of constraints work. Given the itemset lattice from the set of items $M = \{A, B, C, D\}$, the figure shows the portion of the search space that each kind of constraint satisfies over a given generic condition \texttt{cond}. The following proposition holds directly from the definition of monotonicity (Definition 5.5) and anti-monotonicity (Definition 5.4):

\begin{proposition}
The $C_{\text{at-least}}$ constraint is monotone. The $C_{\text{at-most}}$ constraint is anti-monotone. The $C_{\text{exactly}}$ can be treated as a conjunction of the $C_{\text{at-least}}$ and $C_{\text{at-most}}$ constraints. \hfill \Box
\end{proposition}

On the other hand, the answer to question (ii) is provided by the constrained
5.2. CONSTRAINTS-BASED FIM

Figure 5.6: The Ex-Ante procedure at the first level of the level-wise computation.

ExAnte implementation

The ExAnte property states that a transaction which does not satisfy the given monotone constraint can be deleted by the input database since it will never contribute to the support count of any solution (µ-reduction). A consequence of reducing the input database is that it implicitly reduces the support of a large amount of itemsets that do not satisfy the monotone constraint as well, thus resulting in a reduced number of candidate itemsets generated during the mining algorithm. In addition, for the same anti-monotonic property, infrequent singleton items can be deleted from all transactions in the input database (α-reduction), thus reducing the probability that a transaction satisfies a monotone constraint.

In Figure 5.6, we provide an XQuery implementation of the ExAnte procedure as a preprocessing task. The function takes as input $t$-map and $t$-red initialized during the mining preparation phase (Section 5.1.3). Moreover, also the current number of itemsets mining literature. In fact, the evaluation of monotone constraints entirely at post-processing time may yield an efficient and sometimes expensive mining and we have seen that several proposals exploited the synergy between anti-monotonicity and monotonicity [58, 25, 13, 14]. Here, we implement the preprocessing data reduction algorithm ExAnte [13], recently proposed by Bonchi et. al. as a means to reduce both the search space and the input dataset in FIM.

1. declare function local:ex-ante($t$-map as java-object, $old-n$-items as xs:integer, $t$-red as xs:integer*) as xs:integer* {
2.     wfor $t$ at $pos$ in core:for-data()
3.     declare variable $reduced$ := $t$-red
4.     init $reduced$ := let $i$ := j-items:reinitSupports($t$-map)
5.     return $reduced$
6.     iterate if (functx:is-value-in-sequence($reduced, $pos))
7.         then $reduced$
8.         else let $items$ := mi:unroll-items($t$)
9.         return if (j-items:M-pruning($t$-map, $items))
10.     return let $n$-freq := j-items:pruneItems($t$-map, $fi$:min-sup)$
11.         return if ($n$-freq < $old-n$-items)
12.         then local:ex-ante($t$-map, $n$-freq, $reduced$)
13.         else $reduced$
14.     return let $n$-freq := j-items:pruneItems($t$-map, $fi$:min-sup)$
15.     return if ($n$-freq < $old-n$-items)
16.         then local:ex-ante($t$-map, $n$-freq, $reduced$)
17.         else $reduced$
18. };
frequent items is recorded. During the declaration phase, the state variable contains the starting list of transactions doomed to be removed (line 5). Then, we set the support count of frequent items in \texttt{t-map} to 0 (lines 6-7).

Each transaction index is then tested against the state variable to check whether it has been already pruned (line 8). Only if the transaction passes this test, it is then tested against the conjunction of monotone constraints (line 11). The external function \texttt{M-pruning($t$-map, $\texttt{items}$)} first deletes from the overall items in the transaction, the ones that are infrequent (as stated before, this information is encoded in \texttt{$t$-map}). This is the $\alpha$-reduction phase. Then, the conjunction of monotone constraints is tested against the surviving items of the current transaction. If they do not pass this test, such a transaction is marked as reduced (line 12). This is the $\mu$-reduction phase. Otherwise, the support count of survived items in the transaction is increased by 1.

As soon as the input dataset is entirely checked, we filter out the items that are infrequent (line 14) according to the anti-monotonicity of the support constraint. At the end, the \texttt{$n$-freq} variable stores the overall number of frequent items. After this reduction of items, we have the opportunity of recursively reducing again the dataset (line 16) until a fix point is reached and no more items are found to have turned infrequent (line 17). The \texttt{ex-ante} function updates \texttt{t-map} at each recursive invocation and it returns the new list of indexes of the reduced transactions. At this point, the \texttt{apriori} function of Figure 5.2 runs on such data structures and it computes the overall frequent itemsets.

### 5.3 Post-processing enhancement

This section includes two post-processing operators for XQuake. The first one applies a set of mined patterns to new data. The second one filters a collection of frequent itemsets according to a context-based condition.

#### 5.3.1 Frequent itemsets application

An extremely important feature in database mining is the ability to correlate the generated knowledge with further data. In fact, data is not only an input to the KDD process, but also an intermediate and a final output. Even if the association model has mainly a descriptive usage, an interesting operator includes mechanism for the application of the extracted patterns.

For this purpose, we first need to define when an itemset supports or violates a given transaction.

**Definition 5.8.** Given an itemset $I = \{I_1, \ldots, I_n\}$, and a transaction $T$, we say that $T$ supports $I$, i.e. $\text{supp}(T, I)$, if $\forall i \in [1,n]: I_i \in T$. Conversely, we say that $T$ violates $I$, i.e. $\text{viol}(T, I)$, if $\forall i \in [1,n]: I_i \notin T$. 

\[\checkmark\]
The **APPLY ITEMSETS** operator in XQuake provides the capability of connecting the extracted itemsets with new XML data. It takes as input a PMML association model and an XML transaction dataset. For each single transaction, the operator gives information about the number of itemsets that are supported or violated by that transaction.

**Example 5.9.** Let's suppose to have the following XML fragment encoding information about new future mondial organizations:\(^1\)

```
<organizations>
  <org @name="org1" @establishing-date="...">
    <country @name="country1">
    </country>
    <country @name="country2">
    </country>
  </org>
  ...
  <org @name="org2" @establishing-date="...">
    ...
  </org>
  ...
</organizations>.
```

With reference to the itemsets extracted in the Example 5.8, the following XQuake expression returns as output the set of triplets, \((o, v, s)\), where \(o\) is the organization name and \(v\) (resp. \(s\)) is the percentage of the itemsets that violate (resp. comply with) \(o\), over the total number of itemsets.

```
apply itemsets doc("supported-countries")
for association model $m in doc("co-religions")
let metadata field $total := pmml:get-number-itemsets($m)
for data $org in doc("new-org")/org
  for item $country in $org/country
    let active field $item := fn:concat("country=",$country/@name)
    having supported $s, violated $v
    return ($org/@name, ($v div $total) * 100, ($s div $total) * 100)
```

After locating the input source (itemsets, transactions and items), the semantics of this operator amounts to an iterative execution over the input transactions. For each `<organization>` element, the operator binds the number of itemsets that comply with (resp. violate) that transaction to the variable \($s\) (resp. \($v\)). The RETURN clause uses such an information to produce the output triplets.

The general syntax of the operator follows.

```
ApplyItemsetsOp ::= "apply" "itemsets" "doc" "{" QName "}" ForModelClause
  ForTransactionDataClause HavingApplyItemsetsClause+
```

---

\(^1\)Notice that the structure of such an XML document is different w.r.t. the mondial XML database.
ReturnClause

HavingApplyItemsetsClause ::= "having" SupportedExpr ("," ViolatedExpr)?

SupportedExpr ::= "supported" "$" VarName

ViolatedExpr ::= "violated" "$" VarName

The core are the clauses ForModelClause and ForTransactionalDataClause, used to locate the input PMML model and the transaction dataset, $R_F^{<T_1,...,T_n>}$, respectively.

The user preferences are expressed by means of the HavingApplyItemsetsClause. For each transaction $t_i$, the operator automatically binds the values $\sigma$ and $\bar{\sigma}$ to the user-defined variables declared in the SupportedExpr and ViolatedExpr (if any) clauses, where $\sigma = |\{i_j | supp(t_i, i_j), j \in [1, m]\}|$ and $\bar{\sigma} = |\{i_j | viol(t_i, i_j), j \in [1, m]\}|$.

The returnClause expression is invoked for each transaction $T_i$ of $R_F^{<T_1,...,T_n>}$ by evaluating the XQuery statement in the right part of the clause. As shown in the example above, the FOR DATA variable, the variables defined in the FOR ASSOCIATION statement as well as the HAVING clause are in the scope of the RETURN expression.

5.3.2 Frequent itemsets filtering

Given a collection of mined patterns, the FILTER ITEMSETS operator returns those satisfying a specified condition. The condition is expressed by using an XQuery predicate, that concerns both the atomic values of the items contained in the itemset and the related metadata.

Example 5.10. Let’s consider the patterns extracted in the Example 5.2. The goal is to return only those having “Enrico Fermi” as an author, containing at least three items, and finally containing some author whose number of projects is twice than every other author in the itemset.

```
filter itemsets doc("filtered-authors")
for association model $co-authors in doc("co-authors")
let metadata field $projects as node() :=
collection("Dep")//Employee[PersonalInfo/Name eq $ITEM]/Projects
having fn:contains($ITEMSET, "Enrico Fermi") and
  count($ITEMSET) > 2 and
  some $i1 in $ITEMSET->$projects
  satisfies (every $i2 in $ITEMSET->$projects
    satisfies count($i1/*) > 2 * count($i2/*))
```

This operator first searches in the collection of departments the <Projects> XML element (i.e. the metadata) of each distinct author (i.e. the item). The variable $projects binds such a result. Then, the XQuery predicate placed in the HAVING clause is evaluated for each pattern. Only the patterns for which the predicate returns true are returned as answer.
5.4. Summary

In the example above, it is important to note the use of the special variables $ITEM$ and $ITEMSET$. The former binds to the value of each distinct item (in this case the author name). The latter represents a string sequence containing the atomic items of the itemset for which we are testing the condition. Note moreover that another special syntax is used to bind each metadata (in this case the <$Projects$> XML element) to each item of the itemset. The special notation $ITEMSET->$projects means that a sequence of <$Projects$> nodes (one for each item) for that itemset has to be returned.

The general syntax of the operator is reported below.

\[
\text{FilterItemsetsOp ::= "filter" "itemsets" "doc" "{" QName "}" ForModelClause HavingFilterItemsetsClause}
\]

\[
\text{HavingFilterItemsetsClause ::= ExprSingle}
\]

5.4 Summary

Mining of frequent patterns in large data collection is an interesting challenge for several different reasons.

- The task offers a basis for other important methods in data mining, like association rules, sequential patterns and graph mining that allow the discovery of interesting structures among the data.

- The combinatorial nature of the problem may cause the mining of thousands uninteresting results. For this reason, the quality of the result has to be accurately evaluated and the topic of finding frequent patterns under domain-specific constraints is one of the hottest in the research community.

- Integrating the mining capability into existing database technology is also one of the most studied research area in the field of inductive databases. From its introduction, for example, several languages have been developed to provide the capability for mining association rules in relational database systems.

In this chapter, we propose a new mining operator as a solution to the XML FIM problem. The syntax of such an operator is flexible enough to specify a variety of different mining objectives by means of XQuery expressions in the statements. These ones provide personalized constraints, based, for example, on domain knowledge. To justify our findings, we build a recursive implementation of the Apriori, FP-growth and ExAnte algorithms, demonstrating that a synergy among several kinds of domain-dependent and physical optimizations is possible.

The key idea is that, while the domain knowledge is linked to items through XML elements, items are denoted by using simple structured data from the domains of basic data types. This favours both the implementation of efficient data structures and the design of powerful domain-specific optimizations evaluated as deeper as
possible in the extraction process. More important, the process of constraint integration acts in an automatic way by analysing their properties such as monotonicity or anti-monotonicity.

However, we believe that a performance analysis needs to be addressed with particular attention. In this respect, the Apriori algorithm may be an important test bed and the performance of the algorithm in our implementation should be comparable to the performance of similar works.

We conclude highlighting that the importance of constraint-based data mining is not only related to the FIM task, but it plays a crucial role also for classification or clustering tasks. This may be a promising research field in the near future.
Chapter 6

System architecture and performance evaluation

Abstract

This chapter presents XQuake from the physical perspective describing the implementation details of the system architecture over an existing native XML database. The architecture consists of layers for data, models and domain knowledge repository, query interpretation and compilation. The XQuake system also includes a simple GUI for a user friendly input of queries and for browsing the results. The subdivision in four main layers reflects the four main functionalities of our tool: query preparation, mining, staging and visualization of the extracted patterns. According to the requirements reported in Chapter 1, the overall architecture is designed in order to provide a good maintainability and extensibility of the software.

In presenting the system, we also envisage future substantial optimizations. In fact, tailoring of optimization techniques to mining queries is a major research topic in each database-oriented approach. However, in our context this is even more substantial, since it is well-known that XML-based databases cannot compete, in terms of performance, with relational databases. From this perspective, as a final contribute of the chapter, we measure the performance of the system, comparing the performance of the Apriori algorithm with an existing solution based on a different approach.

Also, we complete our research proposing a concise, yet comprehensive set of queries which covers the major aspects of the data mining. Queries are designed over the well-known X-mark XML database, that is a scalable benchmark dataset modeling an Internet auction site.
6.1 The XQuake system prototype

The XQuake prototype is implemented in Java. The system is built on top of BaseX, an open source native XML database developed by the Database and Information Systems Group at the University of Konstanz [52]. BaseX has been chosen among several XQuery implementations, thanks to its compact storage structures, its efficiency and a visual front-end, facilitating interactive access to the data. In order to implement the \texttt{wfor} iterator presented in Section 3.3, several adaptations were made to the BaseX engine involving the parser and the run-time system.

The XQuake architecture is schematically shown in Figure 6.1. A mining task is expressed in XQuake via a specific \textit{text editor}. No matter what kind of GUI the system adopts, a \textit{compiler} automatically generates the appropriate (extended) XQuery code that is then interpreted. The core of the mining process is performed by the \textit{mining engine} that contains the run-time support of the BaseX XQuery engine extended with the \texttt{wfor} iterator. This component uses the \textit{external function module} responsible for providing an XQuery interface to external user-defined Java functions over data structures. At the bottom, we have the BaseX native XML database containing the input and output of mining tasks, as well as XML metadata. Finally, the \textit{XML visualizer module} translates an XML document stored in the DB into a visualization form, accepted by data and model visualization tools, and presented by means of an \textit{output GUI} to the user. Currently, the XML result is transformed into HTML browsable format via XSLT style sheets [113].

Summarizing, XQuake has a four-layer system architecture composed of:

1. the database layer,
2. the mining engine layer,
3. the compiler layer,
4. the GUI layer.

In the following we give an overview of the design of each layer, commenting, when necessary, how the requirements of a support for an efficient algorithm programming and of modularity are addressed (see Section 1.3.3).

6.1.1 The database layer

The bottom layer manages the read/write access to data, models and background knowledge repositories. The database is accessed by means of the XML access optimizer component that contains native indexing techniques to speed-up the access to input data. In fact, a well known way to improve query execution time is to build indexes. A great effort has already been done in the research for indexing semistructured data and XML. BaseX, for example, uses text nodes indexes, attribute indexes and full-text indexes to speed-up the computation. The former allow a speedup of order of magnitudes for text-based queries. They are based on a B-Tree. In contrast, the full-text indexes are optimized to support all full-text features of the XQuery Full-Text recommendation and they are implemented as a sorted array structure.

At the time of writing, the encapsulation of indexes is automatic: as soon as the mining query terminates, the result is serialized into an XML document, and such a document is then mapped into the native database by applying all the available indexing strategies. We emphasise however that the process of integrating indexes may also be manual, i.e. encapsulated into the XQuake syntax, to give to the users the capability to customize their indexes.

For example, we can think to extending XQuake with a new statement, say make index, that applies a specified index to the output result, in order to speed-up future computations. The following query generates an explicit attribute index on the support value of the extracted patterns.

\[
\text{mine itemsets doc("co-authors") using alg:apriori(0.1) for data $paper \in \text{doc("dblp")/(inproceedings|article) for item $aut \in $paper/author let active field $aut-name := fn:concat($aut/FirstName, ", $aut/LastName) make attribute index on support.}
\]

The goal is to optimize future XPath predicates such as,

\[
doc("co-authors")/PMML//Itemset[@support >= 0.5 and @support <= 0.8].
\]
6.1.2 The mining engine layer

The mining engine layer contains the run-time support of the BaseX XQuery kernel. Two main extensions have been made to support efficient mining tasks: the implementation of the \texttt{wfor} iterator (Section 3.3) and the integration of external Java methods (Section 3.3.4).

\textbf{wfor run-time support module}

We recall the \texttt{wfor} iterator in its general schema.

\begin{verbatim}
  wfor $binding-variable in < sequence >
      declare variable $state-variable as < type > := < expression >
      init < expression >
      iterate < expression > while < condition >
      return < expression >
\end{verbatim}

where \texttt{< expression >} is a FLOWR expression or a nested \texttt{wfor}.

As stated in Section 3.3, the \texttt{wfor} statement iterates over an input sequence of length \(N > 0\) and it binds a variable to each item of the sequence. The effect of the \texttt{declare} clause is to introduce the \texttt{$state-variable} and to initialize it with the value of the given expression body. The state variable is in the scope of all the rest of the \texttt{wfor} expression. Also, it is updated at each iteration with the result of the \texttt{iterate} clause, whose aim is to consume the sequence item by item. The \texttt{while} breaks the cycle and forces the execution of the \texttt{return} clause, that returns an output value. At this point, if additional items exist in the input sequence, the computation continues by re-initializing the state variable with the result of the \texttt{init} statement (if any) and by evaluating \texttt{iterate} again.

The core algorithm of the \texttt{wfor} interpreter is reported in Figure 6.2. The evaluation of each XQuery/XPath expression acts by means of sequence iterators. In brief, we provide a Java method, \texttt{iter(ctx)}, that evaluates the expression and returns an iterator on the resulting items (line 84). The method takes as input an instance of the context, i.e. the environment of the declared variables, and it returns a new instance of the \texttt{Iter} class. It starts declaring three fields for such a Java class: the iterator over the stream of the input sequence, the exit condition initialized to false, and the tuple counter (lines 88-90).

Then, the implementation overrides the method \texttt{next()} producing the output stream. Basically, the latter returns the next item or \texttt{null} if no other items are produced. It the input stream is empty, we return an empty sequence setting the \texttt{exit} condition to true (lines 96-99). Otherwise, we evaluate both the body of the declare statement (line 101) and the \texttt{init} clause (if any). The core of the computation is at lines 104-111. It consists in a \texttt{do-while} cycle that evaluates the \texttt{iterate} clause until the evaluation of the \texttt{while} expression (if any) returns true. When this happens, we return the resulting item of the \texttt{return} evaluation (line 112). The termination condition over the input stream is tested at line 107.
6.1. THE XQUAKE SYSTEM PROTOTYPE

Java binding module

The Java binding feature is a mechanism allowing direct calling of Java methods bound as XQuery functions and manipulation of wrapped Java objects. As shown in the previous chapters, this capability opens a vast range of possibilities to implement and manage in an efficient way external data structure via XQuery.

The extension mechanism introduced in BaseX is similar to the mechanism provided by several other XQuery engines like Qizx [115] or Saxon [99]. The implementation performs many automatic conversions, including Java arrays. Basically,
a function name in which the namespace URI starts with java is automatically treated as a call to a Java method.

Java classes and methods are dynamically retrieved via Java reflection. Three different cases are considered: static methods, constructors, and instance methods. Static methods can be called directly. The local name of the function must match the name of a public static method in this class. Java constructors are called by using the function named new(). Finally, instance methods are called by supplying an extra first argument of type Java Object that is the object on which the method is to be invoked (see Section 3.3.4 for some examples of calls to external method).

Parameters are automatically converted from XQuery basic types to Java basic types. Conversely, the return value is converted from Java type to an XQuery type. Moreover, XQuery sequences of basic types are converted to arrays of basic types, and viceversa. The type conversions of Table 6.1 explain their usage.

<table>
<thead>
<tr>
<th>XQuery type</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:string</td>
<td>String.class</td>
</tr>
<tr>
<td>xs:boolean</td>
<td>boolean.class, Boolean.class</td>
</tr>
<tr>
<td>xs:byte</td>
<td>byte.class, Byte.class</td>
</tr>
<tr>
<td>xs:short</td>
<td>short.class, Short.class</td>
</tr>
<tr>
<td>xs:int</td>
<td>int.class, Integer.class</td>
</tr>
<tr>
<td>xs:integer</td>
<td>long.class, Long.class</td>
</tr>
<tr>
<td>xs:float</td>
<td>float.class, Float.class</td>
</tr>
<tr>
<td>xs:double</td>
<td>double.class, Double.class</td>
</tr>
<tr>
<td>empty-sequence</td>
<td>void</td>
</tr>
<tr>
<td>org.basex.query.xquery.item.Node</td>
<td>node()</td>
</tr>
<tr>
<td>xs:string*</td>
<td>String[]</td>
</tr>
<tr>
<td>xs:integer*</td>
<td>int[], Integer[]</td>
</tr>
<tr>
<td>xs:double*</td>
<td>double[], Double[]</td>
</tr>
</tbody>
</table>

Table 6.1: XQuery vs. Java types.

Reflection is a very powerful mechanism, and its usage well fits with our approach. However, it should not be used indiscriminately. In fact, the main drawbacks of reflection are two:

- Reflection requires a run-time permission which may not be present when running under a security manager. This is in an important consideration for code which has to run in a restricted security context.

- The more important drawback is due to performance overhead. Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, especially for data-intensive applications.
The second point above brings to the dilemma whether external Java-based data structures may outperform the adoption of XML-encoded data structures. Clearly, implementing complex data structures by way of XML and managing them via XQuery while maintaining a good level of efficiency is a very complex task.

### 6.1.3 The intermediate layer

The intermediate layer contains the XQuake compiler and optimizer as well as a module responsible to translate the XML output into a visual format.

#### Query compiler

The *compiler* module generates the appropriate (extended) XQuery code that is then executed. It is designed to provide a good level of extensibility of the environment whenever the definition of a new algorithm is provided. This is addressed by splitting the implementation of each XQuake operator into a *dynamic program* and a *static program*. Only the first component is generated by the compiler since it is independent from the mining algorithm used and from the kind of implementation provided.

Essentially, the dynamic component encapsulates the user-defined code fragments of each XQuake statement into XQuery functions or variables. As an example, consider the Figure 6.3 that shows the compilation process for the `PREPARE DISCRETIZATION` operator (see Section 4.1). The general structure of the operator is reported on the left. As a result, the compiler produces two XQuery library modules\(^1\) depicted on the right side of the figure:

- The *core* module (top-right rectangle in the Figure 6.3) contains the implementation of the functions described in Table 3.6, that are common to each XQuake operator. More specifically, variable/function declarations before the XQuake operator are mapped into homonym variables and functions in the output program. Then, each function is generated according to the user-defined code of the XQuake operator. For example, in each operator, the *let-active* function addresses the XQuery code contained in the *let active* XQuake statement, with the variable name that coincides with those declared in the left-part of the *for data* statement.

- The *discr* component is an additional module generated for the discretization operator. It contains specialized functions for the task under consideration. For example, the function `discr:bins($j)` returns the user-defined bins for the discretization variable of index $j$.

---

\(^1\)XQuery functions and variables can be put in library modules, which can be imported by any query. Library modules play the same role of the packages in object-oriented languages.
Group attributes (see Section 3.2.2) are treated at compilation-time in a similar way, as explained in the example below. Figure 6.4 highlights the role of the translator.

Example 6.1. Consider the query of Example 4.4, whose code is reported below for the sake of readability.

```xml
prepare discretization doc("human-density") using alg:equal-frequency()
for data $country in doc("mondial")//country[count(./province) >= 10)]
let group<10> field $r := top-10-density($country)
```
having bin $r-d in ("low-density", "high-density")
return <r @n="{$country/name}">
  {for $i in $r-d return <d>{$i}</d>}</r>

The following XQuery functions are generated by the translator for the core package.

```xquery
declare function core:top-10-density($country) as xs:string* {
  let $r := let $ord := for $i in $country/province
    order by count($i/city) descending
    return $i/population div $i/area
  return for $j in (1 to 10) return $ord[$j]}
};
```

```xquery
declare function core:for-data() as node()* {
  doc("mondial")//country[count(./province) >= 10]}
};
```

```xquery
declare function core:let-active($country as node, $RESERVED1 as xs:int)
  as xs:string {
  let-active-fields($country)[$RESERVED1]
};
```

```xquery
declare function core:let-active-fields($country as node) as xs:string* {
  core:top-10-density($country)
};
```

```xquery
declare function core:num-active-fields() as xs:int {
  10
};
```

Notice the usage of variables having a reserved namespace. Functions for the discr package are similar.

Summing up, functions and variables generated by the translator are invoked by the static program when necessary. The latter is implementation-dependent, but it is not affected by the user-defined code of the XQuake operator. A couple of examples of such a program for the discretization operator have been shown in the
Figure 4.1 and in the Figure 4.2. As one could expect, adding a new mining or preprocessing algorithm to an existing language operator only requires the design of the static component, so the compiler is not affected by this integration.

**Query optimizer**

The translator box should also include ad-hoc *mining-query optimizers* that exploit the kind of query in input and the various constraints to generate access plans that guarantee a good level of performance. The evaluation of an execution plan has to be studied in deep in the near future, since it is worth substantial optimizations.

In our view, optimization strategies can be classified according to several dimensions. The list below is not exhaustive, but provides some example of possible enhancements to our system architecture.

- **Data and mining constraints.** The exploitation of the properties of mining constraints, such as monotonicity or succinctness, typically acts at run-time for pruning the search space or the input database. An example of such an optimization has been considered in detail in Section 5.2.1 and implemented in Section 5.2.3. As mentioned earlier, further kinds of constraints need to be studied, especially those based on a concept hierarchy that well fits both to the patterns analysis and to classification trees.

- **Query rewriting.** The exploitation of the result of the previously executed queries has been studied in [37, 31]. These approaches aim at reducing the time spent for performing a series of subsequent queries in which the element that changes incrementally in queries are the constraints specified by the user. In these cases, the optimizer recognizes if the current query is contained in a previous one and some “incremental” algorithms are launched in order to “adjust” previous result to the current query [72].

- **XQuery-specific optimizations.** XQuery-specific optimizations are strictly related to our context. The efficient processing of XQuery queries has become a new research topic and a few researchers have proposed the methods for optimizing XQuery queries to resolve the problem of efficiency [75]. Even if this research field is outside of the scope of this thesis, we think that a closer look at the optimization problems that are associated with the XQuery language is necessary.

### 6.1.4 The graphical user interface layer

The XQuake system includes a GUI for user friendly input of queries and for browsing the extracted knowledge. See Figure 6.5 for a snapshot of the input GUI. Basically, the GUI of the BaseX system has been modified to allow for:

- opening and modifying an existing query;
• creating a new query through a syntax driven editor;
• executing a validated query;
• transforming query results into HTML browsable format via XSLT style sheets.

Figure 6.5: The GUI of the XQuake system.

However, strictly speaking, the GUI is not part of the core system. In fact, XQuake queries may be generated by other programs, such as vertical applications. Moreover, a data miner may prefer to use flexible and more sophisticated GUIs. For example, in relational technology, QBE (Query By Example) uses visual tables where the user would enter commands and conditions. It provides a powerful way to perform queries without having to know a query language. In a similar way, a DM query language may serve as a kernel for DM system implementations, on top of which various kinds of GUIs may be developed. A future extension may be the implementation of a GUI according to the QBE philosophy.

Also, visual languages, such as [59, 45, 32, 106, 81, 40, 82, 102, 89], are natural higher abstraction layers. The visual metaphor of the KDD process as a graph of tasks (nodes) and flows (arcs) allows for intuitive modelling of the KDD process and for rapid prototyping. We can think of visual languages as friendly interfaces that compile graphs into one or more XQuake query.

6.2 Performance evaluation

In the following we analyze the performance of the Apriori algorithm, implemented as described in Section 5.1.3. In order to compare the performance with an ex-
isting Apriori implementation, we test the effects of a very simple XQuake query, without considering the encapsulation of any kind of constraint. In Section 6.2.2, a more complete mining scenario designed over a scalable Internet auction database is presented.

6.2.1 Testing the Apriori algorithm

We used both real and synthetic datasets stored in the native database BaseX. The real datasets are from the UCI repository [117]. The synthetic databases are generated by means of the IBM generator [55]. Data generation can be tuned according to the following parameters:

- $T$, the average number of items in the transactions;
- $I$, the average number of items in the itemsets;
- $D$, the number of transactions in the database;
- $N$, the number of items.

These datasets are named “TxIyDz” according to the parameters $x, y$ and $z$.

In evaluating a pattern mining system it is very important to test different datasets, since each of them raises different difficulties for a mining algorithm. Mainly, the choice of the datasets is motivated by the need to estimate the contribution of the XQuake approach from dense datasets (Connect and Mushroom) to more sparse ones (T20I6D300K). On the one hand, dense datasets have typically long transactions, producing in this way a large number of frequent itemsets. On the other hand, sparse datasets are usually easier to mine. For these datasets, however, we used an higher number of transactions and a lower level of the minimum support, estimating the scalability of our approach w.r.t. the size of the dataset. A summarization of the databases used in our experiments is given in Table 6.2, in which the last two columns identify the lower minimum threshold of the support used in the experiments and the number of extracted patterns.

We carried out our tests on a dual core Athlon 4000+ running Windows XP. We assigned 1.5Gbyte of memory to the Java Virtual Machine. In order to have an idea of the performance of XQuake, we compared its execution time on the three real datasets with those obtained by running the Java-based Weka system [106]. Weka is based on a very different architectural choice aiming at providing a main-memory Apriori implementation that reads data from a text file. Below we report a fragment of an XML database used to test our Apriori together with the MINE ITEMSETS operator for pattern extraction.

```xml
<data db_name="census">
  <transaction id="1">
    <item>age=1</item>
```
### 6.2. PERFORMANCE EVALUATION

<table>
<thead>
<tr>
<th>Name</th>
<th>Real</th>
<th>Avg trans</th>
<th>Num trans</th>
<th>Num items</th>
<th>Min supp (%)</th>
<th>Num itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>census</td>
<td>yes</td>
<td>15</td>
<td>48,841</td>
<td>135</td>
<td>2</td>
<td>70,826</td>
</tr>
<tr>
<td>mushroom</td>
<td>yes</td>
<td>23</td>
<td>8,122</td>
<td>119</td>
<td>10</td>
<td>574,513</td>
</tr>
<tr>
<td>connect-4</td>
<td>yes</td>
<td>43</td>
<td>67,556</td>
<td>129</td>
<td>88</td>
<td>55,115</td>
</tr>
<tr>
<td>T20I6D100K</td>
<td>no</td>
<td>20</td>
<td>100,000</td>
<td>1,000</td>
<td>0.2</td>
<td>25,820</td>
</tr>
<tr>
<td>T30I10D100K</td>
<td>no</td>
<td>30</td>
<td>100,000</td>
<td>1,000</td>
<td>0.2</td>
<td>108,634</td>
</tr>
<tr>
<td>T20I6D300K</td>
<td>no</td>
<td>20</td>
<td>300,000</td>
<td>3,000</td>
<td>0.2</td>
<td>7,457</td>
</tr>
</tbody>
</table>

Table 6.2: Summary of datasets for experiments.

```xml
<item>workclass=Private</item>
...
</transaction>
<transaction id="2">
...
</transaction>
</data>

```

Obviously, in the XQuake system, it is possible to define more complex mining queries. However, such queries are hardly definable by using different implementations\(^2\). Hence, a comparison of XQuake with other architectural choices is significant only without considering constraints.

Figure 6.6 illustrates the performance obtained on the datasets by varying the value of the minimum support. We modified the Apriori source code of Weka to serialize output patterns in a text file and to exclude the time for generating the association rules. Similarly, the performance illustrated in the graphs includes for XQuake the time to produce an XML file, but excludes the time needed to store such XML into the database.

The first group of experiments is good and promising, thanks both to the excellent performance and indexing technique of BaseX and to the efficient data structures used to hold the support counts of the itemsets [28]. When the mining becomes hard, XQuake outperforms Weka and the differences between the two implementations tend to increase with respect to lower values of the minimum support threshold. The scalability is also acceptable on artificial datasets, on which the performance of the algorithm is quite stable.

\(^2\)And often not definable at all, such as with the Weka system or with plain implementations.
The performance overhead introduced by external Java functions integrated in XQuery is modest. This is confirmed by the graphs of Figure 6.7, that report the overall execution time on the testing datasets as a sum of:

1. the initialization time, i.e. the time to compile the query and to prepare the data structures;

2. the data iteration time, i.e. the time to iterate over the data several times, according to the number of cycles of the Apriori;

3. the mining time, i.e. the time to update data structures at each iteration;

4. the serialization time to produce the output.

The performance of the Apriori worsens when the number of generated patterns is very high - more than 500,000 itemsets with 1.5GByte of memory (see the experiments on the mushroom dataset). Such a behaviour is mainly due to the encoding of the extracted patterns entirely in main memory and to the context-switching overhead due to their serialization.
To conclude, the overall performance also depends strictly on the complexity of the XQuery fragments and an important issue is the optimization of the execution of user-defined XQuery expressions, together or separately. As an example, let us consider the query of Example 5.2. Since path expressions offer in general an efficient way to iterate over items, during the compilation phase we can merge the FOR ITEM and LET ACTIVE clauses over a joint path expression, in the following way:

\[
\text{fn:concat(./FirstName, ' ', ./LastName)}
\]

Experiments showed that such a modification brings an increase to the performance of about 13% in respect to the graphs reported above. Even with its simplicity, this example confirms that a thorough modification of the underlying XML optimizer needs to be investigated in detail. Section 6.1.3 offers a good starting point to this purpose.

### 6.2.2 Application scenario: mining over the xmark database

The experiments earlier presented show promising results, but they have rather limited value due to a couple of reasons: (i) they are limited to the frequent itemsets mining task and (ii) they have rather little to do with real XML data, since datasets are translated from a textual format to XML.
To overcome these problems, we offer here a set of queries where each query is intended to challenge a particular aspect of the data mining, not only related to the extraction of frequent patterns. More important, such queries are designed over a scalable XML database, named \texttt{x-mark}, frequently used by the database community as a real XML benchmark suite.

In short, we propose a concise, yet comprehensive set of queries which covers the major aspects of data mining, ranging from pre-processing, via data mining to post-processing. We complement our research with results we obtained from running the set of queries on several scalable \texttt{xmark} documents.

\section*{Description of the database}

The aim of the XMark project\footnote{http://www.xml-benchmark.org/} is to provide a benchmark suite that allows users and developers to gain insights into the characteristics of their XML repositories \cite{101}.

The structure of the document is modeled after a database as deployed by an Internet auction site. As reported in \cite{101}, the main entities are: \texttt{person}, \texttt{open auction}, \texttt{closed auction}, \texttt{item}, \texttt{category} and \texttt{annotation}. The relationships between such entities are expressed through references depicted in Figure 6.8. The semantics of the main entities used in the experiments is as follows:

1. Items are the objects that are on for sale or that already have been sold. Each item carries a unique identifier and bears properties like payment (credit card, money order, etc.), a reference to the seller, a description, all encoded...
as elements. Each item is assigned a world region represented by the item's parent.

2. Open auctions are auctions in progress. Their properties are the privacy status, the bid history (i.e., increases or decreases over time) along with references to the bidders and the seller, the current bid, the time interval within which bids are accepted, the status of the transaction and a reference to the item being sold, among others.

3. Closed auctions are auctions that are finished. Their properties are the seller (a reference to a person), the buyer (a reference to a person), a reference to the respective item, the price, the number of items sold, the date when the transaction was closed, the type of transaction, and the annotations that were made before, during and after the bidding process.

4. Persons are characterized by name, email address, phone number, mail address, homepage URL, credit card number, profile of their interests, and the (possibly empty) set of open auctions they are interested in and get notifications about.

5. Categories feature a name and a description; they are used to implement a classification scheme of items. A category graph links categories into a network.

The schema is regular on a per entity basis and exceptions, such as that not every person has a homepage, are predictable.

Over the x-mark project, an automatic generator program provides a scalable XML document database. Table 6.3 shows the main characteristics of the four databases used in the experiments. For each row, the table reports the name of the database, the scaling factor used to generate the document, the document size (in Megabytes), the number of XML nodes, the size of the XML tree and, finally, the number of elements in the main entities. Some snippets of the benchmark document are reported in Appendix A.

**Description of the scenario**

This sub-section lists the queries of which the benchmark consists. A brief motivation of their definition is also given.

We are going to suppose that the xmark analyst - in brief the user - is interested to DM techniques over the auctions registered into the database, to the aim of extracting high-level information of the navigational behaviour of the bidders.

As a first request, he/she aims at a mining model capable of classifying the profile of new persons that register to the site or that perform some relevant transaction over it. He/she shows interest into two kinds of experiments. The first one tries to predict whether a person might be interest in visiting auctions belonging to a
certain category (e.g. music, sport, books, etc.). The model extracted from this kind of analysis may serve for on-the-fly banners or page reorganization, thus giving a personalized view of the web site. The second one tries to predict the sex of a person based exclusively on the closed auctions for which he/she has been buyer or seller. Again, the knowledge of the sex allows to target banners, promotions, news, or to give relevance to specific sections of the web site for bidders that do not provide any personal information during the registration stage. In this way, the web site could provide personalization features without forcing registration. In order to accomplish these objectives, we combine some discretization technique with a classification tree extraction.

[D1] Discretize the income (if any) of each person into three distinct bins having an equal width. In the result, append to each <person> tag a new XML element, <income-d>, containing both the original and the discretized value.

```
[prepare discretization doc("people-d") using alg:equal_width() for data $person in doc("x-mark")/site/people/person
let active field $income := xs:double($person/profile/@income)
having bin $income-d as ("low income", "med income", "rich income")
return <person-d>
  ((for $i in $person/* return $i),
   if (empty($income)) then ()
   else <income-d income-num="{$income}"/{$income-d})
} </person-d>
```

[C1] On the basis of the result of the previous query, extract a classification tree able to determine whether a person may be interested or not to a particular category. Attributes of the task are: (i) business information (ii) kind of education (i.e. graduate school, college, high school or other) (iii) a boolean value indicating whether the person has a credit card or not (iv) the discretized value of the income (v) the

Table 6.3: The four x-mark documents used in the experiments.
6.2. PERFORMANCE EVALUATION

gender (vi) a boolean value indicating whether the number of open auctions he/she is interested in is greater than 5. The target is a boolean attribute that indicates whether such a person may be interested to “music” or “film” auctions\textsuperscript{4}.

declare variable $music-cat :=
   for $i in doc("xmark")/site/categories/category
      where $i/description/text contains "music" ftor "film" ftor "dvd" ftor "cd"
         occurs at least 3 times
      return $i;

mine tree doc("interested-tree") using alg:id3()
for data $person in doc("people-d")//person-discr[not(empty(profile/interest))]
let active field $business := $person/profile/business/text()
let active field $education := $person/profile/education/text()
let active field $has-credit-card := string(not(empty($person/creditcard)))
let active field $income := $person/income-d/text()
let active field $gender := $person/profile/gender/text()
let active field $watch := string(count($person/watches/watch) > 5)
let predicted field $has-a-music-interest :=
   string(some $j in $person/profile/interest/@category
      satisfies some $k in $music-cat/@id satisfies $j eq $k)

[D\textsubscript{2}] Discretize the price of each closed auction into twenty distinct bins having an equal width. Return the discretized value, the auction’s seller and the auction’s buyer.

prepare discretization doc("auction-d") using alg:equal-width()
for data $auction in $doc("x-mark")/site/closed-auctions/closed-auction
let active field $price := xs:double($auction/price)
having bin $income-d as for $i in (1 to 20) return fn:concat("price-",$i)
return <auction-discr>
   (if (empty($price)) then () else <price-d>$price-d</price-d>),
      $auction/seller, $auction/buyer)
</auction-discr>

[C\textsubscript{2}] Classify the gender of a person according to the closed auctions he/she has been buyer or seller. The training data is the XML document obtained in the previous query. The active attribute $B_i$ (resp. $S_i$) with $i \in [1..20]$ assumes a boolean value indicating whether the person has been buyer (resp. seller) of some item whose price falls into the interval $i$.

declare variable $auc-g :=
   for $j in (1 to 20)
\textsuperscript{4}This information is obtained by means of the XQuery Full Text, that is the upcoming W3C Recommendation for content-based querying. Specifically, we aim at determine whether some category of interest contains the words “music”, “film”, “dvd” or “cd” in any order but occurring at least three times. For details see \cite{111}.
Then, the user asks for a particular sampling technique, to be used later in future classification purposes. The aim is to sample both open and closed auctions. The parameters of the sampling are specified below.

[S1] Sample both open and closed auctions by using a random sampling without replacement. The number of created samples is equal to the number of world regions. The number of cases in each sample is proportional to the number of items in each region. Label each extracted sample with the region name.

declare variable $regions :=
    ("africa","asia","australia","europe","namerica","samerica");

prepare sampling doc("sampled-auctions") using alg:random-sampling(false())
for data $auction in (doc("x-mark")/site/open_auctions/open_auction,
    doc("x-mark")/site/closed_auctions/closed_auction)
having samples $s in for $i in $input/site/regions/*
    return count($i/item) div count($input/site/regions/*/item)
return <sample region="{for $i at $pos in $s
    return if ($i > 0) then $regions[$pos] else ()}"{ $auction
} </sample>

The next goal is to extract a descriptive model to find frequent correlations among the bidders in all the open auctions. The output model contains patterns
like \{person100, person200\} supp=0.50, which states that, the person person100 and the person person200 appear together as bidders the 50% of the times w.r.t. the open auctions history. Since at this stage the user has not idea on the parameters of the analysis, we suggest him/her to extract the largest number of frequent patterns from the bid history.

**[FI]1** Find frequent correlations among the bidders in all the open auctions. Try to use a minimum support as small as possible.

```xml
mine itemsets doc("co-bidders") using alg:apriori(0.002)
for data $auction in doc("x-mark")/site/open_auctions/open_auction
for item $bidder in $auction/bidder
let active field $name := string($bidder/personref/@person)
```

Suppose now that by inspecting the result of the previous data mining extraction step, the user is interested in investigating all the pairs (seller, buyer) in the closed auctions history that are not violated by the set of extracted patterns. In particular, we suggest him/her to look, for each pair (seller, buyer), at the percentage of itemsets that comply with it. An high value of this variable may indicate that such persons tend also to bid together during the auctions.

**[PP]1** With reference to the itemsets extracted in the previous query, return as output the set of triplets, \((b, s, c)\), where \(b\) (resp. \(s\)) is the buyer (resp. seller) identifier in all the closed auctions, and \(c\) is the percentage of the itemsets that comply with \(b\) and \(s\), over the total number of itemsets.

```xml
apply itemsets doc("supp-seller-buyer")
for association model $m in doc("co-bidders")
let metadata field $total := xs:decimal($m/PMML//@numberOfItemsets)
for data $closed in doc("x-mark")/site/closed_auctions/closed_auction
for item $sel-buy in $closed/(seller|buyer)/@person
let active field $item := string($sel-buy)
having supported $s,
return <closed_auction num-itemsets="$total">
    ($i/buyer, $i/seller, <supp-perc>($s div $total) * 100</supp-perc>)
</closed_auction>
```

The query [FI]1 above generates a very large number of itemsets that are hardly human readable. In order to reduce the number of generated patterns, we suggest the user to use some constraint expressed on the patterns themselves. Specifically, he/she is interested on three kinds of queries. In the first one, he/she decides to change the input data also including auctions that have been closed with a certain happiness. Also, he/she asks to filter out all the frequent 1-itemsets. In the second query he/she requires that at least one person in each itemset bought at least one item. Finally, in the third query below he/she aims at extracting only the patters
including at most two business men whose income is greater than the mean of the people’s incomes. In the latter one, a data filtering constraint also requires to compute only the transactions whose bids fall into a certain time interval.

\[FI_2\] Find frequent correlations among the bidders in open auctions and including also the seller and the buyer in each closed auction. The happiness of each transaction must be greater than 2. Also look only for itemsets of length at least 2.

\[
\text{mine itemsets } \text{doc("co-bidders-2") using alg:apriori(0.001)
for data } \text{$\text{auction$ in (doc("x-mark")/site/open\_auctions/open\_auction,
\text{doc("x-mark")/site/closed\_auctions/closed\_auction)
where } \text{$\text{auction$\text{/annotation\_happiness\text{/text()} > 2
for item } \text{$\text{bid$ in if ($\text{$i$\text{/name()} eq "open\_auction") then $\text{$i$\text{/bidder\_personref\_person
else $\text{$i$\text{/\text{seller|buyer\_person
let active field } \text{$\text{name$ := string($\text{bid$)
having at least 2 item satisfies true()}
\]

\[FI_3\] Find frequent correlations among the bidders in all the open auctions. For each pattern, at least one person bought some item.

\[
\text{mine itemsets } \text{doc("co-bidders-3") using alg:apriori(0.001)
for data } \text{$\text{auction$ in doc("x-mark")/site/open\_auctions/open\_auction
for item } \text{$\text{bidder$ in $\text{auction$\text{/bidder\_personref\_person
let active field } \text{$\text{name$ := string($\text{bidder$)
let metadata field } \text{$\text{has-one-item$ := (some $\text{$i$ in doc("x-mark")/site/closed\_auctions/closed\_auction
satisfies $\text{$i$\text{/buyer\_person eq $\text{name$)
having at least 1 item satisfies $\text{has-one-item$}
\]

\[FI_4\] Find frequent correlations among the bidders in all the open auctions. Each auction must fall into a time interval of 50 days (i.e. the difference among the first bid and the last bid is less or equal than 50 days). Each pattern includes at most two business men whose income is greater than the mean of the people’s incomes.

\[
\text{declare variable$ \text{$\text{mean$ := fn:sum(doc("x-mark")/site/people/person/profile/@income) div count(doc("x-mark")/site/people/person);

declare function local:days-range($\text{auction$)
{ let $\text{first$ := $\text{auction$\text{/bidder[1]/date\_text()
let $\text{last$ := $\text{auction$\text{/bidder[last()]\_date\_text()
return if (empty($\text{first$) or empty($\text{last$)) then 0
else let $\text{d1$ := xs:date(local:format-date($\text{first$))
let $\text{d2$ := xs:date(local:format-date($\text{last$))
return fn:days\-from\-duration($\text{d2$ - $\text{d1$})
};
\]
Finally, the user aims at extracting a semantically different kind of patterns. He/she is interested in frequent correlations among all the open auctions that young people are interested in and get notifications about. For example, the pattern \{auction100, auction200, auction300\} \(\text{supp}=0.03\) means that, for the 3\% of the times, the three auctions were watched together by the young people. Such an information may indicate that persons that frequently tend to visit some open auction also tend to visit other kinds of auctions. The user also constrain the output selecting all the patterns in which a certain number of auctions have a difference among the current price and the initial price above/below certain thresholds.

\[\text{FI}_3\] Find correlations expressing how frequently open auctions are watched together by the under-30 people. In each pattern, at least 5 items increase their initial price of more than 30\% w.r.t. the current price, but at most 10 items increase their initial price of less than 10\%.

Experiments and discussion

The queries scenario above has been designed by taking into account our real experience on data mining. On the one side, the core of the examples are the Apriori and ID3 algorithms, that provide a significant starting point on how an inductive database system has to be designed. On the other side, such scenario highlights the need of combining different tasks, such as classification with discretization (queries \([C_1]\) and \([C_2]\)), or to apply models to further data (query \([PP_1]\)). More important, the queries defined over the frequent itemsets mining task (in particular queries \([FI_3]\), \([FI_4]\) and \([FI_5]\)) show how more complex reasoning schemes can be formalized by combining context knowledge with mining. Finally, preprocessing and post-processing
queries highlights a feature inherited by the XQuery world, that is the capability of constructing and personalizing both the query result and the parameters of the analysis.

The experiments reported here are gathered from our work with XQuake. As mentioned before, since no similar systems exist in both the academic and industrial XML world (to the best of our knowledge), a comparison among XQuake with similar choices is very hard to achieve. So, the time evaluation is only meant as an impression on the performance of the XQuake system.

The time were measured on a dual-core Athlon 4000+ running Windows Vista. We assigned the maximum memory possible to the Java Virtual Machine: 1.6Gbyte of memory. The benchmark documents are scaled at factors 0.1, 1, 5 and 10, as shown in Table 6.3. Again, the execution time includes the time to serialize the XML output into a file, but excludes the time needed to store such source into BaseX. Bulk load and full indexing for the largest dataset - more than 1GB - required about 140 seconds in BaseX. Also, for the hardest mining queries (more precisely [FI₃], [FI₄], [FI₅] and [C₁]), we used the main-memory option available in BaseX⁵, to the aim of reducing the time for evaluating joins. Table 6.4 gives some detail about the parameters used in the queries from [FI₁] to [FI₅]. Specifically, the table reports, for each dataset, the minimum support threshold used (both relative and absolute), the number of transactions, the number of items and, finally, the number of generated patterns.

The running time (in seconds) is displayed in Table 6.5, while Figure 6.9 (left) depicts the scalability for six benchmark queries. As one could observe, queries [FI₃], [FI₄], [FI₅] and [C₁] require some expensive join in order to select data for the mining. Such joins affect the overall execution time. Consider for example the query [FI₅]. It requires a multi-join among some <watch> element of each person with the <open_auction> entity, in order to determine whether such a person is interested in or get notifications about that auction. Over such a query, XQuake spends about 83% of the total processing time in iterating over the data, and in evaluating the mining constraint (see Section 5.2.3). This time dominates the time of the Apriori algorithm to generate the required patterns. A similar consideration can be applied to the query [C₁], in which more than the 90% of the time is spent in extracting the values of the classification field from the original data, while the remaining time is used to build the tree on such values. The graph of Figure 6.9 (right) shows that the time difference between main-memory and on-disk execution may be considerable whenever the query contains some expensive join. On the contrary, queries [FI₁] and [FI₂] highlight a different situation, in which the constraint evaluation is now trivial and only the mining engine of XQuake is stressed. In these cases, the mining time dominates the data preparation time.

⁵In the main-memory mode, both the table data and the text are kept in memory. This option will not create any file on the disk. It can be activated by means of the command “set onthefly on” available on the XQuake GUI.
6.2. PERFORMANCE EVALUATION

Table 6.4: Parameters used for the frequent itemsets extraction queries.

<table>
<thead>
<tr>
<th>Query</th>
<th>DB</th>
<th>Sup %</th>
<th>Sup abs</th>
<th># Trans</th>
<th># Items</th>
<th># Freq. itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>FI_1</td>
<td>Tiny</td>
<td>2,0E-3</td>
<td>2</td>
<td>1.200</td>
<td>2.337</td>
<td>1.937</td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>5,0E-4</td>
<td>6</td>
<td>12.000</td>
<td>21.257</td>
<td>1.593</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>2,3E-4</td>
<td>14</td>
<td>60.000</td>
<td>69.228</td>
<td>67.573</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>1,5E-4</td>
<td>18</td>
<td>120.000</td>
<td>110.640</td>
<td>3.453</td>
</tr>
<tr>
<td>FI_2</td>
<td>Tiny</td>
<td>1,0E-3</td>
<td>2</td>
<td>2.175</td>
<td>2.298</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>3,5E-4</td>
<td>8</td>
<td>21.750</td>
<td>21.338</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>1,5E-4</td>
<td>16</td>
<td>108.750</td>
<td>72.295</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>1,1E-4</td>
<td>24</td>
<td>217.500</td>
<td>116.851</td>
<td>283</td>
</tr>
<tr>
<td>FI_3</td>
<td>Tiny</td>
<td>5,0E-3</td>
<td>6</td>
<td>1.200</td>
<td>2337</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>5,0E-4</td>
<td>6</td>
<td>12.000</td>
<td>21.257</td>
<td>355</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>2,6E-4</td>
<td>16</td>
<td>60.000</td>
<td>69.228</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>1,6E-4</td>
<td>19</td>
<td>120.000</td>
<td>110.640</td>
<td>283</td>
</tr>
<tr>
<td>FI_4</td>
<td>Tiny</td>
<td>1,0E-2</td>
<td>4</td>
<td>410</td>
<td>1.750</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>1,0E-3</td>
<td>4</td>
<td>4.219</td>
<td>14.269</td>
<td>1.875</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>5,0E-4</td>
<td>11</td>
<td>21.095</td>
<td>39.186</td>
<td>1.490</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>4,0E-4</td>
<td>17</td>
<td>41.880</td>
<td>59.296</td>
<td>1.313</td>
</tr>
<tr>
<td>FI_5</td>
<td>Tiny</td>
<td>5,0E-3</td>
<td>2</td>
<td>333</td>
<td>456</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Small</td>
<td>5,0E-4</td>
<td>2</td>
<td>3.415</td>
<td>5.040</td>
<td>4.399</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>2,0E-4</td>
<td>3</td>
<td>16.900</td>
<td>23.914</td>
<td>11.7811</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>1,5E-4</td>
<td>5</td>
<td>38.899</td>
<td>30.733</td>
<td>6.605</td>
</tr>
</tbody>
</table>

Table 6.5: Performance impression of XQuake (time is in seconds). †Query has been executed main memory. *Query has been executed with full-text indexes.

Summing up, the performance of a mining query clearly depends both on the performance of the XQuery run-time support, and on the capability of the mining system to couple with the database. With a certain degree of generality, the total execution time $T_{tot}$ is given as a sum of the iteration time $T_i$ (i.e. the time to iterate over the data in order to prepare data structures), the mining time $T_m$ (i.e. the time to extract the model from such data structures) and a certain overhead $T_{over}$ due to the time spent in coupling the mining with the DBMS. The more such a coupling
is well defined, the more the overhead time is mitigated. While maintaining the
schema of the algorithm in XQuery, the XQuake engine uses external Java functions
to interact with the native XML database. The encoding of the evaluation of XQuery
expressions into more efficient Java structures is part of our $T_{over}$. These experiments
show that this time is acceptable, since it is always dominated by $T_i$ and/or $T_m$.
However, other interesting ways of coping with efficiency need to be investigated
as well (see Section 6.1.3). We expect our work to continue and evolve also in this
direction in the near future.
7.1 Final remarks

There are several reasons that justify the study of a powerful, expressive and efficient XML-based framework for intelligent data analysis. First of all, the proliferation of XML sources offer good opportunities to mine new data. Second, native XML databases seem a natural alternative to relational databases when the aim is of querying both data and the extracted models in an uniform manner.

In this thesis we proposed a query language (and system) as a solution to the XML data mining problem. In our view, an XML native database is used as a storage for KDD entities. Data mining tasks are expressed as a super-set of XQuery and the mining language is designed over a simple extension of the XQuery Data Model. In surveying the state-of-the-art of systems for knowledge discovery in databases, we briefly discussed some of the issues related to the completeness of the process, the expressiveness of the formalism in representing queries and KDD entities, the flexibility and the capability of the system architecture to provide efficient implementations. We took into consideration these problems during our study.

In short, the XQuake language presented has the following features:

- It offers a certain degree of versatility, to specify a variety of different mining task over different XML sources.

- It has an intuitive syntax, inspired by the XQuery world.

- It has a grammar that accepts a certain degree of extensibility in order to introduce user-defined functions in the statements.

- It allows us to turn back the model into the data it was mined from and to pre-process and select the subsets of data to be mined as well.

- It offers facilities for (partially) constructing and customizing the output results as well as the user preferences.
• It verifies the principle of closure.

In particular, the last property is fundamental in inductive databases. It simple
states that any operator returning type $t$ can be used wherever an argument of type
$t$ is required. These features verify all the constraints a query language for KDD
has to verify.

Another important aspect is the ability of directly defining and querying the
prior knowledge to be used within the mining query. The user first expresses how to
locate it as an XML piece of data, then he/she defines the constraints by means of
the querying mechanisms of the language. More important, he/she does not have to
consider the details of the mining algorithm, nor the properties that such constraints
have to verify. These properties are used by the XQuake system to push constraints
as deeper as possible inside the mining process, and this integration is automatic.
The Apriori implementation offers a good example of such an integration.

In order to satisfy all these features, we proposed a tightly-coupled implementa-
tion capable of providing both a direct specification of mining algorithms in XQuery
and the use of separate functions for the efficient implementation of critical aspects.
Despite its recent standardization, arguably, XQuery is the most promising pro-
gramming language for this purpose and this kind of solution has the advantage
of avoiding “black box” implementations of mining algorithms. Some adaptations
to the XQuery language have been presented in order to allow the specification of
the general schema of preprocessing and mining algorithms. The first empirical
assessment reported here exhibits promising results.

7.2 Future research directions

Clearly, there are some issues or further extensions that need an in-depth investiga-
tion. We can devise two orthogonal and often contrasting research lines.

7.2.1 System architecture optimizations

In the previous chapter we have already shown some example of how our approach
could benefit of thorough modifications of the underline system architecture. These
extensions concerns all the levels of the system.

For example, future activities on the GUI level may involve the design of more
user-friendly GUIs built on top of the system. As mentioned, supporting workflows
for the mining process is a feature that is already part of many KDD environments
such as [32, 40, 45, 81, 82, 91, 102, 89, 106]. Of a certain interest is the design of
a compiler from visual descriptions of KDD tasks and flows into an XQuake query.
In this way, the users should have a powerful graphical abstraction that simplifies
the design, without limiting the expressiveness.

The level below, i.e. the compiler level, may be integrated with several opti-
mizations techniques, such as query rewriting [31, 37, 72] or XQuery-specific opti-
mizations. Also the database access can be made more efficient by defining ad-hoc indexing techniques.

Moreover, the problem of introducing facilities for XML updating must be investigated with a great attention. The three mining algorithms presented in this dissertation make use of external methods to compute complex operations over data structures. Clearly, this is a good opportunity in several different cases - e.g. this is confirmed by the performance evaluations shown in the earlier chapter. However, in some other case, such a strategy may reveal inefficient, in particular when the input/output encoding of external methods brings to an high performance overhead. An on-going work is the direct exploitation of update, insert and delete operators directly into the iterate statement of the wfor iterator, in such a way that its semantic is respected. Their definition better justifies the adoption of wfor.

Finally, another important question pertains to the compact tightly-coupled architecture used in XQuake. Since there already exist several open-software KDD systems, may be very interesting to leverage on their success and re-design XQuake as DM middleware loosely coupled with those systems. From an architectural point of view, this solution is possible with a certain cost. For example, the Apriori algorithm of Weka [106] may be integrated in our system by providing the implementation of an “input wrapper” that transforms the data to be mined (i.e. resulting by evaluating the XQuery expressions encoded in XQuake) into instances in the Weka format. Similarly, also the implementation of an “output wrapper” from the set of mined patterns to the PMML format must be provided. The price to pay is a certain time overhead due both to the execution of such wrappers and to the evaluation of mining constraints only at post-processing time instead of during the extraction process, as the tightly-coupled architecture of XQuake permits.

7.2.2 Expressiveness on the prior knowledge

The solution presented in this thesis adopts an XML format to represent context knowledge. An interesting future work includes the exploitation of ontologies. In fact, ontology-driven data mining is an emerging research field having a wide area of applications.

We can perceive the relation between ontologies and data mining in two manners.

- Incorporating knowledge in the process through the use of ontologies, e.g. as intelligent assistants for the discovering process, for interpretation of the results or for validation of mined knowledge.
- Using domain knowledge in the input information or using the ontologies to better represent the mining results.

Clearly, the second manner fits well with our approach, since there is a strong interaction of the system with the data miner expert that defines the ontologies
expressing the information about an application domain as well as the queries to drive the extraction of knowledge into more accurate output. In other terms, the mining analysis is done with support of these ontologies.

An example is reported in [61] in which the authors present an approach to extract patterns from databases containing historical data about earthquakes events and they use ontologies as scientific seismological prior knowledge, in order to improve querying over the patterns. The goal is to replay to questions such as “find similarities in shock sequences happened in Greece during 2004”. Such a query requires the incorporation of more advanced domain knowledge by the expert, i.e. the specification of the “similarity measure” and the definition of the “shock sequence”. To represent the domain of interest, the authors suggest a merge among seismological ontologies and geographical information. Clearly, infinite similar examples exist, such as using ontologies as “enriched taxonomies” to query product hierarchies in a market basket analysis or to guide the construction of a classification tree at multiple levels of abstraction.

Summing up, using an ontology could result in a more focused mining, since it may provide enhanced possibilities to constrain the mining queries in a more expressive way, by exploiting object and data properties. Noticeably, all these examples fit well with our approach, but integrating ontologies in our work is not an easy task and a lot of issues have to be addressed. First of all, the use of ontologies in the data mining process requires choosing a specific ontology language. Languages which have gained high popularity in the last few years are those designed for the semantic web: RDF [109] and OWL [108]. The use of such languages is even more substantial in our project, since they are de facto serialized into XML documents.

On the other hand to get best results a powerful mechanism of querying is necessary. In our context, we address our attention towards XQuery-oriented solutions\(^1\). Since its standardization, extensions of XQuery for the querying of XML combined with OWL and RDF(S) reasoning/querying have been proposed (e.g. [66, 4, 5]). All these approaches share two main common points:

- XQuery and XPath are extended for the traversal of OWL statements;
- the extended XQuery can combine the querying of XML documents together with the reasoning with RDF/OWL.

It is easy to notice that these approaches are attractive, since an uniform mechanism to query data, mining models and prior knowledge represented via ontologies is desirable.

\(^1\)An alternative approach is to use SQL-like languages such as those derived from the SPARQL family. However, with this kind of solution the user must deal with two different (style of) languages: SQL and XQuery. Despite the difficulty to make the two languages coexist, this strategy violates the principle that the data mining is an uniform querying process.
Beside these enhancements, planned activities include the integration of other preprocessing, mining and post-processing operators, to stress XQuake with different classes of applications. As soon as the kernel of the system is robust enough, application-oriented knowledge discovery could pilot XQuake towards a completely general-purpose solution for data mining. Two recent works presented by our research group at the PKDD conference in September 2008 [46] and at the CONTEXT conference in August 2007 [8] may represent two good starting points.

The concepts explained in this manuscript and the work on XQuake will be published in [95] and [94].
Appendix A

XML datasets

This appendix reports some sample of XML dataset as used throughout this thesis. Specifically, Figure A.1 shows a fragment of the online dblp database\(^1\) containing bibliographic information on major computer science journals and proceedings. Figure A.2 depicts a sample document including various information about researchers in a university department. We can suppose to have a distinct XML document for each department of interest, with the overall set of departments information stored into a document collection. The XML document mondial\(^2\) shows in Figure A.3 contains a collection of XML tags each of them storing geographical, economic and political characteristic of a country as well as information on mondial organizations. Finally, Figure A.4 and A.5 report some snippets of the xmark document.

\(^1\)http://dblp.uni-trier.de/xml/.
\(^2\)http://www.dbis.informatik.uni-goettingen.de/Mondial/.
Figure A.1: XML fragments of the `dblp` dataset.
Figure A.2: XML fragments of a department dataset.
Figure A.3: XML fragments of the mondial dataset.
Figure A.4: A sample of the x-mark dataset: items and persons.
Bibliography


[82] ORANGE DATA MINING FRUITFULL&FUN. magix.fri.uni-lj.si/orange/.


[89] Rakotomalala, R. TANAGRA. chirobble.univ-lyon2.fr/~ricco/tanagra/.


