UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Feature grammar systems. Incremental maintenance of indexes to digital media warehouses

Windhouwer, M.A.

Publication date
2003

Link to publication

Citation for published version (APA):
Windhouwer, M. A. (2003). *Feature grammar systems. Incremental maintenance of indexes to digital media warehouses*. [Thesis, externally prepared, Universiteit van Amsterdam].

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:18 Nov 2026

Chapter 5

Feature Databases

Systems have sub-systems and sub-systems have sub-systems and so on ad finitum - which is why we're always starting over.

Every program is a part of some other program and rarely fits.

Alan J. Perlis – *Epigrams on Programming*

The FDE implementation described in the previous chapter produces a forest of parse trees, *i. e.* elementary and auxiliary trees. These trees are stored in a feature database for two reasons: primarily as a persistent buffer for the on line use by the DMW search engine and, secondarily, as a lookup table for memoized detector calls. The parse trees produced by the FDE are in fact XML documents. The mass storage of XML documents has been a major research topic since the rise of XML as *the* data exchange format for Internet-based applications. In this chapter the storage scheme for XML documents as used by the current Acoi implementation is described in more detail. The backend of this XML mapping is the Monet database kernel. The Acoi system functioned as a test case for many of its unique aspects. These unique aspects are introduced in the next section, while the other sections will reflect on the mapping used and the lessons learned.

5.1 The Monet Database Kernel

The Monet database kernel [BK95, Bon02] provides, for main memory optimized, access to *Binary Association Tables* (BATs). BATs are the actual implementation primitives for the *Decomposed Storage Model* (DSM) [CK85]. On top of this kernel several front-ends have been build. These front-ends use the extensibility features of Monet: the Monet Interpreter Language (MIL) [BK99] and its dynamic loading mechanism for accessing libraries of *C* code. In the case of the relational model

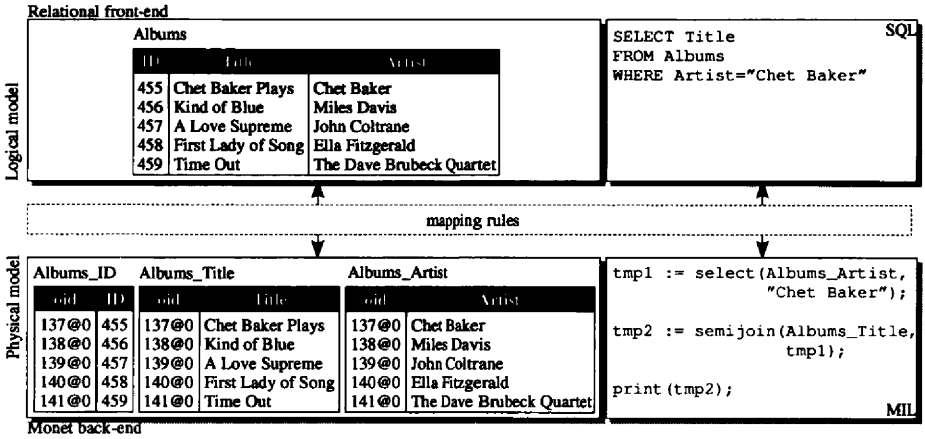

Figure 5.1: A relational front-end for Monet.

tables are vertically decomposed into binary tables, see Figure 5.1. SQL queries are translated into MIL commands which provide access to the appropriate BATs.

5.1.1 Monet and XML

Just as for the relational model an XML specific Monet front-end can be build. In fact several of such front-ends have been built, and they will be shortly described in this section. Where appropriate the mappings are illustrated with (parts of) the parse forest shown in Figure 4.9, assuming that confidences (ρ) are stored as attributes, that end-of-sentence markers ($) are not stored, and that all leafs contain a lexical instantiation.

An easy way to store XML documents in a database is into a *binary large object* (BLOB). This has been a popular way in the early days of the integration of XML into databases. However, its drawback is that to access the XML contents the XML document has to be (re)parsed. This approach prevents the use of the query optimization facilities of a DBMS. The solution to these problems is shredding. Shredding means that the XML document is parsed only once and the contents are directly exposed to the DBMS, which can thus optimize the access to it. All the methods described in the upcoming sections use a shredding approach.

Semistructured Data

The Magnum Object Algebra (MOA) [BWK98] is an intermediate language between an object calculus, *e. g.* OQL, and the database execution language, *i. e.* MIL. In [vZAW99] the authors investigate an extension to MOA to also handle semistructured

data in the form of XML documents. The tree is represented by a set of binary associations. Each association describes a parent/child combination.

Taking the example parse forest the database stores these associations:

$$
\begin{aligned}
S_S[\rho] &= \{< o_1, \text{“1.00”} >\},\\
S_S/Im &= \{< o_1, o_2 >\},\\
Im/Lo &= \{< o_2, o_3 >, < o_2, o_4 >\},\\
Im/\alpha &= \{< o_2, o_7 >\},\\
\alpha/Co &= \{< o_7, o_8 >\},\\
&\vdots\\
Sk/cdata &= \{< o_{18}, o_{19} >\},\\
Fa/cdata &= \{< o_{20}, o_{21} >\},\\
cdata[string] &= \{< o_5, \text{“http://...”} >, < o_6, \text{“http://...”} >,\\
&\quad < o_{10}, \text{“29053”} >, < o_{12}, \text{“0.03”} >, < o_{14}, \text{“0.19”} >,\\
&\quad < o_{17}, \text{“true”} >, < o_{19}, \text{“00...”} >, < o_{21}, \text{“1”} >\}
\end{aligned}
$$

In this case there is no large overlap in structure, but when instantiations of a parent/child relationship occur distributed over the document they will all end up in the same association, *e. g.* like the $cdata[string]$ BAT.

This mapping provides a good on average query performance, even when used with an off-the-shelf DBMS, as has been benchmarked by [FK99].

Monet XML and XMark

Monet XML has been developed with the parent/child mapping from the MOA approach as starting point. Two basic features distinguish Monet XML [SKWW00, Sch02] from other XML to database tables mappings:

1. the decomposition method is independent of the presence of a *Document Type Definition* (DTD) or other schema, but explores the structure of the document at runtime;

2. it tries to minimize the volume of data to be processed during a query by storing associations according to their context in the tree.

This basically means that database tables are created upon need, and these tables are not only vertically decomposed, but also horizontal. The horizontal decomposition is administered by the path catalog which contains information about the specific context of the associations stored in the table. This leads to this specific database instantiation for the example parse forest:

$$
\begin{aligned}
S_S[\rho] &= \{< o_1, \text{“1.00”} >\},\\
S_S/Im &= \{< o_1, o_2 >\},\\
S_S/Im/Lo &= \{< o_2, o_3 >, < o_2, o_4 >\},\\
S_S/Im/Lo/cdata &= \{< o_3, o_5 >, < o_4, o_6 >\},\\
S_S/Im/Lo/cdata[string] &= \{< o_5, \text{“http://...”} >, < o_6, \text{“http://...”} >\},\\
&\vdots\\
S_S/Im/\alpha/Cl/Fa &= \{< o_{15}, o_{20} >\},\\
S_S/Im/\alpha/Cl/Fa[\rho] &= \{< o_{20}, \text{“0.77”} >\},\\
S_S/Im/\alpha/Cl/Fa/cdata &= \{< o_{20}, o_{21} >\},\\
S_S/Im/\alpha/Cl/Fa/cdata[string] &= \{< o_{21}, \text{“1”} >\}
\end{aligned}
$$

It is clear that this approach uses a larger number of tables due to the use of more context in the distribution of the nodes, *i. e.* several tables contain $cdata[string]$ information. This makes it possible to directly zoom in on a specific part of the XML document by resolving path expressions mainly in the path catalog. On the other hand complete reconstruction of an XML document is more expensive.

The Monet XML project also includes the definition of the XMark benchmark [SWK+01, SWK+02]. This benchmark is used to assess an XML database's abilities to cope with a broad spectrum of different queries, typically posted in real-world application scenarios. It is widely used to assess systems.

XQuery

Based on [Gru02] an XQuery interface on top of Monet is currently under construction [GvKT03]. [GvK03] describes one of the major optimized operations: the staircase join. The optimizations in this implementation make use of a node numbering scheme. Each node is assigned a *pre*- and a *post*-order, *i. e.* resulting in a coordinate for a node in the *pre/post* plane. The staircase join uses extensive knowledge about the distribution of nodes in this plane with respect to a certain context node to prune areas from the search space.

Figure 5.2 shows the parse forest of Figure 4.9 in the pre/post plane. The information about these nodes is stored in a small number of BATs from which each has the unique and dense pre-orders as the head column.

5.2 A Feature Database

Most of the XML facilities for Monet were developed concurrently with Acoi. As such, an Acoi specific mapping had to be defined and implemented. In this section

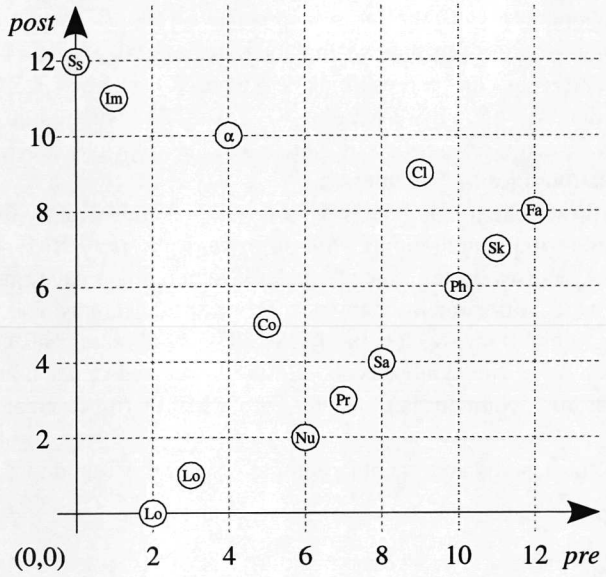

Figure 5.2: The *pre/post* plane.

this mapping is described. It is based on the parent/child mapping from the MOA approach. Care has been taken to keep the interaction, from the FDE and FDS standpoints, purely XML and hence independent of the storage system and mapping. In this case the XML documents are transformed by an XSLT stylesheet [W3C01b] into a MIL script. This script inserts the data from the parse forest into the database.

Notice that this mapping is just a baseline implementation. Other mappings and systems, *i. e.* the discussed mappings for Monet or the XML support of an off-the-shelf DBMS, may prove to be a more effective and efficient XML storage alternative. Due to the XML exchange layer these alternatives can relatively easy replace the current storage backend.

5.2.1 A Database Schema

DTD-based or schema-less XML mappings support only one basic data type: character data (CDATA). However, a feature grammar contains information about the atomic types of the data leafs in the parse forest. To create a database schema which takes advantage of this information, *e. g.* the integer equivalent of $int(29053)$ is cheaper to store than the corresponding character string, the grammar can be translated into a XML document providing schema information. Currently there are several competing XML schema languages. The major ones are: XML Schema [Fal01], Relax NG

[CM01] and Schematron [Jel02]. These languages can be partially intertwined. For example Schematron assertions may be embedded in an XML Schema [XFr01], while XML Schema datatypes can be reused inside Relax NG schemas [CK01]. All these language have their strong and weak points. A favorable combination [vdV01] may look as follows: structures described by Relax NG, data types by XML Schema[1] and additional validation rules by Schematron.

Once more these schema languages were developed concurrently to the Acoi system. In the current implementation a straightforward propriety XML-based schema language is used. However, any other "standard" schema language may replace this language. The schema document contains a list of non-terminals, *i. e.* all the LHSs, and their possible children, *i. e.* all the RHSs. The symbols are all annotated with meta-information, *e. g.* the symbol type, and the lower and upper bound. Using an XSLT stylesheet this document is translated into a MIL script to create the database tables.

This part of the schema document (a complete version is found in Appendix C.1):

```
...
<Image:Color type=".non-terminal.detector.blackbox.">
  <Image:RGB type=".non-terminal." coll="list" lbnd="0"
    hbnd="infinit"/>
  ...
</Image:Color>
<Image:RGB type=".non-terminal.">
  <Image:Red type=".non-terminal."/>
  ...
</Image:RGB>
<Image:Red type=".non-terminal.">
  <fg:int type=".terminal.atom."/>
</Image:Red>
...
```

is translated into these MIL statements:

```
...
VAR Image_Color_Image_RGB_parent := new(void,oid);
VAR Image_Color_Image_RGB_child  := new(void,oid);
VAR Image_RGB_Image_Red_parent   := new(void,oid);
VAR Image_RGB_Image_Red_child    := new(void,oid);
VAR Image_Red_fg_int_parent      := new(void,oid);
VAR Image_Red_fg_int_child       := new(void,oid);
VAR fg_int                       := new(oid,int);
...
```

[1] The data types of XML Schema lack a decent type system[Lew02], however, at least it provides an extension to the limited set of DTD data types.

The BATs with *void* head and *oid* tail will store the tree structure of the parse forest. Data from leaf nodes are stored in specific BATs which contain a tail column of the atomic type.

The *void* head produces a dense numbering scheme for a specific edge type, resulting in aligned array access of all the base and meta-data associated to the edge. This meta-data is stored in additional BATs. For example this BAT stores the position of a *RGB* instance in a specific list:

```
1 VAR Image_Color_Image_RGB_list := new(void,int);
```

Other examples of needed meta-data are the context and scope of the nodes, the version and confidence of detectors and their input relations.

Next to these data BATs also information about the feature grammar system is stored. This enables the use of generic procedures which follow the dependencies between the various nodes, *e. g.* to reconstruct the original XML document.

5.2.2 A parse forest XML document

The FDE contains in memory a parse forest in the form of an XML document. This internal document contains more meta-data than needs to be stored in the database. Using an XSLT script this internal format is stripped down. Appendix C.2 contains an example of the final parse forest XML document. Some portions of this document will be described in this section.

```
1  <?xml version="1.0"?>
2  <fg:forest
3     xmlns:fg="http://www.cwi.nl/~acoi/fg/forest"
4     xmlns:WWW="http://www.cwi.nl/~acoi/WWW"
5     xmlns:Image="http://www.cwi.nl/~acoi/Image"
6  >
7     <fg:elementary context="1:1" confidence="1.00" idrefs="2@1"
8        start="WWW:WebObject" date="20030625"
9     >
10 ...
11    </fg:elementary>
12    <fg:auxiliary date="20030625">
13 ...
14    </fg:auxiliary>
15    <fg:auxiliary date="20030625">
16 ...
17    </fg:auxiliary>
18 </fg:forest>
```

The root of the document $fg : forest$ contains information about the feature grammar modules used, *i. e.* they are mapped to XML namespaces. The root contains at least one $fg : elementary$ child node and zero or more $fg : auxiliary$ child nodes.

```
...
   <fg:elementary context="1:1" confidence="1.00" idrefs="2@1"
      start="WWW:WebObject" date="20030625"
   >
      <WWW:WebObject id="5478@0" context="1:1">
         <WWW:Location id="1" context="1:1">
            <WWW:url id="2" context="1:1">
               <![CDATA[http://...]]>
            </WWW:url>
         </WWW:Location>
         <WWW:WebHeader idrefs="5479@0" context="1:1"/>
         <WWW:WebBody id="7" context="1:1">
            <Image:Image id="8" context="1:1">
               <Image:Color idrefs="5480@0" context="1:1"/>
               <Image:Class id="15" context="1:1">
                  <Image:Photo idrefs="5486@0" context="1:1"/>
                  <Image:Skin idrefs="5487@0" context="1:1"/>
                  <Image:Faces idrefs="5488@0" context="1:1"/>
               </Image:Class>
            </Image:Image>
         </WWW:WebBody>
      </WWW:WebObject>
   </fg:elementary>
...
```

Each parse forest is based on one start symbol, which roots the elementary trees. To these elementary trees auxiliary trees, which are rooted by detectors or references, may be attached. The *idrefs* attributes of inner nodes, *i. e.* a quasi-root, refer to specific instantiations of the auxiliary trees, *i. e.* the quasi-foot nodes. When there is more than one reference the node is ambiguous and each *idref* will point to a detector call for a different context. The *idrefs* attribute of *fg : elementary* refer to the initial tokens.

```
...
   <fg:auxiliary date="20030625">
      <Image:Color id="5480@0" idrefs="2" context="1:1"
         confidence="1.00" version="1.0.0"
      >
         <Image:Number id="9" context="1:1">
            <fg:int id="10" context="1:1">
               <![CDATA[29053]]>
            </fg:int>
         </Image:Number>
         <Image:Prevalent id="11" context="1:1">
            <fg:flt id="12" context="1:1">
               <![CDATA[0.03]]>
            </fg:flt>
```

```
15          </Image:Prevalent>
16          <Image:Saturation id="13" context="1:1">
17             <fg:flt id="14" context="1:1">
18                <![CDATA[0.19]]>
19             </fg:flt>
20          </Image:Saturation>
21       </Image:Color>
22    </fg:auxiliary>
23 ...
```

Auxiliary trees contain the output of a detector call or are placeholders for a reference to an elementary tree.

Notice that only root nodes contain an *id* with a @0 prefix, which indicates that it is database unique. The *id* attribute of an inner node is just a normal integer and needs to be turned into a database unique identifier upon insertion into the database. This minimizes the need for the FDE to request unique identifiers from the database when a node is added to the tree.

5.2.3 Inserting a Parse Forest

The insertion script for Monet is generated just as the schema script: by an XSLT stylesheet. For the example auxiliary tree these MIL statements are generated:

```
1  ...
2  Image_Color_idrefs.insert(id2oid("5480@0"),id2oid("2"));
3  Image_Color_context.insert(id2oid("5480@0"),context("1:1"));
4  Image_Color_confidence.insert(id2oid("5480@0"),flt("1.00"));
5  Image_Color_version.insert(id2oid("5480@0"),version("1.0.0"));
6  Image_Color_Image_Number_parent.insert(id2oid("5480@0"));
7  Image_Color_Image_Number_child.insert(id2oid("9"));
8  Image_Number_context.insert(id2oid("9"),context("1:1"));
9  Image_Number_fg_int_parent.insert(id2oid("9"));
10 Image_Number_fg_int_child.insert(id2oid("10"));
11 fg_int.insert(id2oid("10"),int("29053"));
12 fg_int_context.insert(id2oid("10"),context("1:1"));
13 ...
```

5.2.4 Replacing a (Partial) Parse Forest

The roots of the elementary and auxiliary (partial) parse forests contain database unique identifiers. Those are used to check if the forest is already stored in the database. If this is true the new forest will replace the old one. As this new forest may have a complete new shape and thus not neatly replace the old forest, the old forest is deleted from the database before the new forest is inserted. To support this a stored procedure, generated by XSLT from the schema document, is called. This

procedure knows which BATs are involved with this specific type op (partial) parse forests and pointer chases the specific forest or, in the case of a bulk operation, forests.

5.2.5 Query Facilities

As MIL is still the primary means to interact with Monet (the SQL and XQuery interfaces are still under development) an Acoi specific query interface has been developed. Once more this interface is based on the combination of an XML document and an XSLT style sheet.

The XML document contains zero or more selection trees and one projection tree. In the selection trees predicates on terminal values are specified. More than one selection tree is needed when the predicates are disjunctive or there are several conjunctive predicates on the same terminal. In the projection tree the nodes the user wants to be part of the answer XML document are marked. This query document requests all portraits from the database.

```
<?xml version="1.0"?>
<fg:query
    xmlns:fg="http://www.cwi.nl/~acoi/fg/query"
    xmlns:WWW="http://www.cwi.nl/~acoi/WWW"
    xmlns:Image="http://www.cwi.nl/~acoi/Image"
    grammar="video" start="WWW:WebObject"
>
  <fg:select>
    <WWW:WebObject>
      <WWW:WebBody>
        <Image:Image>
          <Image:Class>
            <Image:Faces>
              <fg:int min="1" max="1"/>
            </Image:Faces>
          </Image:Class>
        </Image:Image>
      </WWW:WebBody>
    </WWW:WebObject>
  </fg:select>
  <fg:project>
    <WWW:WebObject>
      <WWW:Location project="true">
        <WWW:url project="true"/>
      </WWW:Location>
    </WWW:WebObject>
  </fg:project>
</fg:query>
```

The XSLT sheet translates this query document into a MIL script which starts with

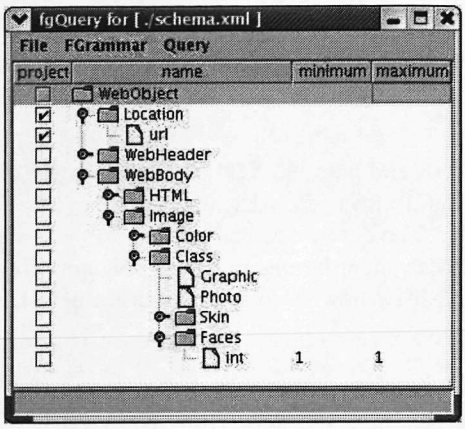

Figure 5.3: The query interface.

the predicates and traverses up the selection tree. After all selected trees are collected a traversal down the projection tree for each selected tree is started and each requested projection is printed. Some special measures are needed to check the contextual validity, *i. e.* a conjunction is only valid within the same context.

As query documents get quite verbose a simple *graphical user interface* GUI shows the tree derived from a specific schema document (see Figure 5.3). Using this tree control projection nodes can be marked and simple predicates can be defined. The query can then be stored as an XML document or directly be executed.

5.2.6 Adding Database Management to a Database Kernel

Monet is a database kernel, which means that it only provides the kernel primitives for a full fledged DBMS. The previous sections described how a feature grammar specific XML front-end was build on top of this kernel, however, there are still some key components lacking. To get a reasonable data throughput for a web crawler concurrent updates of the database are needed. BATs are by default not locked on read or write access, *i. e.* locking is left to the application programmer. The default extension modules offer the *fork* command and the *lock* atom type as building blocks for concurrency and a transaction mechanism. Using these a simple transaction system on the MIL level was realized, thus allowing concurrent access.

To allow asynchronous communication between the Acoi tools and the Monet backend a queuing mechanism was added. This enables a FDE to put its XML insertion request into the queue and request the next instructions from the FDS.

The bottom-line was achieved when all queries spend the major part of their idle time in waiting on the non reentrant MIL parser. The next major version of Monet will

contain a reentrant MIL parser.

5.3 Discussion

This chapter described several possible XML storage schemes. This type of research has been developing rapidly over the past few years (see [Bou03] for an extensive overview of mappings). The current, rather ad-hoc, implementation used by Acoi is just a bottom line. Replacement by one of those, concurrently developed or newer, schemes, *e. g.* Monet XML, is now more of an engineering task than a research topic.