Università di Pisa
Dipartimento di Informatica

Technical Report: TR-09-07

XML Data Mining

Andrea Romei Franco Turini

April 6, 2009

ADDRESS: Largo B. Pontecorvo 3, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

XML Data Mining

Andrea Romei Franco Turini

April 6, 2009

Abstract

XML is the standard language for representing semi-structured data. With the spreading of XML sources, mining XML data can be an important objective in the near future. This paper presents a project focussed on designing a general-purpose query language in support of mining XML data. In our framework, raw data, mining models and domain knowledge are represented by way of XML documents and stored inside XML native databases. Data mining tasks are expressed in an extension of XQuery. Special attention is given to the frequent pattern discovery problem, and a way of exploiting domain-dependent optimizations and efficient data structures as deeper as possible in the extraction process is presented. We report the results of a first bunch of experiments, showing that a good trade-off between expressiveness and efficiency in XML data mining is not a chimera.

1 Introduction

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). KDD and DM technologies have reached a maturity state as far as the design of efficient knowledge extraction algorithms is concerned. This is witnessed by the large number of commercial tools and RDBMS offering KDD algorithms ([3, 29, 18], only to cite a few ones). On the contrary, the design of applications is still an "art", aimed at smoothly composing algorithm libraries, proprietary APIs, SQL queries and stored procedure calls to RDBMS, and "much much code". This is the main motivation that inspired Imielinski and Mannila to launch the idea of *Inductive Databases* (IDBs), general purpose databases in which both the data and the knowledge are represented, retrieved, and manipulated together or separately [20]. IDBs should help the analyst in the hard task of producing a KDD application as a mixture of data pre-processing, data mining and post-processing steps.

A critical aspect 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 on them. Because data typically reside in relational databases, a considerable

number of different efforts propose to integrate a mining tool in a relational DBMS [3, 9, 13, 21, 26, 35]. According to the closure principle of IDBs, both the source data and the induced models are represented as database relations, while queries are specified in an SQL-like language.

However, the simplicity of the relational data model does not provide an easy way to manage different models, as well as complex domain knowledge. Furthermore, the database support need not be necessarily provided by a relational one.

Why mining XML data? The past few years have seen a growth in the adoption of the *eXtensible Markup Language* (XML), thanks to its ability to provide a standardized, extensible mean of including semantics information within semi-structured data. The flexible nature of XML makes it an ideal basis for defining arbitrary languages. One such example is the Predictive Modelling Markup Language (PMML) [30], an industry standard for the representation of mined models as XML documents.

This feature allows to represent raw data and patterns in an uniform and flexible way. In addition, the use of XML permits the description of complex formats, such as trees or clusters, as well as the domain knowledge. From this perspective, ontologies in DM 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.

Moreover, the increasing adoption of XML has also raised new challenges. 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 cumbersome and more optimal compared to the relational format. As an example, geographical and spatial data are becoming increasingly important, due to the pervasiveness of location-acquisition technologies such as GPS, GSM networks, etc.

How mining XML data? 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* [33], which is drawing much research and for which a large number of implementations already exists. From the XML data warehousing point of view, native XML databases are designed for seamless storage, retrieval and manipulating of XML data.

The goal of our research effort is the design and implementation of a mining language and system where an XML native 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 our project 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, like as instance indexes; (v) a coherent formalism capable of dealing uniformly with raw data, induced knowledge and background knowledge.

- We propose an approach 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. 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. As we will show later, some adaptations over the XQuery language are required in order to allow the specification of the general schema of preprocessing and mining algorithms. However, the above strategy allows both the exploitation of domain knowledge within the algorithm specification, for domain-dependent optimizations, and the deferment of the implementation of critical points to user-defined functions, for domain-independent optimizations.

- As a case study, the paper presents how such a technique can be used to implement the Apriori [2] algorithm. Despite its datedness, the algorithm is still a significant example of how the background knowledge can be encapsulated as deeply as possible into the extraction process, to help in improving the performance of the mining. In this respect, particular attention is given to the evaluation of the capability of our framework to accept optimizations proposed in the past literature. At the same time, the performance should be comparable to the performance of similar works.

Paper organization. Section 2 presents the proposed language by means of several examples that illustrate its expressive power and versatility. Section 3 discusses details of the implementation and of the system architecture. Section 4 provides an overview of related work. Section 5 compares the performance of the Apriori with an existing solution based on a different approach. Finally, Section 6 contains the final remarks and suggests avenues for future works. In the remainder, it is assumed that the reader is familiar whit the fundamentals of data mining, XML and XQuery.

2 XQuake

In this section, the XQuake (XQUery-based Applications for Knowledge Extraction) mining language is presented. Due to space restrictions, we present the basic idea behind the language and some examples of the query language at work.

All examples assume the availability of XML data in a native XML database. Specifically, Figure 1a shows a fragment of the online `dblp` database[1] containing bibliographic information on major computer science journals and proceedings. Figure 1b depicts a sample document including various information about researchers in a university department. Finally, the XML document `mondial`[2] of Figure 1c contains a collection of XML tags each of them storing geographical, economic and political characteristic of a country.

2.1 The XQuake philosophy

XQuake adopts an XQuery-like syntax to facilitate high level DM and in-deep integration within XML native databases. Essentially, XQuery expressions are used to identify XML sources, to express constraints on the domain knowledge, to specify interesting measures and the format of the XML output. We consciously chose the most basic KDD techniques 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* [2] and the *ID3* [25] (Chapter 3) algorithms for frequent itemsets mining and decision tree classification, respectively. Moreover, extracted models can be applied to "new" instances to predict features or to select data accordingly to the knowledge stored in the model.

A mining query begins with a collection of XQuery functions and variables declarations followed by an XQuake operator. The syntax of each operator includes five basic statements that specify of five major activities in DM: (i) task and method specification, (ii) domain entities identification, (iii) data filtering, (iv) constraints on domain knowledge and interesting measures specification and, finally, (v) output construction. The outline of a generic operator is explained below. The reader is referred to Appendix A for the syntax of the operators presented in this paper.

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 and <*apply*> is used for modeling application the input data. Several other kinds of constructs are possible to deal with knowledge filtering, model evaluation or model meta-reasoning. As an instance, the following XQuake fragment denotes a data sampling task:

```
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

[1] `http://dblp.uni-trier.de/xml/`.
[2] `http://www.dbis.informatik.uni-goettingen.de/Mondial/`.

```
- <dblp>
  - <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
    - <author dep="id392">
        <first-name>Tomasz</first-name>
        <last-name>Imielinski</last-name>
      </author>
    - <author dep="id456">
        <first-name>Heikki</first-name>
        <last-name>Mannila</last-name>
      </author>
      <title>A database perspective on knowledge discovery</title>
      <journal>Comm. of the ACM</journal>
      <volume>39(11)</volume>
      <pages>58-64</pages>
      <year>1996</year>
      <cite label="BOU00"/>
      ...
      <cite label="CEU02"/>
    - <keywords>
        <keyword>data mining</keyword>
        <keyword>databases</keyword>
        <keyword>sql</keyword>
      </keywords>
      <abstract>The concept of data mining ...</abstract>
    </article>
  - <inproceedings mdate="2002-07-19" key="conf/ic/NayakWT02">
      <author dep="id949">...</author>
      ...
    </inproceedings>
    ...
  </dblp>
```

(a)

```
- <Department id="dep145">
    <Name>Dep. of Computer Science</Name>
    <University> ... </University>
    <Address> ... </Address>
  - <People>
    - <Employee>
      - <PersonalInfo>
          <Name> ... </Name>
        </PersonalInfo>
      - <Education>
        - <Higher>
            <MsC year="1996"> yes </MsC>
            <PhD year="1999"> yes </PhD>
          </Higher>
        </Education>
      - <Awards>
        - <Award year=" 2001" dociety="IEEE">
            <Description>This award ...</Description>
          </Award>
        </Awards>
      - <Projects>
        - <Project active="yes" isDirector="yes" name="p1">
            <Description> The project ... </Description>
          </Project>
        - <Project active="no" name="p2">
            <Description> The project ... </Description>
          </Project>
        </Projects>
      </Employee>
      ...
    </People>
  </Department>
```

(b)

```
- <mondial>
  - <country car_code="I" area="301230" capital="cty-Italy-Rome" memberships="org-EU org-FAO org-G-8 org-NATO ...">
      <name>Italy</name>
      <population>57460274</population>
      <population_growth>0.13</population_growth>
      <infant_mortality>6.9</infant_mortality>
      <gdp_total>1088600</gdp_total>
      <gdp_agri>2.9</gdp_agri>
      <gdp_ind>31.6</gdp_ind>
      <gdp_serv>65.5</gdp_serv>
      <inflation>5.4</inflation>
      <government>republic</government>
      <encompassed continent="europe" percentage="100"/>
      <religions percentage="98">Roman Catholic</religions>
    - <province id="prov-cid-cia-Italy-68" capital="cty-Italy-Rome" country="I">
        <name>Lazio</name>
        <area>17203</area>
        <population>5185000</population>
      - <city id="cty-Italy-Rome" is_country_cap="yes" is_state_cap="yes" country="I" province="prov-cid-cia-Italy-68">
          <name>Rome</name>
          <longitude>12.6</longitude>
          <latitude>41.8</latitude>
          <population year="90">2791354</population>
        </city>
        ...
      </province>
      ...
    </country>
    ...
  </mondial>
```

(c)

Figure 1: XML fragments of the `dblp` dataset (a), the `departments` dataset (b) and the `mondial` dataset (c).

atomic parameters such as the minimum support for association rule mining algorithms or the confidence for pruning in classification tasks.

Domain entities identification. Any KDD task may need to specify the set of relevant entities as input of 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 FLOWR syntax, in which the result of the evaluation of an expression is linked to a variable in `for` and `let` clauses. For instance, here is a simple fragment statement that can be used to locate both transactions in an association rule mining task or training instances in a clustering activity (we use a `different font` to indicate an XQuery expression).

```
for data $my-tuple in < XQuery expression >
let active field $my-field := < XQuery expression on $my-tuple >
```

Input data is typically a sequence of XML nodes. The <*for*> expression above binds the variable `$my-tuple` to each item during the evaluation of the operator, while the <*let*> clause identifies an attribute of the data. From the DM point of view, the keyword after the <*let*> refers to the role of such attribute in the mining activity of interest. More specifically:

- <*active*> specifies that the field is used as input of the analysis;
- <*predicted*> specifies that it is a prediction attribute;
- <*supplementary*> states that it holds additional descriptive information;
- <*group*> is used to group attributes;
- <*metadata*> defines a special field, used to bind domain knowledge information to an existent single-value field.

Since we refer here to a classical DM approach, mining fields in input to the mining task are typically required to be atomic, i.e. an instance of one of the built-in data types defined by XML schemas, such as string, integer, decimals and date. A richer set of types may be available by extending the data types of XQuery, for example considering ordering or cyclical data types. Moreover, XQuake respects the typing philosophy of XQuery by offering a method to equip attributes with logical information. In the example below, an explicit type is specified by the user for the field `$has-kdd-keyword`.

```
for data $paper in doc("dblp")//article
let active field $has-kdd-keyword as xs:boolean :=
   some $keyword in $paper/keywords/keyword satisfies $keyword eq 'KDD'
```

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. As 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 that expected type. The grammar also allows us to specify other kinds of qualifiers for an attribute, for example in order to assign an weight before the construction of a model.

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 algorithm will take care of handling such missing values according to the targets of the analysis. In the next query fragment below, we specify a numeric attribute `$gdp` that admits missing values if the homonym XML element is absent.

```
for data $country in doc("mondial")//country
let active field $gdp as xs:double?  :=
      if empty($country/gdp) then () else $country/gdp cast as xs:double
```

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

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

From the mining models perspective, a similar syntax may be used to locate (parts of) a (new or extracted) PMML model.

```
let tree $model := doc("my-tree-model")
let supplementary field $ms := pmml:get-mining-schema($model)
```

In the example above, the construct automatically binds - by means of the `get-mining-schema` built-in function - the mining schema XML element of `my-tree-model` to the variable `$ms`. The type of the variable and the location of such an element are implicit, since XQuake fixes the input and output knowledge to be a PMML model. This seems an appropriate choice for two main reasons. First, it becomes very difficult to encode a model as the output of a mining query, since, typically, its nested structure is complex and hard to specify. Second, PMML is becoming a primary standard for actual models representation as XML documents, adopted by major commercial suites.

Data filtering. For operators accepting as input sources data, XQuake 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.

Constraints on domain knowledge and interest measures. 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 key idea of XQuake is the integration of XQuery expressions inside an ad-hoc custom statement that depends on the kind of target and on the kind of knowledge to be mined. For example, for frequent itemsets, it allows the specification of domain-dependent constraints on the format of the input itemsets (see also Section 2.3).

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 <*return*> clause 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.

2.2 XML data preprocessing

Data preprocessing [27] is a time-consuming phase of the KDD process, including tasks such as data selection, filtering, merging, cleaning, aggregating and many others. At the time of writing, XQuake offers two operators for data discretization and sampling.

Data discretization. As a first example, we consider the problem of unsupervised discretization that has the objective of producing k intervals of a numeric attribute.

Example 2.1. In the `mondial` dataset, the tags `population` and `inflation` have continuous values. In order to exploit classification tasks, we need a preprocessing phase in which a mapping of the continuous values to discrete intervals is provided. The problem can be solved by means of the <*prepare discretization*> operator.

```
prepare discretization doc("europe-discr") using alg:natural-binning()
for data $country in doc("mondial")/mondial/country
where some $j in $country/encompassed_continent satisfies $j eq 'europe'
let active field $inf := xs:double($country/inflation)
let active field $pop := xs:double($country/population)
having bin $inf-d in ('low', 'med', 'high'),
       bin $pop-d in discr:format-bins(100)
return <country> {
  for $i in $country/*
  return if ($i/fn:name() eq 'population')
         then <population @discr-as="{$pop-d}"> {$pop} </population>
         else if ($i/fn:name() eq 'inflation')
              then <inflation @discr-as="{$inf-d}"> {$inf} </inflation>
```

```
else $i} </country>
```

The algorithm used here is the natural binning discretization. The <*for*> clause selects the XML nodes that constitute the data set, i.e. the set of countries. For each node, a nonempty set of <*let active*> clauses denotes - again through XQuery expressions - the fields that will be considered for the discretization, i.e. the ones that are active in this mining task. The `$pop` and `$inf` variables are implicitly labelled as numeric types, and missing tags for the <`population`> or <`inflation`> XML elements are directly treated as unknown values by the discretization algorithm.

The user preferences are specified in the <*having*> construct through a nonempty set of <*bin*> clauses. Specifically, we aim at computing three distinct bins for the inflation attribute and 100 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. Labels may be also expressed by means of complex user-defined functions, since the body of the <*bin*> clauses accepts any XQuery expression whose returning type is `string+`. Each clause automatically binds the result of the discretization to a variable (respectively `inf-d` and `pop-d` in the example) whose expected type is `string?`, i.e. missing input field values will be labelled with missing discretized values. Notice that the scope of such variables is the <*return*> clause, that is evaluated for each country that satisfies the condition expressed in the <*where*> clause to produce a collection of XML nodes. Specifically, the latter one eliminates all the non European countries from the output.

An example of fragment of the output produced by the above operator is the following.

```
<country>
     <name>Greece</name>
     <population @discr-as="[10000000, 20000000]">10538594</population>
     <inflation @discr-as="high">8.1</inflation>
     ...
</country>
<country>
     <name>Italy</name>
     <population @discr-as="[50000000, 60000000]">57460274</population>
     <inflation @discr-as="med">5.4</inflation>
     ...
</country>
```

Example 2.2. We aim at discretizing the measurement of population per unit area of the top ten provinces in a country, ordered according to the number of cities, into two distinct bins.

prepare discretization `doc("human-density")` **using** `alg:equal-frequency()`
for data `$country` **in** `doc("mondial")/mondial/country`

```
where count($country/province) >= 10
let group field $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]
having bin $r-d in ('low-density', 'high-density')
return <r @n="{$country/name}"> {for $i in $r-d return <d>{$i}</d>} </r>
```

In contrast to the Example 2.1, the usage of the <*group*> keyword in the <*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 <*where*> clause ensures that the countries having less than ten regions are eliminated from the discretization process.

Data sampling. Sampling is another widely used task: for an input table, a subset of rows is selected according to a strategy.

Example 2.3. Below an XQuake query selecting two different samples from all the proceedings papers of the `dblp` database by using a random strategy is presented. The samples have respectively 50% and 25% cases over the total number of proceedings.

```
prepare sampling doc("s-proceedings") using alg:random-sampling(false())
for data $paper in doc("dblp")//Inproceedings
having samples $s in (0.50, 0.25)
return let $t1 := if ($s[1]=1) then <train> {$paper} </train> else ()
       let $t2 := if ($s[2]=1) then <test> {$paper} </test> else ()
       return ($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`, the indexes of the samples in which the current paper has been included, according to the sampling strategy. More formally, let $(v_1, \ldots, v_n)$ be the input XML nodes for which a sampling is required. Let $(p_1, \ldots, p_m)$ be the percentages of the samples with $m \geq 1$ and $0 \leq \sum_{i=1}^{m} p_i \leq 1$. For each node v_i, $1 \leq i \leq n$, the operator defines $s^i = (s^i_1, \ldots, s^i_m)$ where s^i_j is the number of the times the value v_i has been included into the sample j, $1 \leq j \leq m$. A simple random sampling ensures that: $\forall j \in [1..m]$, $\sum_{i=1}^{n} s^i_j \simeq n \cdot p_j$, while in a "without replacement" strategy, the operator also ensures that: $\forall i \in [1..n]$, $\sum_{j=1}^{m} s^i_j \leq 1$.

As in the discretization task, any XQuery expression may be placed in the body of the <*samples*> clause. For example, in the fragment of query below, a partition of 100 equal-size samples is created.

having samples `$index` **in** `for $i in (1 to 100) return 0.01`.

2.3 XML data mining

Constraints-based pattern mining. Nowadays, one of the most important open issue in frequent itemset mining is the too large number of generated patterns. The constraint-based pattern mining paradigm has been introduced with the aim of providing the analyst with a domain-dependent tool for driving the discovery process directly towards potentially interesting patterns.

In this section, we present an operator to handle constrained itemset mining from XML data. We focus on the semantic aspects and on the expressiveness offered by the operator here, whereas in Section 3.2 we discuss how to speed-up the computation by incorporating such constraints directly into the Apriori algorithm.

Example 2.4. We wish to discover correlations among authors in the `dblp` database. In our analysis, we consider only the patterns with a minimum support of 10% in which occurs some "leader" author and in which at most two authors received a PhD after 2002. We consider as "leader" those authors that have been awarded by the "IEEE" society or, alternatively, are directors of an active project. The resulting XQuake expression is the following.

```
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 $dept := collection("Dep")/Deparment[@id=$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 query is not hard to understand for readers familiar with XQuery. 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 next statement uses a similar syntax to identify items of a transaction, i.e. the authors of a publication, binding each XML node `Author` to the variable `$author`. The keyword <*item*> specifies in this case that we are iterating over the items of a transaction. The <*let active*> clause uses a built-in function to format the required author name, i.e. the atomic values in the itemset. In addition, we bind to the variable

`$employee` an XML element encoding the domain knowledge (i.e. the employee information of the department of interest to each distinct author - see also Figure 1b). This is achieved through the <*let metadata*> statement containing, in the body, an XQuery expression that first searches the department of interest among the collection of the various departments, and then searches the author name among its employees. Notice the use of both the `$aut` and `$aut-name` variables in the body expression.

The set of the itemset constraints occur in the <*having*> clause. Specifically, they constraint the number of the items of an itemset that satisfy 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 that, in the example 2.4, denote both the author's name and the employee metadata. The examples below explain 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 in the Cambridge University.

As a further requirement, we restrict the source data to the papers appeared in journals or proceedings focussed on the "KDD" field. Notice that the latter one is a data filtering constraint, in contrast to knowledge constraints expressed on the itemsets.

```
for data $article in db("DBLP")/(inproceedings|article)
where fn:contains($article/(abstract|title), 'KDD')
```

Notice that now the resulting transactions are a subset of the original ones and the relative minimum support assumes the right meaning on the new set of data. Also observe the use of the built-in XQuery function `fn:contains`. It makes it easier to write specific full-text queries and typically its implementation is based on physical indexes to speed-up the execution.

In the above queries we have supposed the availability of metadata. In

several cases, XQuery can be also 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.

```
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.

```
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 it does not exist one author that is also editor.

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

The versatility of the <*mine itemsets*> operator permits to easily comply with the domain and the interpretation of the items. For example, if we aim at finding the correlations among the research topics that have been investigated in recent years, we can use the following query.

```
mine itemsets doc("topics") using alg:apriori(0.01)
for data $paper in doc("dblp")/(inproceedings|article)
for item $keyword in $t/keywords/*.
let active field $item := string($keyword)
```

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 2% of the total.

In the next example, we exploit categorical attributes, instead of boolean ones.

Example 2.5. In the `mondial` dataset, we are interested in finding correlations among the kind of religions and the government type of a country. In other terms, we are interested in itemsets such as: {`Orthodox = medium, Catholic = high, government = monarchy`} `supp=0.35`.

```
mine itemsets doc("co-religions") using alg:apriori(0.25)
for data $country in doc("mondial")/Country
```

```
for item $religion in $country/(government|religions)
let active field $item := local:format($religion)
```

In this query, each country denotes a transaction, while an item is either the government type or the religion followed by a certain population percentage. Inside the <*let*> clause, a user-defined function transforms the input XML element into a "categorical version" of the item.

Tree classification. Let's us now consider another typical data mining problem, the classification problem. A generic classification task consists of two phases: the *training phase* in which a set of classified data is analyzed in order to extract a classification model, and a *predictive phase*, in which the model is used to predict the class label of unknown records. In the rest of this section we focus on the training phase, and, especially, on the issue of specifying the input to a decision tree algorithm in the <*mine tree*> XQuake operator.

The example below selects 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.

Example 2.6. 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 population and the level of inflation (see example 2.1);
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 an extension greater than a fixed value.

The data mining task that leads to the generation of the classification model can be specified by means of the <*mine tree*> operator:

```
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/@disr-as
let active field $inflation := $c/inflation/@discr-as
let active field $is-fao := if (fn:contains($c/memberships, 'org-FAO'))
                            then 'yes' else 'no'
let active field $ext-cap := let $c-d := $c//city[@is-capital = 'yes']/gdp
                             return if ($c-d > 10000) then 't' else 'f'
```

```
let predictive field $class := if (fn:contains($c/memberships, 'UNESCO'))
                               then 'yes' else 'no'
```

The statement can be read as follows. The <*using*> clause specifies the mining algorithm, ID3 [25] in this case. The <*for*> clause identifies the XML nodes that denote the records of the training set. Each <*let*> element specifies the attributes in the source data set tat are 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`. In this example, attributes are explicit, but the operator also admits the specification of group attributes.

2.4 XML post-processing enhancement

An extremely important feature in database mining is the ability to correlate the generated knowledge with further data.

The <*apply itemsets*> operator in XQuake provides the capability of connecting the extracted itemsets with the data. It takes as input a PMML association model and a set of XML transactions. For each transaction, the operator gives information about the number of itemsets that are supported or violated by that transaction. More formally, given an itemset $I = \{I_1, \ldots, I_n\}$, and a transaction $T = \{I_1, \ldots, I_m\}$, T supports I, i.e. $support(T, I)$, if $\forall i \in [1, n]$: $I_i \in T$. T violates I, i.e. $violate(T, I)$, if $\forall i \in [1, n]$: $I_i \notin T$.

Example 2.7. With reference to the `mondial` dataset, the following query returns as output the set of triplets, (c, v, s) where c is the country name and v (resp. s) is the percentage of the itemsets that violate (resp. comply with) c, over the total number of itemsets.

```
apply itemsets doc("info-supported-countries")
let association model $m := doc("co-religions")
let supplementary field $total := pmml:get-number-itemsets($m)
for data $country in doc("new-mondial")/Country
for item $religion in $country/(government|religions)
let active field $item := local:format($religion)
having supported is $s, violated is $v
return ($country/name, ($v div $total) * 100, ($s div $total) * 100)
```

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

A similar operator, namely <*apply tree*>, uses a decision tree to predict the class label of unknown data. Both operators also permit one form of nesting

within the mining query:

```
apply itemsets doc("my-doc-name-1")
let association model $m := (mine itemsets doc("my-doc-name-2") ...
                             < mine itemsets specification >)
< apply itemsets specification >
```

Generally, the language satisfies a closure principle, namely that any operator returning type t can be used wherever an argument of type t is required.

3 The physical level

The implementation of the aforementioned operators 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. Consider for example the XQuake expression of the Example 2.4. It contains both a data filtering condition, expressed by means of the <*where*> clause, and some knowledge constraints located in the <*having*> clause. The data filtering may be evaluated at preprocessing time, or more efficiently, during the first iteration of the frequent itemset mining algorithm. Constraints on itemsets can be classified according to the peculiar property they satisfy (e.g. monotone, anti-monotone, succinct, convertible, etc.) [5]. For each of these classes, there exists a specialized algorithm which is able to take advantage of such properties, for instance, by evaluating the constraint during the exploration of the search space instead of at post-processing time.

In [1], the authors consider a spectrum of architectural alternatives for coupling mining with database systems. Two orthogonal solutions are:

- 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*);
- 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*).

Both the solutions suffer of some limitations. The main limitation of the first kind of approach is that they make it difficult to implement ad-hoc data structures and procedures that allow physical optimizations for the algorithm [1]. Over the last years, only the SQL-based language ATLaS [23] has been designed to overcome this limitation, by means of well-defined mechanisms via a tightly-coupled approach.

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. Nowadays, there is a plethora of high-level loosely-coupled DM systems (e.g. [36, 28, 11]). They offer a mixture of preprocessing, mining and post-processing features, but they do not allow the encapsulation of the background knowledge deeply in the mining process, nor the rigorous formalization of mining constraints.

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" of the 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 remainder of this section is organized as follow. The purpose of Section 3.1 is to extend XQuery to support mining primitives. The goal is to have a powerful extension that is appropriate for all use cases, including preprocessing and mining algorithms. Section 3.2 discusses in detail the implementation of the *<mine itemsets>* operator and in particular of the Apriori algorithm. Finally, Section 3.3 presents the system architecture of XQuake.

3.1 Extending XQuery for a better implementation of data mining primitives

Motivation. The capability of iterating on data is the basis of KDD algorithms. Many base 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 := init();
do {

```
        for each tuple t in D {
                S := f(S, t);                                    (⋆)
        }
} while(g(S));
return h(S);
```

where *init*, f, g and h are user-defined functions.

The above iterative schema is common to many frequent itemsets, sequential patterns, classification and clustering algorithms. 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 *init* function at each iteration of the external loop, or we can replace the *for* loop with a windowing computation, in which the overall dataset is split into *data windows* (i.e. a subset 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* ($\star$)? 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 FLOWR expression are twofold. From one side, it is difficult to define queries over windows of data and this complicates the implementation of data preparation tasks. From the other side, also the managing of local temporary variables in an iterative computation is not straightforward. Consider for example the following simple FLOWR expression.

```
let $i := 0
for $j in (1,2,3)
let $i := $i + $j
return $i.
```

Users new to XQuery sometimes expect one evaluation of the body of a `for` expression to affect later evaluations, as it would be 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 ($\star$) 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 `FLOWR` expression and capable of overcoming the limitations reported above. We integrated such extension into a Java-based open source XQuery engine (see Section 3.3).

Run-through examples. The `wfor` statement iterates over an input sequence and it binds a variable with every iteration. In addition, the statement permits to specify an `init-next-close` computation like for data cursors. A complete grammar is given in Figure 2a.

The `init` clause declares and initializes a local variable with a syntax similar to the standard `let` statement. The effect is to introduce the variable named by the `init` - 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 `next`, `close` as well as for the `open` and `until` clauses when present. The state variable is updated at each iteration with the result of the `next` clause, whose aim is to consume the sequence item by item. Finally, the `close` clause returns an output value of the overall computation. Observe that any XQuery expression can be used in the `init`, `open`, `next` and `close` statements, including `FLOWR` expressions or nested `wfor` expressions. This feature allows one to iterate over an input sequence several times.

Example 3.1 (average absolute deviation). In order to compute the average absolute deviation of a sequence of numeric values, `$seq`, the following query can be used.

```
wfor $value1 in $seq
   init $m as xs:int* := (0, 0)
   next ($m[1] + $value1, $m[2] + 1)
   close (let $mean := $m[1] div $m[2]
          wfor $value2 in $seq
             init $sum as xs:double := 0
             next $sum + fn:abs($mean - $value2)
             close $sum div $m[2]).
```

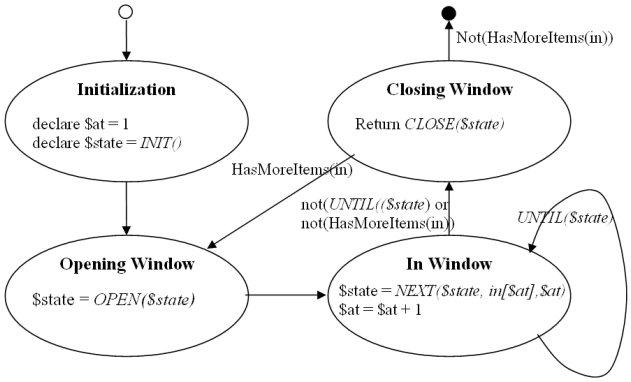
Two scans are performed over the input sequence. During the first iteration (external `wfor`), we calculate the mean value. The `$value1` variable is bound to each item of the input sequence. The state variable, `$m`, stores the temporary result of the computation and contains the current sum and the number of visited elements. `init` initializes the total sum and the count to 0. `next` consumes an item of the sequence and updates `$m` until further items are available. `close` terminates the first scan of data by computing the mean of the items. During the second iteration (internal `wfor`), for each item the value of the state variable is increased with the absolute difference between that item and the mean. `close` returns the final result (a single value in this case).

In order to specify more complex computations, we permit the specification of simple tumbling window queries[3] by adding an `until` clause and an `open`

[3]In tumbling windows items do not overlap. There are other different kinds of windows,

```
WforExpr ::= "wfor" "$" VarName AtClause?  "in" ExprSingle
             InitClause OpenClause? NextClause CloseClause
AtClause ::= "at" "$" VarName
InitClause ::= "init" "$" VarName ("as" TypeDeclaration)?  ":=" (FLWORExpr | WforExpr)
OpenClause ::= "open" (FLWORExpr | WforExpr)
NextClause ::= "next" (FLWORExpr | WforExpr) ("until" ExprSingle)?
CloseClause ::= "close" (FLWORExpr | WforExpr)
```

(a)



(b)

Figure 2: Syntax (a) and intuitive semantics (b) of the `wfor` iterator.

clause (see Figure 2a). Basically, the first one permits (i) to break the execution of `next` and (ii) to anticipate the evaluation of `close` 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 `open` statement.

Example 3.2 (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:

```
wfor $value in $seq
   init $state as xs:int* := ()
   open (0,0)
   next ($state[1] + $value, $state[2] + 1) until $state[2] < 100
   close $state[1] div $state[2].
```

In this example, a new window is opened at each evaluation of `open` by re-initializing the state variable every time. `next` consumes an item and updates `$state` as usual. Each time the `until` clause holds - i.e. the number of items

i.e. *sliding windows* and *landmark windows*. The state-of-the-art in extending XQuery with window functions is out of the scope of this paper. We refer the interested reader to [6].

read is 100 - `close` terminates the computation over the current window and it returns as output the mean value. The iteration continues by evaluating `open` on a new window of values, until the items of the input sequence are entirely consumed.

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

Example 3.3 (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 `$seq` is the sequence $(10, 20, 30, 40)$, the output sequence is $(10, 15, 20, 25)$. The `wfor` expression is as follows.

```
wfor $value in $seq
   init $state as xs:int* := (0,0)
   next ($state[1] + $value, $state[2] + 1) until fn:false()
   close $state[1] div $state[2].
```

The intuitive semantics of the `wfor` iterator can be better explained by means of the automaton depicted in Figure 2b. The formal semantics is reported in Appendix B.

wfor for data mining. In the following, we provide some examples on how the `wfor` operator can facilitate the definition of DM tasks.

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 techniques are *smoothing* techniques [27] (Chapter 9).

Example 3.4 (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:

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

```
wfor $v at $k in $X
   init $window as xs:double* := ()
   open if (empty($window)) then () else fn:remove($window, 1)
   next ($window, $v) until $pos < (2 * $k) + 1
   close local:median($window).
```

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

Example 3.5 (linear scaling transformation) The *linear scaling normalization* [27] (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.

```
wfor $i in $seq
  init $min-max := ($i, $i)
  next let $new-min := if ($min-max[1] < $i) then $min-max[1] else $i
       let $new-max := if ($i > $min-max[2]) then $i else $min-max[2]
       return ($new-min, $new-max)
  close for $j in $seq
        return ($j - $min-max[1]) div ($min-max[2] - $min-max[1]).
```

Example 3.6 (sorted-neighborhood method). The query of example 3.4 can be also adapted to the problem of merging multiple datasets, frequently encountered in DM as a crucial first step of the KDD process. For example, the basic *sorted-neighborhood method* [16] 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 `close` statement can be easily rewritten to ensure this task.

For efficiency purposes, we also conceived 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 hash-table:

```
let $my-hash-table as java-object := java.utils.HashTable:new(10).
```

We can also use instance methods in an intuitive way:

```
let $value := java.utils.HashTable:get($my-hash-table, 'my-key').
```

3.2 Implementation of the frequent pattern mining task

In providing an efficient implementation of the <*mine itemsets*> operator (Section 2.3), and of the Apriori algorithm [2], 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.

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.
- `t-tree`: an external total support tree [10] 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 out 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 operate as follows.

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 the `t-map`, setting the count to 1 and joining that item value with the metadata located through the <*let metadata*> clause(s). By using such information, we are able to evaluate, during the first data iteration, the atomic predicates of the mining constraints (i.e. the <*satisfies*> clauses of the <*at least*>, <*at most*>, <*exactly*> conditions) 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.

```
1. declare function local:apriori($t-tree as java-object,
2.                                $t-map as java-object,
3.                                $t-red as xs:integer*,
4.                                $k as xs:integer) as node* {
5.  wfor $t at $pos in mi:for-data()
6.    init $s := TTree:candidateGen($t-tree, $k)
7.    next if (functx:is-value-in-sequence($t-red, $pos))
8.         then $s
9.         else let $items as xs:string* := mi:format(mi:for-item($t))
10.             return TTree:suppCounting($s, $items, ($k + 1), $t-map)
11.   close if (TTree:AM-Pruning($s, $t-map, $k, $mi:min-supp) <= 1)
12.         then TTree:M-Pruning($s, $t-map)
13.         else local:apriori($s, $t-map, $t-red, ($k + 1))
14. };
```

Figure 3: The main procedure of the Apriori algorithm.

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 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 by using a FLOWR expression that incorporates external functions over `t-tree` and `t-map`. We omit the code for sake of brevity.

The Apriori algorithm. The Apriori [2] operates in a level-wise fashion moving bottom-up in the itemset lattice, from small to large itemsets. It requires several iterations on data, according to the size of the largest found itemset. The main implementation of the Apriori is shown in fig. 3. The implementation recursively extends the `t-tree` level by level, until no more frequent itemsets can be found. The function takes the (references to) `t-tree` and `t-map` as well as the `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 scans the input transactions. The `for-data` (resp. the `for-item`) function basically contains the code of the <*for data*> (resp. <*for item*>) XQuake statement. The `init` clause extends the total tree to a new level. The external function `candidateGen($t-tree, $k)` combines the frequent k-itemsets in `t-tree` to generate the candidate itemsets of size k+1.

The `next` 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 `format` that encapsulates

the code of the <*let active*> XQuake statement. 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 `close` routine prunes the results and tests whether no more frequent itemsets can be found. This is achieved by means of the external function `AM-Pruning($t-tree, $t-map, $k, $ms)` (line 11), that uses the anti-monotone property to filter out the candidates from `$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 below). `t-map` can be used to check whether such a constraint holds. The function returns the number of itemsets, at the given level `$k`, that survive the pruning. If there are surviving itemsets, then the Apriori algorithm is invoked recursively (line 13) to extend the total tree to a new level k+1 and the process is repeated until no more candidates can be found. At this point, the function `M-Pruning($t-tree, $t-map)` (line 12) prunes the frequent itemsets that do not satisfy a monotone constraint from `t-tree`. Again, `t-map` encodes such information. The external function returns the frequent PMML itemsets as a sequence of XML nodes.

Constraint evaluation. In the definition of the aforementioned Apriori schema we considered the encapsulation of two important kinds of constraints: *anti-monotone* and *monotone* ones.

Since any conjunction of anti-monotone constraints is anti-monotone, this kind of constraints is exploited in conjunction with the frequency constraint, by avoiding the generation of candidate itemsets that have an infrequent subset. As before mentioned, we encapsulated such exploitation level by level in the external function `AM-Pruning` of the Apriori schema. The monotone constraints are exploited at post-processing time on the total tree `t-tree`, by means of the external function `M-pruning`, and before producing the output itemsets.

Two essential questions arise: (i) *Are the constraints defined in the* <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 to use a more efficient synergy among this two kinds of constraints*?

In order to answer the first question, Figure 4a 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 *cond*. The following proposition holds:

Proposition 3.1. In the <*mine itemsets*> operator, the <*at most*> clause is anti-monotone. The <*at least*> clause is monotone. The <*exactly*> clause can be treated as a conjunction of the <*at least*> and <*at most*> constraints.

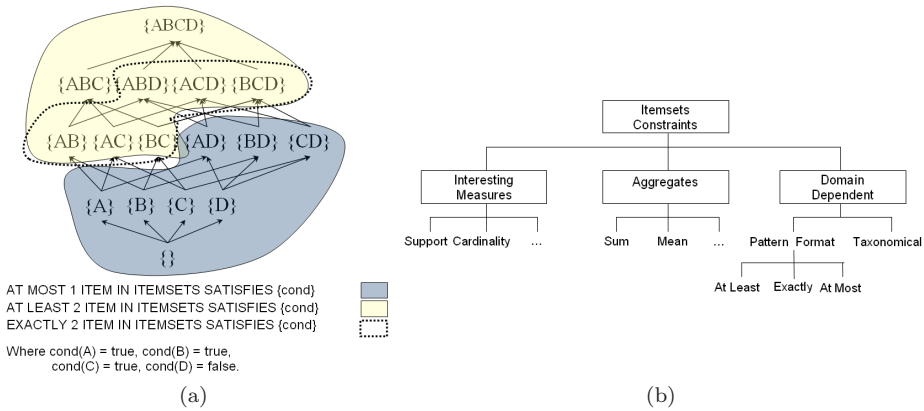


Figure 4: Monotonicity and anti-monotonicity in the `MINE ITEMSETS` operator (a) and the hierarchy of the itemset constraints (b)

On the other hand, the answer to the question (ii) is provided by the constrained itemsets mining literature. In fact, the evaluation of monotone constraints entirely at post-processing time may yield an efficient and sometimes expensive mining and several proposals exploited the synergy between anti-monotonicity and monotonicity [22, 8, 4, 5]. In this paper we implement, by means of a simple `wfor` iterator, the preprocessing data reduction algorithm *ExAnte* [4, 5], recently proposed by Bonchi et. al. as a means to reduce both the search space and the input dataset in constrained frequent patterns mining.

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. We are inside a virtuous cycle, where two different kinds of data-reductions cooperates to reduce the search space and the input database.

In Figure 5, we provide an XQuery implementation of the ex-ante procedure as a preprocessing task. The function takes as input `t-map` and `t-red` initialized during the mining preparation phase. Moreover, also the current number of frequent items is recorded. During the `init` phase, we set the support count of frequent items in `t-map` to 0 (line 5). The state variable contains the starting list of transactions doomed to be removed (line 6).

Each transaction index is then tested against the state variable to check whether it has been already pruned (line 7). Only if the transaction passes

```
1. declare function local:ex-ante($t-map as java-object,
2.                                $old-n-items as xs:integer,
3.                                $t-red as xs:integer*) as xs:integer* {
4.  wfor $t at $pos in mi:for-data()
5.    init $reduced := let $i := j-items:reinitSupports($t-map)
6.                     return $t-red
7.    next if (functx:is-value-in-sequence($t-red, $pos))
8.         then $reduced
9.         else let $items := mi:fomat(mi:for-item($t))
10.             return if (j-items:M-pruning($t-map, $items))
11.                    then ($reduced, $pos)
12.                    else $reduced
13.   close let $n-freq := j-items:pruneItems($t-map, $fi:min-supp)
14.         return if ($n-freq < $old-n-items)
15.                then local:ex-ante($t-map, $n-freq, $reduced)
16.                else $reduced
17. };
```

Figure 5: The Ex-Ante procedure at the first level of the level-wise computation.

this test, it is then tested against the conjunction of monotone constraints (line 10). The external function `M-pruning($t-map, $items)` first deletes from the overall items in the transaction, the ones that are infrequent (as stated before, this information is encoded in `$t-map`). This is the α-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 11). This is the μ-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 13) according to the anti-monotonicity of the support constraint. The `$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 15) until a fix point is reached and no more items are found to have turned infrequent (line 16). The `ex-ante` function updates `t-map` at each recursive invocation and it returns the new list of indexes of the reduced transactions. At this point, the `apriori` function of Figure 3 runs on such data structures and it computes the overall frequent itemsets.

Summing up, we built a recursive implementation of the Ex-Ante and Apriori algorithms, demonstrating that the encapsulation of several kinds of both domain-dependent and physical optimizations is possible in an XQuery-based solution. Further kinds of constraints may be exploited in our environment (see Figure 4b) by analyzing other interesting properties besides monotonicity and anti-monotonicity. In this respect, our attention is particularly placed on constraints over item hierarchies.

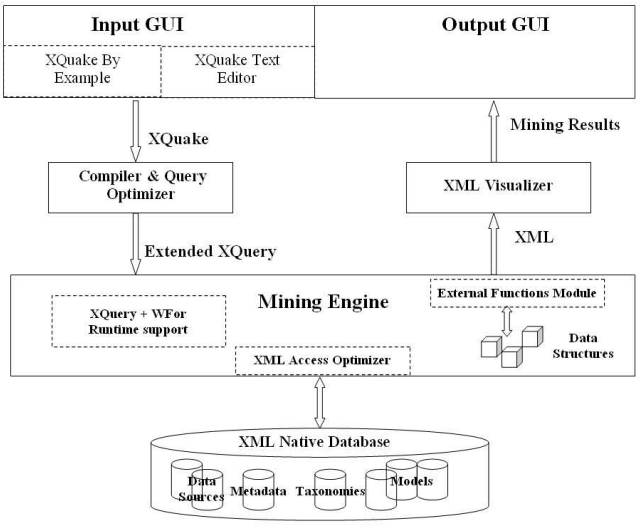


Figure 6: XQuake system architecture.

3.3 System Architecture

The XQuake prototype is implemented in Java. The system is built on top of BaseX [17], an open source native XML database developed by the Database and Information Systems Group at the University of Konstanz. BaseX has been chosen among several XQuery implementations, thanks to its compact storage structures, the efficiency and a visual front-end, facilitating interactive access to the data. In order to implement the `wfor` iterator presented in Section 3.1, 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.

A mining task is expressed in XQuake via a specific *text editor*. A planned extension is the implementation of a GUI according to the QBE philosophy.

No matter what kind of GUI the system adopts, a *compiler* automatically generates the appropriate (extended) XQuery code that is then interpreted. The compiler is designed to provide a good level of extensibility of the environment whenever the definition of a new algorithm is provided. This is achieved 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 operator into XQuery functions. For example, in the <*mine itemsets*> operator, the function `for-data()` (see Figure 3) addresses the XQuery code contained in the <*for data*> XQuake element. Such a function is invoked by the static program that is independent from the user-defined code of the XQuake operator. 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.

The core of the mining process is performed by the *mining engine* that contains the run-time support of the BaseX XQuery engine extended with the `wfor` iterator. This component uses the *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. The database is accessed by means of the *XML access optimizer* component that will contain (proprietary or native) indexing techniques to speed-up the access to input data. We remark that the process of integrating new indexes may be both manual (i.e. encapsulated into the XQuake syntax) or automatic.

Finally, the *XML visualizer module* translates an XML documents stored in the DB into a visualization form, accepted by data and model visualization tools, and presented by means of an *output GUI* to the user. Currently, the XML result is transformed into HTML browsable format via XSL style sheets.

4 Related Work

Our work is based on ideas and experience developed in the inductive databases field. For example, the concept of a logic-based KDD environment and the idea of integrating datalog with iterative user-defined predicates has been presented in [15]. The `wfor` clause presented in Section 3.1 follows the original idea of ATLaS [23], that uses the ability of defining User-Defined Aggregates (UDAs) for implementing mining primitives in a relational DBMS. Finally, the XQuake language presented in Section 2 has been partially inspired by the XMineRule operator [7].

Mining from relational data. The idea of integrating knowledge discovery into relational DBMS has been addressed in a considerable number of different papers [3, 9, 13, 21, 26, 35]. All these approaches extend SQL with operators for extracting models from relational data. The contributions of this research includes (i) the capability of satisfying the closure property, and (ii) the capability of providing a mixture of data pre-processing, mining and model post-processing. However, the relational model suffers of some limitations.

On the one 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. ATLaS [23] is a recent proposal for handling data stream that also tries to answer the KDD needs. 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. As mentioned, UDAs have inspired the idea of `wfor`.

On the other hand, SQL-based approaches generally suffers of low expressiveness in representing mining models and domain knowledge, and the extension to deal with further models and new user-defined functionalities does not seem so easy. At the best of our knowledge, this is still an open problem.

Mining from XML data. An important aspect to take into account in DM is how to make all the heterogeneous patterns, sources of data and other KDD objects coexist in a single framework. A solution considered in the last few years is the exploitation of XML as a flexible instrument for IDBs [24, 28, 11, 7].

In [24] XML has been used as the basis on which a semi-structured data model designed for KDD, called XDM, is defined. The model stores both data and mining models in the same XML database. This allows the reuse of patterns by the inductive database management system. The perspective suggested by XDM is also taken in the KDDML [28] and Yale [11] view. Essentially, the KDD process is modeled as an XML document and the description of an operator application is encoded via an XML element. The result achieved by these proposals is a more flexible way to represent heterogeneous patterns as well as the mining process, overcoming, in this respect, the relational model. Both KDDML and XDM integrate XQuery expressions into the mining process. For instance, XDM encodes XPath expressions into XML attributes to select sources for the mining, while KDDML uses an XQuery expression to evaluate a condition on a mining model. In our opinion, XQuake offers a deeper amalgamation with the XQuery language and consequently a better integration among data mining and XML native databases.

One of the first attempts to extract association rules by implementing the APriori algorithm directly using XQuery is [34]. The proposed work does not offer an integrated mechanism to support efficient data-intensive computations, and the Apriori implementation is highly inefficient.

The XMineRule operator [7] defines the basic concept of association rules for XML documents. Two are the main differences with respect to XQuake. From the physical point of view, XMineRule requires that the data are mapped to the relational model and it uses SQL-oriented algorithms to do the mining. Also the output rules are translated into an XML representation. As a consequence, the loosely-coupled architecture of XMineRule makes it difficult to use optimizations based on the pruning of the search space, since constraints can be evaluated only at pre- or post-processing time. From the semantics perspective, items have an XML-based hierarchical tree structure in which rules describe interesting relations among fragments of the XML source [12]. In contrast, in our approach items are denoted by using simple structured data from the domains of basic data types, favouring both the implementation of efficient data structures and the design of powerful domain-specific optimizations. Furthermore, knowledge is linked to items through XML metadata elements.

5 Performance evaluation

In the following we analyze the impact of the architecture presented in Section 3 on the frequent itemsets problem. In order to compare the performance with an existent Apriori implementation, we tested the effects of a very simple XQuake query, without considering the encapsulation of any kind of constraints. Below, we show a fragment of an input XML database used to test our Apriori together with the <*mine itemsets*> operator for pattern extraction.

```
<data db_name="census">
   <transaction id="123">
     <item>age=1</item>
     <item>workclass=Private</item>
     ...
   </transaction>
   <transaction id="124">
     <item>age=2</item>
     <item>workclass=Local-gov</item>
     ...
   </transaction>
...
</data>
```

mine itemsets `doc("my-out")` **using** `alg:apriori(0.01)`
for data `$data` **in** `doc("census")/data/*`
for item `$item` **in** `$data/*`
let active field `$field := string($item)`

Datasets. We used both real and synthetic datasets stored into the native database BaseX with proprietary standard indexes on XML elements and attributes. The real datasets are from the UCI repository [31]. The synthetic databases are generated by the IBM generator [19]. These datasets are named in the form of "TxIyDz" according to the parameters x, y, z indicating the average number of items in the transactions, the average number of items in the large itemsets, and the number of transactions in the database, respectively. A summarization of the databases used in our experiments is outlined in Table 1, in which the last two columns identify the lower minimum support threshold used in the experiments and the number of extracted patterns. The choice of the datasets is mainly motivated by the need to estimate the contribution of the XQuake approach from dense datasets (`Connect` and `Mushroom`) to more sparse ones (`T20I6D300K`).

Experiments. 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. Appendix C illustrates the performance obtained on the datasets by varying the value of the minimum support. 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 [36] (Figure 7, 8 and 9). We

Name	Real	Avg trans	Num trans	Num items	Min supp (%)	Num itemsets[1]
census	yes	15	48,841	135	2	70,826
mushroom	yes	23	8,122	119	10	574,513
connect-4	yes	43	67,556	129	88	55,115
T20I6D100K	no	20	100,000	1,000	0.2	25,820
T30I10D100K	no	30	100,000	1,000	0.2	108,634
T20I6D300K	no	20	300,000	3,000	0.2	7,457

Table 1: Summary of datasets for experiments. [1]The number of itemsets refers to the minimum support used.

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 [10]. 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 (Figure 10), on which the performance of the algorithm resulted to be quite stable. The performance overhead introduced by external Java functions integrated in XQuery is modest. This is confirmed by the graph of Figure 11, that reports the overall execution time on the testing datasets as a sum of: (i) the *initialization time*, i.e. the time to compile the query and to prepare the data structures; (ii) the *data iteration time*, i.e. the time to iterate over the data several times, according to the number of cycles of the Apriori; (iii) the *mining time*, i.e. the time to update data structures at each iteration; (iv) an estimation of the *overhead due to external functions* and finally (v) the *serialization time* to produce the output. The performance of the Apriori shows a worsening when the number of generated patterns is very high - more than 500,000 itemsets with 1,5GByte of memory (Figure 11b) . 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.

Note moreover that 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 to consider the query of Example 2.4. 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:

```
$paper/Author[fn:concat(./FirstName,' ',./LastName)].
```

Experiments showed that such a modification brings an increasing of the performance of about 13% in respect to the graphs reported in Appendix C. Even if with its simplicity, the above example shows that a thorough modification of the underlying XML optimizer needs to be investigated.

6 Conclusion

In this paper, we proposed a new query language as a solution to the XML data mining problem. As a starting point, we referred to several past works on XML and relational domains [23, 24, 7]. In our view, an XML native database is used as a storage for KDD entities. DM tasks are expressed in an XQuery-like language. Also we have determined what extensions are required for turning XQuery into a language supportive of the implementation of DM primitives (Section 3.1).

The first empirical assessment reported in Section 5 exhibits promising results, even if related to the XML frequent itemsets mining problem. In this context, an in-depth study of how to provide high-level optimizations by means of both a direct exploitation of background knowledge and efficient data structures has been conducted. In particular, the grammar of the language is flexible enough to specify a variety of different mining tasks by means of user-defined functions in the statements. These ones allow the user to provide personalized sophisticated constraints, based, for example, on domain knowledge. In addition, we push constraints deeper inside the mining process and, more important, this integration is automatic.

To conclude, our project has a general-purpose objective. Clearly, the study of this feature is even more substantial in XML-based languages, since no general-purpose XML mining language has been yet proposed (at the best of our knowledge). An interesting on-going work includes the exploitation of ontologies to represent the metadata. As an example, ontologies may represent enriched taxonomies, used to describe the application domain by means of data and object properties. As a consequence, they may provide enhanced possibilities to constrain the mining queries in a more expressive way. This opportunity is even more substantial in our project, since ontologies are typically represented via the Web Ontology Language (OWL) [32], de facto an XML-based language. The challenge we are going to raise is whether XQuery may be versatile enough to also support queries on OWL documents.

References

[1] Agrawal, R., Sarawagi, S., and Thomas, S. Integrating association rule mining with relational database systems: alternatives and implications. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1998), ACM Press, pp. 343–354.

[2] Agrawal, R., and Srikant, R. Fast algorithms for mining association rules. In *VLDB '94: Proceedings on the 20th International Conference on Very Large Data Bases* (12–15 1994), J. B. Bocca, M. Jarke, and C. Zaniolo, Eds., Morgan Kaufmann, pp. 487–499.

[3] Bernhardt, J., Chaudhuri, S., Fayyad, U., and Netz, A. Integrating data mining with SQL databases: OLE DB for data mining. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)* (2001), pp. 379–387.

[4] Bonchi, F., Giannotti, F., Mazzanti, A., and Pedreschi, D. Exante: A preprocessing method for frequent-pattern mining. *IEEE Intelligent Systems 20*, 3 (2005), 25–31.

[5] Bonchi, F., and Lucchese, C. Extending the state-of-the-art of constraint-based pattern discovery. *Data and Knowledge Engineering 60*, 2 (2007), 377–399.

[6] Botan, I., Kossmann, D., Fischer, P. M., Kraska, T., Florescu, D., and Tamosevicius, R. Extending xquery with window functions. In *VLDB '07: Proceedings of the 33rd international conference on Very Large Data Bases* (Vienna, Austria, 2007), VLDB Endowment, pp. 75–86.

[7] Braga, D., Campi, A., Ceri, S., Klemettinen, M., and Lanzi, P. L. A tool for extracting xml association rules. *Tools with Artificial Intelligence, IEEE International Conference on 0* (2002), 57.

[8] Bucilă, C., Gehrke, J., Kifer, D., and White, W. Dualminer: A dual-pruning algorithm for itemsets with constraints. *Data Min. Knowl. Discov. 7*, 3 (2003), 241–272.

[9] Ceri, S., Meo, R., and Psaila, G. An extension to SQL for mining association rules. *Data Min. Knowl. Discov. 2*, 2 (1998), 195–224.

[10] Coenen, F., Leng, P., and Ahmed, S. Data structure for association rule mining: T-trees and p-trees. *IEEE Transactions on Knowledge and Data Engineering 16*, 6 (2004), 774–778.

[11] Euler, T., Klinkenberg, R., Mierswa, I., Scholz, M., and Wurst, M. YALE: rapid prototyping for complex data mining tasks. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, 2006), ACM Press, pp. 935–940.

[12] Feng, L., and Dillon, T. S. Mining interesting xml-enabled association rules with templates. In *KDID* (2004), pp. 66–88.

[13] Fu, Y., Han, J., Koperski, K., Wang, W., and Zaiane, O. DMQL: A data mining query language for relational databases. In *SIGMOD'96 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'96)* (Montreal, Canada, 1996), pp. 122–133.

[14] Ghelli, G., Onose, N., Rose, K. H., and Siméon, J. A better semantics for xquery with side-effects. In *DBPL* (2007), pp. 81–96.

[15] Giannotti, F., Manco, G., and Turini, F. Specifying mining algorithms with iterative user-defined aggregates. *IEEE Transactions on Knowledge and Data Engineering 16*, 10 (2004), 1232–1246.

[16] Hernández, M. A., and Stolfo, S. J. The merge/purge problem for large databases. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1995), ACM, pp. 127–138.

[17] Holupirek, A., Grün, C., and Scholl, M. H. BaseX and DeepFS - Joint Storage for Filesystem and Database. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT 2009) - To Appear* (2009). `www.inf.uni-konstanz.de/dbis/basex/index.php`.

[18] IBM. Intelligent Miner 8.2, 2008. `www-01.ibm.com/software/data/iminer/`.

[19] IBM Almaden Research Center. Synthetic Data Generation Code for Associations and Sequential Patterns. `www.almaden.ibm.com/cs/disciplines/iis/`.

[20] Imielinski, T., and Mannila, H. A database perspective on knowledge discovery. *Comm. Of The Acm 39*, 11 (1996), 58–64.

[21] Imielinski, T., and Virmani, A. MSQL: A query language for database mining. *Data Mining and Knowledge Discovery 2*, 4 (1999), 373–408.

[22] Jeudy, B., and Boulicaut, J.-F. Optimization of association rule mining queries. *Intelligent Data Analysis 6*, 4 (2002), 341–357.

[23] Law, Y., Wang, H., and Zaniolo, C. Query languages and data models for database sequences and data streams. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases* (Toronto, Canada, 2004), pp. 492–503.

[24] Meo, R., and Psaila, G. An XML-based database for knowledge discovery. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT)* (2006), pp. 814–828.

[25] Mitchell, T. M. *Machine Learning.* McGraw-Hill, 1997.

[26] Morzy, T., and Zakrzewicz, M. SQL-like language for database mining. In *First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97)* (St.-Petersburg, September 1997), vol. 1, pp. 331–317.

[27] Pyle, D. *Data preparation for data mining.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[28] Romei, A., Ruggieri, S., and Turini, F. KDDML: a middleware language and system for knowledge discovery in databases. *Data Knowl. Eng. 57*, 2 (2006), 179–220.

[29] SPSS. Clementine 12.0, 2008. `www.spss.com/clementine`.

[30] The Data Mining Group. The Predictive Model Markup Language (PMML). Version 3.2. `www.dmg.org/pmml-v3-2.html`.

[31] The UC Irvine Machine Learning Repository. `http://archive.ics.uci.edu/ml/`.

[32] W3C World Wide Web Consortium. OWL Web Ontology Language. On-line documentation, W3C Recommendation 10 February 2004. `http://www.w3.org/TR/owl-features/`.

[33] W3C World Wide Web Consortium. XQuery 1.0: An XML Query Language. On-line documentation, W3C Recommendation 23 January 2007. `http://www.w3.org/TR/Query`.

[34] Wan, J. W. W., and Dobbie, G. Mining association rules from xml data using xquery. In *ACSW Frontiers '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation* (Darlinghurst, Australia, 2004), Australian Computer Society, Inc., pp. 169–174.

[35] Wicker, J., Richter, L., Kessler, K., and Kramer, S. Sindbad and siql: An inductive database and query language in the relational model. In *ECML/PKDD (2)* (2008), pp. 690–694.

[36] Witten, I., and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan & Kaufmann, 2000. Version 3.4.3 from `http://www.cs.waikato.ac.nz/ml/weka`.

A Syntax of the XQuake mining language

```
XQuakeMain ::= VarDecl* FuntionDecl* XQuakeOp
XQuakeOp ::= (DiscrOp | SamplingOp | MineItemsetsOp | MineTreeOp | ApplyItemsetsOp)

DiscrOp ::= "prepare" "discretization" "doc" "{" QName "}" UsingClause ForDataClause
            WhereClause? LetFieldClause+ HavingDiscrClause? ReturnClause
HavingDiscrClause ::= "having" BinExpr ("," BinExpr)*
BinExpr ::= "bin" "$" VarName "in" ExprSingle

SamplingOp ::= "prepare" "sampling" "doc" "{" QName "}" UsingClause ForDataClause
               WhereClause? LetFieldClause* HavingSamplingClause? ReturnClause
HavingSamplingClause ::= "having" "samples" "$" VarName "in" ExprSingle

MineItemsetsOp ::= "mine" "itemsets" "doc" "{" QName "}" UsingClause ForDataClause
                   WhereClause? ForItemClause LetFieldClause* HavingItemsetClause?
HavingItemsetClause ::= CardinalityExpr ("," CardinalityExpr)*
CardinalityExpr ::= ("at least"|"at most"|"exactly") ExprSingle "item"
                    "satisfies" ExprSingle

MineTreeOp ::= "mine" "tree" "doc" "{" QName "}" UsingClause ForDataClause
               WhereClause? LetFieldClause*

ApplyItemsetsOp ::= "apply" "itemsets" "doc" "{" QName "}" UsingClause LetAssocClause
                    LetFieldClause* ForDataClause WhereClause? ForItemClause
                    LetFieldClause* HavingApplyItemsetsClause ReturnClause
HavingApplyItemsetsClause ::= "supported" "is" "$" VarName
                              ("," "violated" "is" "$" VarName)?

UsingClause ::= "using" "alg:" QName "(" (ExprSingle ("," ExprSingle)*)? ")"
ForDataClause ::= "for" "data" "$" VarName TypeDeclaration? "in"
                  (ExprSingle | XQuakeOp)
ForItemClause ::= "for" "item" "$" VarName TypeDeclaration? "in" ExprSingle
LetFieldClause ::= "let" ("active"|"predicted"|"group"|"metadata"|"supplementary")
                   "field" "$"  VarName TypeDeclaration? WeightDeclaration?
                   ":=" ExprSingle
WeightDeclaration ::= "weight" Number
WhereClause ::= "where" ExprSingle
ReturnClause ::= "return" ExprSingle
LetAssocClause ::= "let" "association" "model" "$" VarName ":="
                   ("doc" "{" QName "}" | XQuakeOp)
```

B Dynamic semantics of `wfor` in XQueryU core with tuples

This appendix provides the definition of the core semantics of `wfor` in XQueryU [14].

B.1 Core semantics of `wfor`

As stated in [14], the dynamic semantics of the core is defined by the following judgment, where Σ is the dynamic environment, σ is the store, e is a core expression and v is a value:

$$\Sigma; \sigma \vdash e \Rightarrow v; \sigma^{'}.$$

Let $e_{window} \equiv$ **init**$\$s := e_{init}$ **open** e_{open} **next** e_{next} **until** e_{until} **close** e_{close}, the rules for providing the judgment of the `wfor` are:

$$\Rightarrow_{\text{wfor-1}} \frac{\begin{array}{c} \Sigma; \sigma \vdash e_{in} \Rightarrow (v_1, \ldots, v_n); \sigma_1 \qquad n > 0 \qquad \Sigma; \sigma_1 \vdash e_{init} \Rightarrow v^{init}; \sigma_1^{init} \\ (\Sigma, \$s \mapsto v^{init}); \sigma_1 \vdash e_{open} \Rightarrow v_1^{open}; \sigma_1^{open} \\ \forall i \in 1, \ldots, n : ((\Sigma, \$x \mapsto v_i), \$s \mapsto v_i^{open}); \sigma_i^{open} \vdash e_{next} \Rightarrow v_i^{next}; \sigma_i^{next} \\ \forall i \in 1, \ldots, n-1 : (\Sigma, \$s \mapsto v_i^{next}); \sigma_i^{next} \vdash e_{window} \Rightarrow (v_{i+1}^{open}, v_i^{out}); \sigma_{i+1}^{open} \\ (\Sigma, \$s \mapsto v_n^{next}); \sigma_n^{next} \vdash e_{close} \Rightarrow v_n^{out}; \sigma_{n+1} \end{array}}{\Sigma; \sigma \vdash \textbf{wfor } \$x \textbf{ in } e_{in}\ e_{window} \Rightarrow (v_1^{out}, \ldots, v_n^{out}); \sigma_{n+1}}$$

$$\Rightarrow_{\text{wfor-2}} \frac{\Sigma; \sigma \vdash e_{in} \Rightarrow (); \sigma^{'}}{\Sigma; \sigma \vdash \textbf{wfor } \$x \textbf{ in } e_{in}\ e_{window} \Rightarrow (); \sigma^{'}}$$

$$\Rightarrow_{\text{window-true}} \frac{\Sigma; \sigma \vdash e_{until} \Rightarrow v; \sigma^{'} \qquad v \neq ()}{\Sigma; \sigma \vdash e_{window} \Rightarrow (\Sigma(\$s), ()); \sigma^{'}}$$

$$\Rightarrow_{\text{window-false}} \frac{\begin{array}{c} \Sigma; \sigma \vdash e_{until} \Rightarrow (); \sigma^{'} \qquad \Sigma; \sigma^{'} \vdash e_{close} \Rightarrow v; \sigma^{''} \\ \Sigma; \sigma^{''} \vdash e_{open} \Rightarrow v^{'}; \sigma^{'''} \end{array}}{\Sigma; \sigma \vdash e_{window} \Rightarrow (v^{'}, v); \sigma^{'''}}$$

C Performance graphs of XQuake

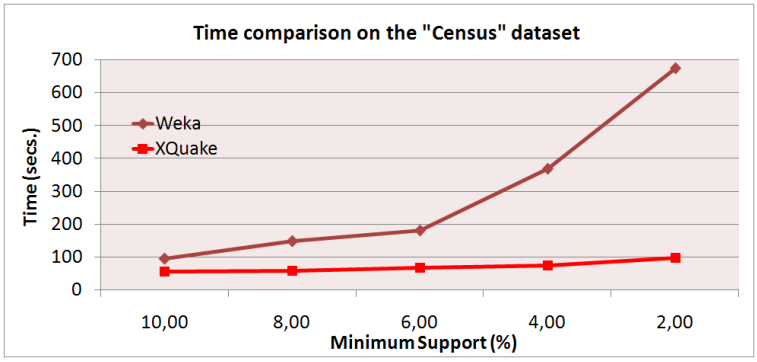


Figure 7: Runtime comparison among XQuake and Weka on the census dataset.

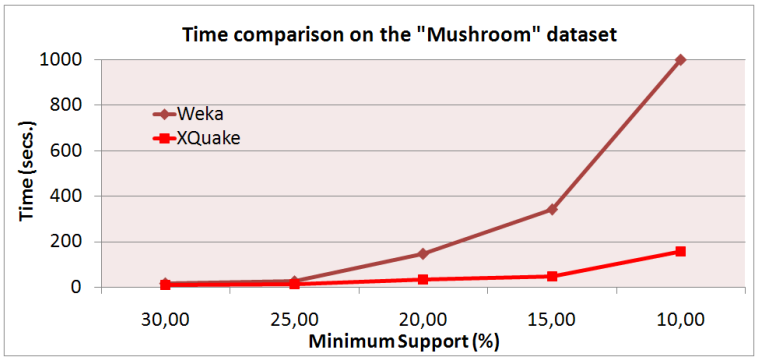


Figure 8: Runtime comparison among XQuake and Weka on the mushroom dataset.

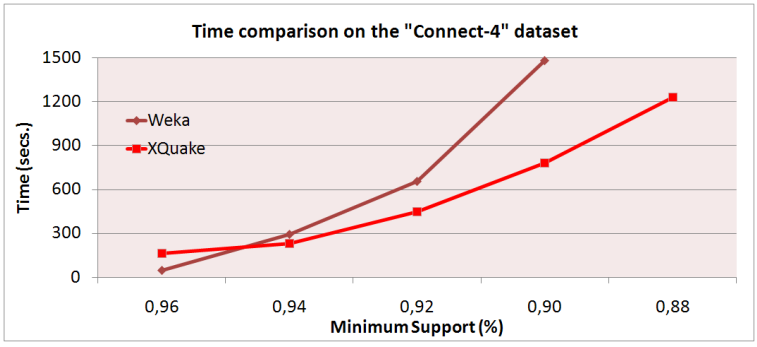


Figure 9: Runtime comparison among XQuake and Weka on the connect-4 dataset.

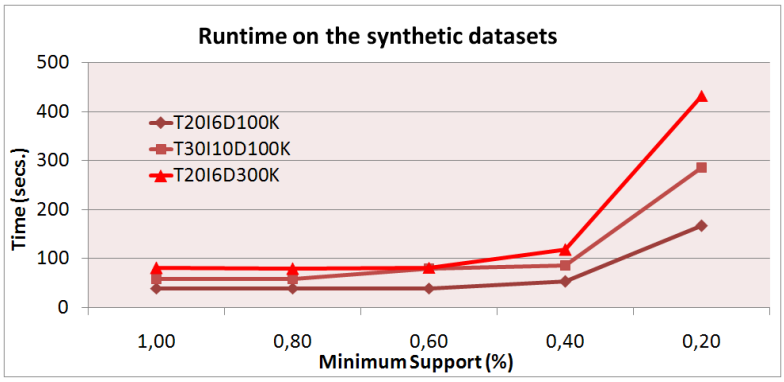


Figure 10: XQuake performance on the synthetic datasets.

Figure 11: Time breakdown on the `census` dataset (a), `mushroom` dataset (b), `connect-4` dataset (c) and `T20I6D300K` dataset (d).