X2S

XPath query processing for relational databases

Diplomarbeit

Am Institut für Programmstrukturen und Datenorganisation (IPD)
Fakultät für Informatik der Universität Karlsruhe
und FZI Forschungszentrum Informatik

Tom Gelhausen

Juli 2002

Verantwortlich:	Prof. Dr. Peter C. Lockemann
Betreuer:	Aleksei Valikov
	Wassilios Kazakos

Abstract

This thesis shows a way to represent relational database content as a virtual XML document and how to process queries in natural XML access language (XPath) against this data. Design principals were both, efficiency and minimal system requirements to support also relational database systems without any type of build-in XML support. This is a read-only approach so far.

The supported XPath features are

- NodeTest: NameTest and NodeType
- 8 axes (**`ancestor`**, **`ancestor-or-self`**, **`attribute`**, **`child`**, **`descendant`**, **`descendant-or-self`**, **`parent`**, **`self`**, unrolling is limited to mapping definitions depth)
- relative navigation (starting from context node)
- Constants: Numbers and Literals (Strings)
- Operations: in principle all standard XPath 1.0 operations (additive (+, -), multiplicative (*, div, mod), and, or, equality (=, ≠), relational (>, ≥, <, ≤), union, sign)

Table of Contents

Table of Figures

Table of Abbreviations

DOM	Document Object Model, see [W3C98b]
DBMS	Database Management System
PTG	Path Tracing Graph, see Section 5.7
RDBMS	Relational Database Management System
SAX	Simple API for XML Processing
URL	Uniform Resource Locator
W3C	World Wide Web Consortium, see http://www.w3c.org
XDR	XML-Data Reduced Schema
XML	Extensible Markup Language, see [W3C98a]
XSD	XML Schema Definition

Preface

This diploma thesis emerged from an 'it-would-be-nice-to-have' idea. Starting to search for an appropriate tool, we soon figured out that no existing[1] tool fits exactly our wishes. Indeed, a lot of approaches existed to bridge the gap between XML and relational databases. In each of them were lacks of features for our special requirements. So, we started the research for this thesis.

Not only the results evolved through the time of working, but also the problem of the postulation of requirements has been clarified increasingly. Task and solution were improved together with the growth of understanding the problem. A tool emerged that provides a unique combination of purpose and approaches up to date[2].

[1] November 2001
[2] July 2002

1 Introduction

Due to the simple and nonetheless powerful structure and the good data exchangeability between distributed systems, the Extensible Markup Language (XML) has become more and more popular as a technology for structured and semi-structured representation of information. The well known advantages of this technique lead to a steadily growing number of IT solutions using XML to store, to exchange, or to transform data and even to communicate through XML based protocols like SOAP.

Meanwhile, there exist long evolved applications which are build on top of old database systems. Some of these systems have very limited support for XML, others may have a lack of support for this technology at all.

Currently a lot of different approaches try to connect relational databases and the world of XML. Each of them has a slightly different attitude on how to do that. Ours is to provide a direct mapping (a "view" in database theory terms) from XML to relational data, so that the resulting *virtual XML document* can be queried just like a normal XML document using the XPath query language.[3]

1.1 *Motivation*

The idea is to query a relational database by XPath to allow modern XML based components to access existing data in relational database systems, even if they have none or only limited support for XML. It seems adjacent to choose XPath, since it is a very popular query language for XML documents and other query languages for semi-structured data like X-QL or XQuery (see [ChF+01]) are based on it.

Another way to obtain relational data within an XML solution is to use embedded SQL commands, for instance. However, this possibility forces the developer to switch between the notions of the relational data model, the relational schema and the semi-structured document model concepts and the schema of his XML component. The thus created module depends not only on the XML design but also on the relational systems layout. Exchanging the underlying relational database against a native XML database later on becomes unnecessarily difficult, for instance.

[3] XPath is a language for addressing parts of an XML document. It is standardized by the W3C (see http://www.w3.org/TR/xpath) and designed to be used in XSLT and XPointer, two more standards of the W3C. Even though we give a very short introduction to what we need, we expect the reader of this document to be handy with the relational model, the semi-structured model of XML data, the query language XPath, and SQL.

A platform independent middle tier translating XPath queries into SQL queries and returning the results in XML form would allow XML based components to access the relational data in document schema terms instead in relational database schema terms, without the need to install a new RDBMS with some kind of XML support build in. The solution should just rely on the similarities (SQL-92 entry level) of the common relational database systems, thus run on the major commercial database servers and on most freely available systems, too.

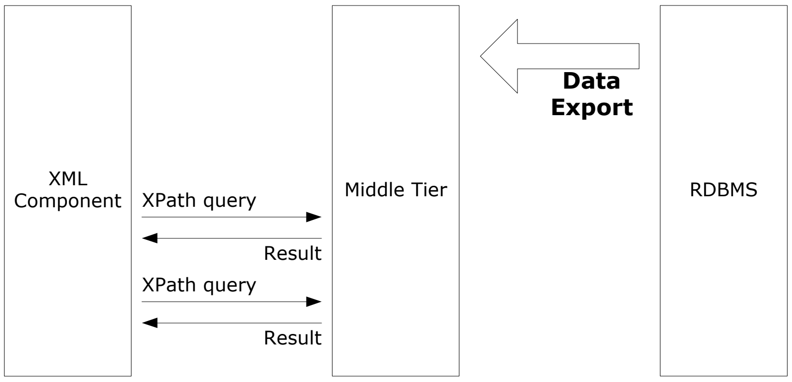


Figure 1-1 *Naïve Approach*

A pretty simple and straight forward solution to allow XPath processing on relational database data would be to export all data from the database into an XML document and then use an existing tool to evaluate XPath expressions against it as depicted in Figure 1-1. For a database of several terabytes in size this is obviously impossible. Assuming we only want to get the name attribute of an element with a given ID, the naïve approach drags substantially more data to the client computer to just discard it right after instead of only obtaining what is needed. So, even for a system containing very few data, this procedure would create a tremendous overhead in comprehension to the amount of data typically selected by an XPath expression.

1.2 *An Example*

Say we have a database containing three tables **COMPANY**, **DIVISION**, and **EMPLOYEE**. Each company, each division, and each employee are identified by an ID and a name attribute. Each division belongs to one company and each employee belongs to one division. Further on, each employee might report to an other employee (see Figure 1-2).

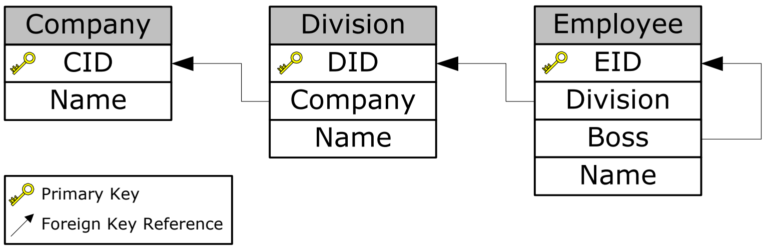


Figure 1-2 *An example database schema*

The relational model expresses this kind of containment ('belongs to') by foreign key constraints and join condition in SQL queries. An XML document, on the other hand, can even better express such containments, namely through nested elements.

Within an XML solution, we would want to query the database in the form **`/company/@name`** to obtain the name of a company. Likewise **`/company[@name='FZI']/division`** should select all divisions of the company named FZI. Of course, the meaningful queries (those that potentially could return a result) depend on the structure of the virtual document. A possibility for the relational content as an XML document is shown in Figure 1-3.

```
<company id="0" name="FZI">
    <division id="0" name="DBS">
        <employee id="0" name="Lockemann">
            <employee id="1" name="Kazakos">
                <employee id="2" name="Valikov"/>
            </employee>
        </employee>
    </division>
    <division id="1" name="Prost"/>
</company>
```

Figure 1-3 *An XML document representing relational database content*

1.3 *Scopes*

The goal of this thesis is to find an efficient way to transform an XPath query into a query that a relational database system can process. Since most of the existing relational database systems support SQL-92 entry level as query language, in order to support a wide range of platforms, an XPath query is translated into one or several equivalent SQL queries. Since XPath and SQL are *declarative query languages* (the user states *what* to return and *not how* to obtain), the main task is to close the gap (see Section 3.1) between the relational model and the tree-like XML data model.

For performance reasons, as much of the result computation as possible should be made within the relational query engine. Contrariwise, as few as possible distinct SQL statements should be created to allow the relational query optimizer to work efficiently and to reduce the connection overhead.

A view shall be provided to allow natural XPath querying of the relational content. This view shall be configurable to allow user customization. To avoid the drawback of 'deep-materialized' result tree fragments (see Section 3.3), the solution ought to provide an object model allowing the access to the whole virtual document requiring only a very limited portion of it to be materialized. Besides the matter of scalability, this enables relative queries and thus user interactivity.

1.4 *Overview*

Chapter 2 will give an overview of the concepts of XML, XPath, and relational databases as far as we need them for this thesis. As we expect the reader of this thesis not to be a newbie to these topics, this chapter might be regarded as recall of previous knowledge and as clarification of the terms that will be used further on in this thesis.

The conceptual considerations of Chapter 3 reveals some basic phenomenon, that every approach must have been taking (or at least should have been taking) into account before the design started.

The next step is to find out how current approaches meet our requirements, this can be found in Chapter 4. We will look upon some solutions that come from research groups as well as those that are available from the "big three" (Oracle, Microsoft, and IBM) in their relational database products.

The main chapter of this thesis, Chapter 5, deals with the proposed solution for the task. First, the abstract data types, or in other words "the object model", will be introduced. Since the task is to find an algorithm, it is necessary to explain the model, its terms, and its functions provided to build the complex algorithm upon. Of course, the abstraction level stays high enough to be implementation independent.

This thesis is closed by the Chapter 6, "Conclusion and ".

2 Basics

This chapter gives a quick introduction to the terms and conventions we will use in the following. It is not intended to teach SQL, XML, or XPath. For this purpose, we would advise to refer to the literature mentioned in the corresponding section.

2.1 *Relational Databases*

The interface between this approach and the RDBMS's it is expected to work on, is SQL. Consequently we can not reflect on the mathematically accurately defined relational data model itself but what SQL presents us thereof. So, when talking of 'the relational model' the view onto it provided by SQL is referred to.

Basically, the relational model stores *tuples*, sometimes also referred to as *rows*. These tuples consists of *values* from different *domains*. Informally spoken, a domain is a data type in conjunction with an interpretation of the value, in example an integer representing the index number of an element or the age (in years) of a car. The domains are also referred to as *columns*.

The total amount of tuples of one type (unique combination of domains) is stored in a *relation*, often referred to as *table*. Per se, a table is not a mathematical set[4] and may contain duplicates. Duplicates are two or more tuples containing the same values for each domain.

Duplicates in tables may be undesired. For this case, most RDBMS's provide *constraints*. Constraints are conditions that always have to be fulfilled by every single row of the table. So if we require the tuples to contain a unique value for a certain domain (or a unique combination of values for a certain combination of domains), the set can not contain duplicates any more, since the compliance of constraints is always ensured by the DBMS. This is called *key constraint*, the corresponding domain is often called *the key* of the relation, or *the key column* of a table. For if multiple domains have been inflicted a key constraint, we call it *combined key*.

Another common constraint is the *foreign key constraint*. It demands, that the tuples of a table only contain values in a certain domain which also occur in the key column of another table. This is also valid for combined keys, of course.

[4] not in the SQL view of the relational data model, see [LaLo95]

For further details on this topic, refer to [LaLo95]. For details on SQL we would recommend [Date93] or [GrWe94].

2.2 *XML*

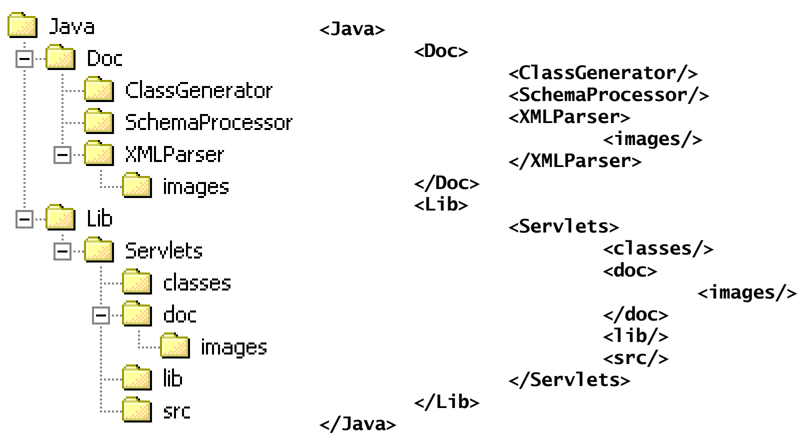
XML is a method to present (semi-)structured information (*documents*) as text. This text is not intended to be read by a user but to be processable (and debugable by a developer with the aid of a simple text editor, in time of need) in a platform independent manner. More precisely, XML provides a syntax to markup documents in a way that is independent of any further way of processing or presenting.

Two terms that need to be distinguished are the *document instance* (or only *document*) and the *document type* (or its *schema*). The document type describes the structure of document instances. These instances' content is independent among them, only their assemblies are equal.

XML documents are build from *elements* and their *attributes* (and some other constructs we will not need in the following). Elements can be *nested*, that is to say an element may have other child elements. The structure is thus recursive.

Usually, recursive structures are presented as a tree. Accordingly, the W3C who 'recommended' (their term for 'adoption') XML (see [W3C98a]) introduced the *Document Object Model* (DOM) (see [W3C98b]). This is a set of interfaces allowing tree-like navigation through the *nodes* (both, elements and attributes are considered as nodes) of an XML document and to access their values via the non-complex data type string.

For further details on XML, refer to [KaST02], [HaMe01], or [Brad01].



```
<Java>
        <Doc>
                <ClassGenerator/>
                <SchemaProcessor/>
                <XMLParser>
                        <images/>
                </XMLParser>
        </Doc>
        <Lib>
                <Servlets>
                        <classes/>
                        <doc>
                                <images/>
                        </doc>
                        <lib/>
                        <src/>
                </Servlets>
        </Lib>
</Java>
```

Figure 2-1 *An example illustrating the comparableness of directory tree and XML tree*

2.3 *XPath*

XPath has been invented to select parts of an XML document. According to the path expressions in Unix which navigate through the directory tree, XPath expressions navigate through the tree formed by the elements of an XML document. An example illustrating the comparableness of a directory tree and the document tree of an XML file can be seen in Figure 2-1. An expression like `/Lib/Servlets/src` would select the very last node of that example, as well in XPath as in Unix directory expression syntax. (Indeed, the Unix directory expression and the XPath query look exactly alike in this case.)

A *location path*, the XPath term for an expression like the example from the previous paragraph, consists of multiple *steps*. In this case the steps are `Lib`, `Servlet`, and `src`. Each step selects a set of nodes from the XML document and passes them to the next step. For this next step, they form the context the processing of the next result set is starting from. So the first step, `Lib`, selects all nodes with the name 'Lib', the second step selects all its children with the name 'Servlet' (there is only one in this example), and so on.

Two additional features, Unix path expressions do not have, are *predicates* and *axes*. Predicates are conditions which are inflicted on certain steps. Only nodes are put into the result set that evaluate these conditions to true. The usage of predicates is optional. But also multiple predicates may be appended to a step – each in squared brackets right before the next slash ('/'). The condition itself is an XPath expression (recursiveness!) whose result is interpreted as Boolean value.

The concept of axes extends the functionality known of '.' and '..' in Unix path expressions. The single dot '.' selects the directory itself, the double dot '..' the parent directory. Likewise, the `self` axis selects the node itself, the `parent` axis selects the parent node. Further on, there exist several other axes as shown in Figure 2-2.

The axis is an integral part of every step. They were just invisible in the above example, since it made use of a legal abbreviation: the `child` axis is default and can thus be omitted. In all other cases, the axis needs to be defined, either through its name, followed by two colons ('::') in front of the step, or indirectly through an abbreviation ('.', '..', and '@' for the `attribute` axis are also allowed).

Six axes are not mentioned in Figure 2-2. These are `attribute`, `following`, `following-sibling`, `previous`, `previous-sibling`, and `namespace`. The `attribute` axis simple selects the set of attributes of the context element, the other five axes are not supported within this approach and will thus not be introduced here.

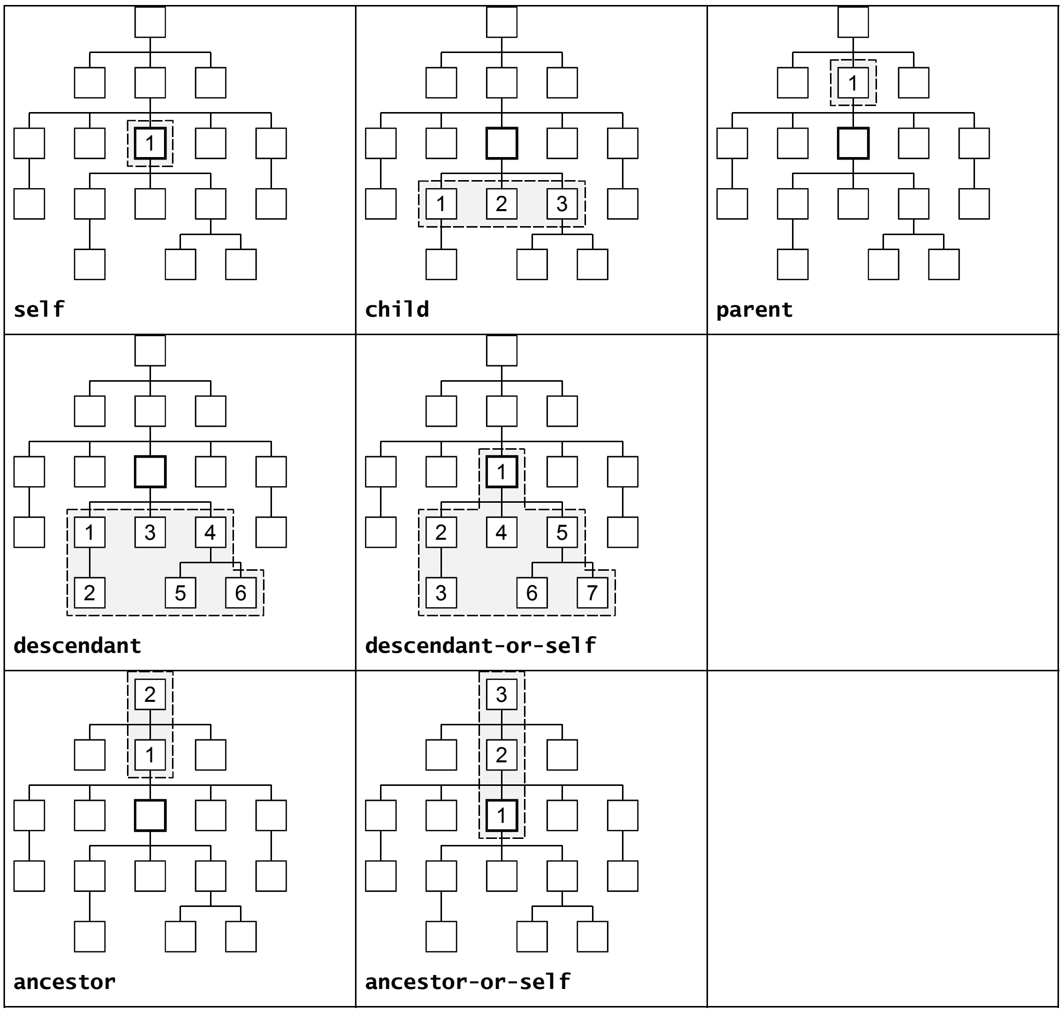


Figure 2-2 *Visualization of the axes*

The axis provides a kind of preselection of possible result nodes of the corresponding step. All nodes that lie on the axis are candidates. All candidates that match the *name test* (‘Lib’, ‘Servlet’ and ‘src’ for the above example) and the optional predicates of the step are added to the result set. The last step of a location path returns the result nodes of the complete query.

[KaST02] gives a short introduction to XPath including some constructs missing here. Alternatively, [Kay01] may be used as very detailed reference also containing a lot of examples and hints. The pictures in this section have been taken from [Vali02] with kind permission of the author.

3 Conceptual Considerations

Before we compare existing solutions with what we demand, we will take a quick look onto some peculiarities, every approach has to be regarded. These due to the incompatibilities of the relational and the semi-structured data model, and to the requirements claimed by a user of such a solution.

3.1 *Paradigm gaps*

The main problem in the process of translating XPath queries into SQL queries is due to the paradigm gap between XML's tree-like document structure in contradiction to the flat set of collections structure of relational data. This section provides a quick survey on them.

3.1.1 Recursive Structure of XML documents

In an XML document, the different elements are recursively nested. The level of nesting is unlimited. XPath reflects this property by providing mechanisms to deep search the document tree. The **`descendant-or-self`** axis for instance provides such a function.

On the other hand, most current database management systems comply only with the entry level of the SQL-92 standard. Neither the standard nor the relational model itself does permit recursive queries in any way. Different suppliers offer non-standard extensions to provide support for recursive queries. But because of the postulated platform independency, our approach can not take them into account.

Apart from that, the database misses meta information that may be contained in an XML document according to the explanations given in section 1.2 pertaining the representation. Of course, this metadata must be provided as well to the view as to a translator.

The metadata declaring the mapping from the relational structure to the structure of the virtual document is called *Setup* in this approach. For further details see section 5.4, we just want to introduce the term here.

3.1.2 Order of elements in an XML document

In an XML document, all elements are strictly ordered. This becomes quite obvious when thinking of another representative from the SGML family, HTML. The order of the paragraphs and headline elements is of great importance. Due to their document centric design, XML documents fully comply with this property. The `following` axis for instance selects all the document content, from the cursor up to the end.

The relational model again has no correspondence to this feature. Nevertheless, SQL provides ordering via the `ORDER BY` clause. This allows us to define an order within equal element types of our virtual document, based on the values of one or more of their attributes. However, a mixed structure, something like alternating paragraphs and headlines in HTML documents, will not be possible.

Due to the nature of a relational database, the maximum order we can achieve in a virtual XML document is

a) **Structurally ordered** (all elements of one type are either before or after all elements of an other type, i.e. AABBCC, not ACBABC): $\forall c_1,c_2,c_3 \in C : t(c_1)=t(c_2) \wedge t(c_1)\neq t(c_3) \rightarrow (p(c_1)>p(c_3) \wedge p(c_2)>p(c_3)) \vee (p(c_1)<p(c_3) \wedge p(c_2)<p(c_3))$, and

b) **Internally ordered** (the order of elements of one type is determined on the basis of one or several of their attributes): $\forall c_1,c_2 \in C : t(c_1)=t(c_2) \wedge v(c_1.a)>v(c_2.a) \rightarrow p(c_1)>p(c_2)$

whereas C is the set of child elements of a node, t(x) is the element type of x, p(x) is the position of x in the virtual document, v(x) is the value of x, and x.a is the attribute selected as sort criteria.

The structural order is relatively easy to fulfill. It just requires that items of one type all are processed at the same time and before or after items of other types. This should normally be ensured automatically by any naïve approach to obtain data from different sources.

A deterministic internal order may be simulated by `ORDER BY` clauses in the translated queries. Thus, if internal ordering is required, hints on how to totally order every tables elements have to be provided. These hints must contain information on every columns priority and sorting direction.[5]

Since this approach automatically ensures structural order as described above and neither supports the `preceding`, `preceding-sibling`, `following`, and `following-`

[5] We can weaken this demand to "every key columns priority and sorting direction" if no special order is required.

`sibling` axes, nor any XPath functions like `position()` or `last()` for now, we will not reflect on ordering in the realization part.

3.1.3 Navigation and Context Nodes

XPath queries support context nodes as origin for relative queries. Having obtained a single node, we can still access the rest of the document by relative queries, starting from the result node.[6] This is possible, because every node has a certain identity that is independent of any attribute values. A node may have a unique ID attribute or it might have been assigned such an attribute internally, but the point is that it has *conceptually* an identity, no matter how it is expressed (or implemented).

In contradiction to the theoretical model, where relations are mathematical sets, SQL treats them as collections that may contain duplicates (see [LaLo95]). Consequently a tuple in SQL does not have an identity like a node in an XML document. It may fulfill a constraint, the uniqueness of some attributes values within its collection (key constraint), though. Providing the values of this attributes in an SQL statement enables us to select one single, 'unique' node from the according set, but only as long as the underlying RDBMS ensures this constraint. An approach must be able to map the uniqueness of XML elements to relational data to support relative queries and thus navigation.

3.2 *Choosing the Mapping*

There are several possibilities to map relational data into an XML document. Some are presented in more detail in Chapter 4. At this point, only two extreme paradigms are presented. These are

a) **The simulation** of the relational structure with XML elements (Figure 3-1), using the XML tags only as delimiters for tables, rows, and columns. The simulation does not contain metadata that goes beyond what the database system probably knows about the data. The structure is flat, the minimal and maximal element nesting level is known to be 3.

b) **The representation** of the data in terms of XML without any meta information on the original table and column structure (Figure 1-3).

[6] A query like `./ancestor-or-self::*/descendant-or-self::*` would select all nodes within the current document starting from any arbitrary node of it.

Instead, the representation is structured semantically and thus contains metadata that would only reside in the application logic otherwise.

Of course, any granularity of realizations in between those both paradigms is possible. Perhaps one could think of the simulation combined with nested tables or its row elements containing the columns as attributes. But this here is just to point up the difference between both approaches.

```
<table name="COMPANY">
     <row>
          <column name="CID" datatype="int" primarykey="true">0</column>
          <column name="Name" datatype="varchar">FZI</column>
     </row>
</table>
<table name="DIVISION">
     <row>
          <column name="DID" datatype="int" primarykey="true">0</column>
          <column name="Company" datatype="int" keyreftab="COMPANY"
               keyrefcol='CID'>0</column>
          <column name="Name" datatype="varchar">DBS</column>
     </row>
     <row>
          <column name="DID" datatype="int" primarykey="true">1</column>
          <column name="Company" datatype="int" keyreftab="COMPANY"
               keyrefcol='CID'>0</column>
          <column name="Name" datatype="varchar">Prost</column>
     </row>
</table>
<table name="EMPLOYEE">
     <row>
          <column name="EID" datatype="int" primarykey="true">0</column>
          <column name="Division" datatype="int" keyreftab="DIVISION"
               keyrefcol='DID'>0</column>
          <column name="Boss" datatype="int" keyreftab="EMPLOYEE"
               keyrefcol='EID'></column>
          <column name="Name" datatype="varchar">Lockemann</column>
     </row>
     <row>
          <column name="EID" datatype="int" primarykey="true">1</column>
          <column name="Division" datatype="int" keyreftab="DIVISION"
               keyrefcol='DID'>0</column>
          <column name="Boss" datatype="int" keyreftab="EMPLOYEE"
               keyrefcol='EID'></column>
          <column name="Name" datatype="varchar">Kazakos</column>
     </row>
     <row>
          <column name="EID" datatype="int" primarykey="true">2</column>
          <column name="Division" datatype="int" keyreftab="DIVISION"
               keyrefcol='DID'>0</column>
          <column name="Boss" datatype="int" keyreftab="EMPLOYEE"
               keyrefcol='EID'></column>
          <column name="Name" datatype="varchar">Valikov</column>
     </row>
</table>
```

Figure 3-1 *An XML document simulating relational database content*

According to the above example a query for a company's name that has a division named 'DBS' would look like Figure 3-2 as XPath expression for the first mapping and like Figure 3-3 for the second one. Even though the query in Figure 3-2 is structurally much closer to the desired SQL query (Figure 3-4) and thus probably simpler to translate, probably any user would prefer to use the query from Figure 3-3. It is much easier to understand, hence easier to create and to debug since it is by far closer to a real XPath expressions over 'normal' XML documents.

```
/table[@name='COMPANY']/row[column[@name='CID']=/table[@name='DIVISION']/row[col
umn[@name='Name']='DBS']/column[@name='Company']]/column[@name='Name']
```

Figure 3-2 *Example XPath query for a simulating document*

```
/company[division/@name='DBS']/@name
```

Figure 3-3 *Example XPath query for a representing document*

```
SELECT c.Name
FROM COMPANY c, DIVISION d
WHERE d.Company=c.CID
AND d.Name='DBS'
```

Figure 3-4 *Translated SQL query*

For this reason *the representation* has been chosen as the desired structure of the XML view onto the relational world. Consequently tables are represented as elements and their columns as their attributes. Node types other than elements and attributes do not occur in this view. Since attributes need to be atomic types, we also expect the columns to contain values of atomic types. This is a deliberate restriction for the simplicity of the model.

3.3 *Returned Results*

All approaches that will be investigated in chapter 4 have one thing in common: they return a textual XML representation of the queries results. This is a main difference between them and our new approach. We propose instead to return a proxy-object representation as specified in Section 5.1 for several reasons. The advantages of a virtual approach in contradiction to the materialization of the results embrace:

- Probably the results will not be stored as XML files. But further processing requires an additional parsing step and building of an object tree to reverse the previous serialization.
- Serialization may become a problem for huge result tree fragments (i.e. if the root node is part of it). To only serialize parts of it is arbitrary, since perspicuous borders can not be found.
- Support for relative queries can only be realized manually by the aid of attributes, known to be unique. Further on, multiple occurrences of elements are absolutely prohibited to be able to find a node(-type) again. But having one element occur more than once may have some advantages in searching or iterating through them (see section 5.2).
- If serialization is required, it can still be done afterwards. The result object model just has to provide basic navigation functions for the naïve

approach. Also other, optimized approaches for this task may be utilized without interfering with the rest of this thesis.

- Consistency is easier to achieve (see section 3.4).

3.4 *Consistency*

Consistency is an essential feature of all kinds of database systems. So even if this is not the main theme of this thesis, at least the aspect of the transactional borders is worth being considered.

Though we only consider read operations here, we need transactions, since even if we only read, dirty reads, non-repeatable reads, or phantom reads are still possible. We want to be able to leave the decision to the user, what he wants to allow.

Unfortunately XML solutions do not provide transaction support per se (indeed, normal XML solutions do not need to), so we have to find a way to circumvent this drawback. Since we do not want to extend the standards with proprietary constructs, we have to handle them internally.

According to a basic SQL statement, which in fact is an atomic transaction, we could expect one XPath query to also react like an atomic transaction, disregarding how many sub queries our query processor really has to evaluate against the relational database system. Thus we must start a transaction before putting a first SQL sub query and commit it after the last SQL sub query that belongs to one XPath query. The decision whether or not dirty, non-repeatable, or phantom reads are allowed within this transaction, can then be left to the user configuring the underlying RDBMS.

The next thing we need to take into account is the result of a query. Ignoring that it must be any kind of set with one arbitrary number of elements for the moment[7], we need to pay attention to the possible types of those elements. Such an element might be like a pointer into the database system[8], pointing to the information item it represents. Alternatively, it could be the root of a materialized copy of the whole result fragment tree. Besides performance and system requirements, consistency is an interesting issue here.

[7] This just multiplies the problem, but does not influence the problem itself.

[8] To be platform independent, 'pointer' refers to a **`SELECT data FROM table WHERE keycolumn=value`** statement, thus the pointer consists of a table name, one (or more for combined keys) key columns and as many values to assign to these columns to uniquely select a row, and a column name selecting the specific data within that row.

It is obvious that it is much more simple to keep the information consistent, if there is only one pointer to the information item which is evaluated every time the value is accessed and no copies have to be managed. But even in this case, consistency might become a problem! Though we easily can find out if the information item itself still exists in the database[9], we do not know whether or not the virtual node representing that information still exists by simply following that pointer. Following it can lead us to a node that either might have moved to a different position in the virtual document, possibly not fulfilling its original XPath query conditions any more. It may also lead us to a node that does not even exist any more in the virtual document.[10]

If a virtual node is likely to move or to be deleted before or while being processed, there are two workarounds: (a) we could expand the transaction borders to include the processing after the querying, or (b) we could save extra information along with the pointer, at least describing the structural position or better the whole XPath query leading to this node.

The first method has definitely lower costs to fetch the information item itself (probably a simple SELECT * FROM table WHERE rowPointerCondition), while the second one might require to reapply a complex query containing multiple table joins. The disadvantage of the first method is the costs of a longer transaction, especially if it demands serializability. Thus a trade off has to be found between the costs of a longer transaction and the costs of a re-execution of (parts of) the query. If the result nodes are directly processed and afterwards discarded, method (a) should certainly be preferred. If the result nodes are to be stored and processed later, method (b) should be better. To decide which method to choose, a timer could be implemented that is reset by any access to an element (**`getParent()`**, **`getChildren()`**, **`getValue()`**, etc.) and that closes the transaction once the timer has expired. After the transaction has been closed, an access to the element must use method (b) to ensure consistency. Then again, the transaction could be held open until the timer expires.

Now that we recognized that it is not too easy to keep one single information item consistent, we can allow ourselves to assert that it does not make sense to keep a whole in-memory copy of a document fragment consistent with the database. The only thing we can demand here is that at least the 'snapshot' is a consistent image of the underlying database. Thus the procedure of taking that snapshot has

[9] The item is no more existent, if the 'pointer' does not select anything.

[10] For instance if the information item itself is still in the database but the information item representing the parent node has been deleted.

to be within a transaction with the querying. For a huge result tree fragment, the costs of this transaction can be tremendous.

As a result of the previous deliberations, we can say at least the querying process has to be transaction save. How far we need to extend the transaction, depends on the result we want to return.

4 State of the Art

This Chapter contains some insights on related work. The three major database systems provide a way to create materialized XML views or relational data. They and some other approaches shall here be compared to this thesis' proposals.

4.1 *Microsoft SQL Server 2000 XPath Querying*

Microsoft SQL Server 2000 has one of the best currently available implementations for querying relational data via XPath. It provides three access methods through two different channels to obtain relational data as XML results. The first channel is ADO (Advanced Data Objects), Microsoft's standard high level API for database access. The second channel is HTTP, where the queries are passed to the server via URL's and GET or POST. But more interesting than the channels are the three different access methods: via Transact-SQL, via Templates, and via XPath queries. All three access methods act alike for both channels, you may use the same query format, the returned results are equal.

4.1.1 Retrieving XML Data via Transact-SQL

Probably the simplest way to retrieve XML data is using Transact-SQL, Microsoft's answer to Oracle's well known PL-SQL. The simplest, since it does not require knowledge of any concepts, does not need explicit schema definitions, and it can be done by just appending a **FOR XML AUTO** clause to the end of a normal SQL **SELECT** statement. A pretty good explanation of the syntax, as well as of the modes of such a statement (**AUTO**, **RAW**, **EXPLICIT**) and some examples can be found in [Malc01]. The disadvantage of this easy to use approach to query relational data via Transact-SQL is the fast growing complexity of queries that produce reasonably structured results.

4.1.2 Retrieving XML Data via Templates

The second possibility to obtain an XML document containing relational data from the SQL Server 2000 is the querying via templates. It is not really an own access method, since it relies on either the Transact-SQL or the XPath methods. But it allows to embrace multiple queries of either type (even mixed) to return as one result document. Also the parameterization of the query becomes simpler, since the parameters are explicitly defined within the template and not coded within the URL.

There is one interesting note in [Isem01] (Chapter 13) concerning multiple queries within one template. It shall here be quoted, since it endorses our deliberations from Section 3.4 that we expect one XPath query to behave as one transaction:

> *Each* **`<sql:query>`** *or* **`<sql:XPath-query>`** *represents a separate transaction. Therefore, if you have multiple* **`<sql:query>`** *or* **`<sql:XPath-query>`** *tags in the template*[11]*, and if one fails, the others will proceed.*

4.1.3 Retrieving XML data via XPath

Since the access methods via Transact-SQL and templates differ quite a lot from what we are doing, we will not take a closer lock onto them. A detailed description can be found in [Malc01] and in [Isem01]. Instead, we will dwell on the third method, since its concepts are really near to this approach.

To access the stored relational data via XPath, also an XML view onto the database is utilized. The definition of such a view is called *annotated XML-Data Reduced Schemas* (XDR), sometimes also referred to as *mapping schema*. This corresponds to what we call the Setup (see Section 5.5). In fact, the use of an XDR as source for the Setup seems imaginable.

To pose an XPath query against an SQL Server 2000, you can specify it directly in an URL as shown in Figure 4-1, for instance. In this example URL, not only the XPath query (**`/Customer[@CustomerID='ALFKI']`**) is coded, but also several other parameters telling the http server with the name **`IISServer`** how to process it. First, the virtual directory **`nwind`** is selected. Using the 'IIS Virtual Directory Management for SQL Server Utility', the administrator created this virtual directory to allow access to a certain database within the SQL Server 2000. The next part of the path expression, the virtual name **`useSchema`**, is used to tell the server that the following file, **`schemafile.xml`**, contains an annotated XML-Data Reduced Schema that shall be used in conjunction with the appended XPath query.

```
http://IISServer/nwind/useSchema/schemafile.xml/Customer[@CustomerID="ALFKI"]
```

Figure 4-1 *Example XPath Query using HTTP*

[11] Templates are XML documents

In conjunction with the mapped schema from Figure 4-2, the returned result might look like Figure 4-3. This example has been assorted from [Isem01]. More examples, specifications and comments can be found there.

```
<?xml version="1.0" ?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
    xmlns:dt=" urn:schemas-microsoft-com:datatypes"
    xmlns:sql=" urn:schemas-microsoft-com:xml-sql">
<ElementType name="Customer" sql:relation="Customers">
    <AttributeType name="CustomerID" />
    <AttributeType name="Company" />
    <attribute type="CustomerID" />
    <attribute type="Company" />
</ElementType>
</Schema>
```

Figure 4-2 *Example annotated XML-Data Reduced Schema*

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
    <Customer CustomerID="ALFKI" Company="Alfreds Futterkiste" />
</ROOT>
```

Figure 4-3 *Result of the example query*

4.1.4 Method: The EXPLICIT Mode

Unfortunately there is no detailed description about the steps the SQL Server 2000 does to evaluate XPath queries against the relational data. In order to better understand the process, we used the Query Profiler shipped with the SQL Server 2000. The Query Profiler lets you watch all connections established to the database engine and queries that are issued against it. From the output of the profiler we could figure out, that Microsoft uses a three step process to handle XPath queries as shown in Figure 4-4.

When you pose an XPath query, one single SQL query is issued and processed within the relational engine. It is a query using the so-called EXPLICIT mode, which is very well documented in [MSDN01]. The additional **`FOR XML EXPLICIT`** directive at the end of the SELECT statement tells the output formatter not to return table data but to use the metadata in the column names (!) to format an XML output.

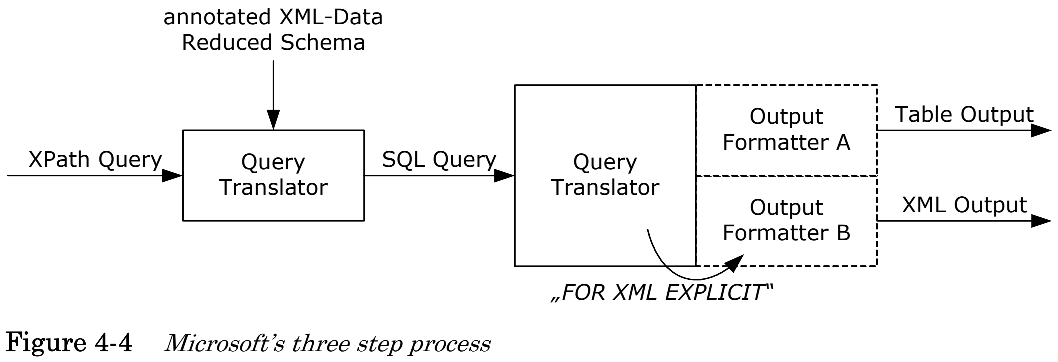


Figure 4-4 *Microsoft's three step process*

Without the **FOR XML EXPLICIT** directive, the 'universal table' (that is how Microsoft calls it) is returned. It contains one tuple for every element that will be in the resulting XML document. Each tuple consists of the total number of attributes in the resulting XML document (+2) values, approximately (+some hidden columns). Almost all of those values are NULL, except those of the attributes of the element a tuple is representing. We can get an impression of the universal table from Figure 4-5, which shows the universal table for the query **/company[@name='FZI']** applied to the view defined through the XDR from Figure 4-6. The intermediate SQL query can be found in Appendix C.

TAG	parent	company!1!name	company!1!ID!hide	division!2!name	division!2!ID!hide	employee!3!name
1	0	FZI	1	*NULL*	*NULL*	*NULL*
2	1	*NULL*	1	DBS	1	*NULL*
3	2	*NULL*	1	*NULL*	1	P.C. Lockemann
3	2	*NULL*	1	*NULL*	1	Wassili Kazakos
3	2	*NULL*	1	*NULL*	1	Alexey Valikov
3	2	*NULL*	1	*NULL*	1	Andreas Schmidt
2	1	*NULL*	1	PROST	2	*NULL*
3	2	*NULL*	1	*NULL*	2	Gerhard Goos
3	2	*NULL*	1	*NULL*	2	Benedikt Schulz
3	2	*NULL*	1	*NULL*	2	Thomas Genßler
2	1	*NULL*	1	SWT	3	*NULL*
3	2	*NULL*	1	*NULL*	3	Walter F. Tichy
3	2	*NULL*	1	*NULL*	3	Andreas Judt
3	2	*NULL*	1	*NULL*	3	Alexander Christoph
3	2	*NULL*	1	*NULL*	3	James J. Hunt

Figure 4-5 *Universal Table for* **/company[@name='FZI']**

```
<?xml version="1.0"?>
<Schema  name="DemoSchemaAnn"
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="company" sql:relation="Company">
    <AttributeType name="name"/>
    <attribute type="name" sql:field="Name"/>
    <element type="division">
        <sql:relationship key-relation="Company"
                    key="ID"
                    foreign-relation="Division"
                    foreign-key="Company"/>
    </element>
</ElementType>
<ElementType name="division" sql:relation="Division">
    <AttributeType name="name"/>
    <attribute type="name" sql:field="Name"/>
    <element type="employee">
        <sql:relationship key-relation="Division"
                    key="ID"
                    foreign-relation="Employee"
                    foreign-key="Division"/>
    </element>
</ElementType>
<ElementType name="employee" sql:relation="Employee">
    <AttributeType name="name"/>
    <attribute type="name" sql:field="Name"/>
</ElementType>
</Schema>
```

Figure 4-6 *XDR for the example from Section 1.2*

The concept of the universal table seems suitable for serializing a document fragment. It has a clear, defined structure with limited complexity and is able to carry its metadata through the SQL statement to the serializer (allows a sensible encapsulation of the query translator and the serializer at the cost of little bulkier column names).

"The universal table rowset (containing all data and meta data) is scanned one row at a time, in a forward-only manner, producing the resulting XML tree. To yield the proper XML document hierarchy, it is also important to specify the order of rows in the universal table. This is achieved by using the ORDER BY clause in the query." (source: [MSDN01])

Unfortunately, a universal table can only be obtained using the UNION construct, which causes problems in optimizing the query. Applied conditions as well as joins have to be recalculated multiple times, if the optimizer is not able to detect (and make use of) common sub expressions (see SQL query text from Appendix C).

4.1.5 Distinctions of Approaches

Besides platform dependence, there are other distinctions between Microsoft's approach and ours. The view concept underlies both approaches to map XPath selectable elements and attributes to tables and columns. Unfortunately, Microsoft does not provide access to this view. Instead of benefiting of all the advantages of it (including relative queries and navigateable results), SQL Server 2000 uses the meta information for query translation and result formatting, only. The result is provided as XML text (as a string) instead of returning a pointer to a virtual node like we do.

The drawback of the return of the result in text form is, that the returned XML document has to be filled out completely, it has to be *deep-materialized*. Every node of the result tree fragment, up to the most distant leaf, has to be obtained from the tables, resulting in deeply structured and huge materialized documents for more complex XDR's. According to 5.5.3, we encourage the user to nest elements wherever this is possible, since it simplifies the XPath queries required to express his desired result. With the Microsoft approach, the user is rather discouraged to do so for the fear of performance problems. As a consequence, typically XDR's are defined with a very limited nesting level.

The broadness of the defined document is usually not as important, as every step narrows down the volume of result nodes by an order of magnitude, since neither **`ascendant`**, **`ascendant-or-self`**, **`descendant`**, and **`descendant-or-self`** axes, the selection of the root node, the wildcard node test "*" (for example, **`child::*`**), nor the union operator "|" are supported. (A detailed description of what is supported an what not can be found in [Isem01].) It is impossible to select more than one certain kind of nodes from their virtual document. This enforces the selectiveness of queries to be comparatively high and thus reduces the overall amount of data to return at the cost of compatibility and comfort. Of course we do not know, whether or not that is reason for these XPath features not to be supported, but it seems very likely.

A workaround to query a top level element without having all its nested elements materialized (for instance since we do not need the information contained in there) is to redefine the top level element with an other name and without its nested elements for an other time. This could be done for every nesting level or at least for those, known to be required by the application. Again, this requires the applications needs to be known before the view definition. Alternatively, the user could abandon the comfort of querying for nested elements (remember: a design criteria for XPath) and use links as described in 5.5.3.

4.2 *IBM (XPERANTO/XTABLES)*

Even though IBM's DB2 offers XML support via their DB2 XML Extender[12], we will focus on an approach from the IBM Almaden Research Center in San Jose, since its functionality is much closer to our aims. The DB2 extender offers the storage and querying of XML documents in single columns via **XMLCLOB**s, **XMLVARCHAR**s and **XMLFile**s. Further on, it allows shredding and (re-) composition of XML data from or into relational tables. But in the latter case, no XPath (or comparable) querying is available.

XPERANTO (see [CFI+00]) might be considered as the mother-of-all approaches on this field. Every other paper mentions XPERANTO, and we can also not miss out this important publication.

Basically, XPERANTO also embodies a middleware to connect the worlds of XML and of relational databases by processing a native XML query language, translating the queries to XML and returning the resulting relational table data as XML data. In contradiction to our support for XPath, XPERANTO uses XML-QL (see [DeFF99]) as querying language. The obtained results are tagged and returned as strings, the disadvantages of this approach have already been elucidated in Section 3.3 and 4.1.5. XPERANTO uses the so called *sorted outer union* approach for materializing the results (see [SSB+00] for details).

The translation process of XPERANTO is on a very rough level equal to ours: first the original query language is parsed and translated to an intermediate query representation. What we call the Expression (see Section 5.6) is called XQGM (XML Query Graph Model), an extension to DB2's QGM.

[12] see http://www-3.ibm.com/software/data/db2/extenders/xmlext/index.html

Since the two original papers (see [CFI+00] and [CKS+00]) on XPERANTO are relatively rare, the interested has to resort to [SKS+01], [SSB+00], or [Shan01] for more information. There is also an other paper concerning XTABLES (see [FFL+02]), which seems to be the new name for XPERANTO concepts, since it covers the same content, it is available at the IBM XPERANTO web sites and it is of newer date (March 2002). Hence, the following details have been taken from [SKS+01] and [FFL+02].

The approach provides a *Default XML View* which maps every table to an element with the tag name of the element equal to the table name it represents. Beyond these elements, **row** elements are listed, each representing one tuple from the table. The row element again contains the column values as sub elements, these sub elements carry the name of the column they correspond to. The topmost element has the tag name **db**. The structure of this view corresponds to what we called the simulation (see Section 3.2).

Fortunately, the user of XTABLES does not have to labor (see Figure 3-2 for a query on a simulating document) with this view, at least not in an advanced development stage. XTABLES in contradiction to XPERANTO supports XQuery as querying language. A query in this language also creates a view on the queried data, the data is returned in the thus defined structure. These views are stored within XTABLES and may be referenced by future queries. That way, a kind of view repository emerges. This is shown in Figure 4-7.

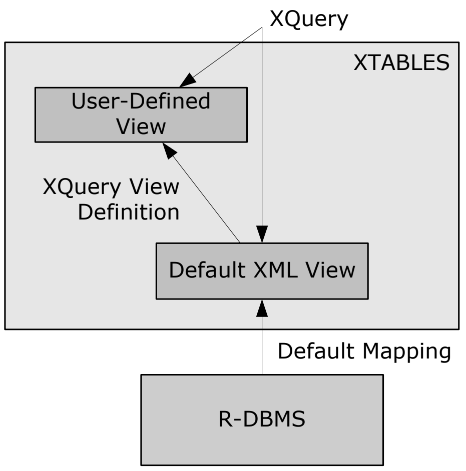


Figure 4-7 *XTABLES Query Processing Architecture*

The translation process, described in [Shan01], is a lot more complex than ours. According to a PowerPoint presentation of Javel Shanmugasundaram found in the web[13], the translation of a query about 12 tables takes about 200 ms (pure translation time, not the processing of the SQL query). This probably dues to the

[13] see http://www.cs.cornell.edu/People/jai/presentations/QueryingXMLViews.ppt

additional features like view definition, materialization, and reuse. In contradiction, we applied certain, not very limiting conventions, allowing us to provide a much simpler but still useful solution.

4.3 *Jain/Mahajan/Suciu (University of Washington)*

Sushuant Jain, Ratul Mahajan, and Daniel Suciu presented an interesting approach on the 11th WWW Conference in Honolulu in May, 2002. Their idea was to translate a complete XSLT style sheet into corresponding SQL statements to return relational table data in XML form (see [JaMS02]). The advantage over a pure XPath querying approach is to overcome the problem, that no assembled results are presentable within a single query. This is also the main animadversion of [SKS+01] on XPath approaches (especially the Microsoft approach): *'XPath cannot specify joins'*[14].

The supported fragment of the XSLT language is very limited though. And also the XPath support does not transcend Microsoft's approach by far, yet wildcards and the union operator are supported. This can be read out of their grammar, found at their projects homepage (see [JaMS02], a copy of the grammar can be found in Appendix E).

A problem involved by their way of processing is the need for the optimization of the generated SQL queries, especially nested, correlated IN sub queries. Another optimization they introduce are *QTree Reductions* (QTree is the name for one of their internal query representations, it is kind of a mixture of our Path Tracing Graph (see Section 5.7) and our Expression (see Section 5.6)). They apply shortcuts to *'long path with unreferenced intermediate nodes'* ([JaMS02], Section 5.3), which means that *'no intermediate node in the path is referred to by any other part of the QTree* [remember, it also contains conditions] *except the immediate child and parent of that node on that path'.* But this may lead to incorrect results (see Section 5.4.4), if it is not done very carefully. Thus such optimizations are beyond our scope (see Section 5.4.3.1).

[14] By the way, this statement as it is, is incorrect: XPath can very well specify joins within predicates, it just can not assemble output from multiple sources.

4.4 *Other Approaches*

The following approaches are not observed as accurate, since their aim differs more or less from ours. But since they are also loosely related to our theme, we will have a quick look onto them.

4.4.1 XRel

The XRel approach (see [YASU01]) tries to find a way of storing and retrieving XML documents in and from relational database systems. Even though we were not bound to the task of storing the documents, XRel was the approach we were originating from, since it bases on a very simple and yet efficient idea of how to process XPath queries.

The process of querying is bound to the way the data is stored, of course. [YASU01] discerns two possible ways for storing XML documents in relational databases: the *model-mapping approach* and the *structure-mapping approach*. In the former, the relational database schema represents the constructs of the XML document model. There may, for instance be a table for attributes, a table for elements and a table for assigning attributes to elements and elements to their parent elements. In the later approach, the database schema is based on the XML document structure, expressible by a DTD (Document Type Definition) or an XML Schema (http://www.w3.org/XML/Schema). This is much closer to the terms of real relational database but is liable to the limitation, that only documents complying to the once modeled schema can be stored. While [YASU01] sticks to the model-mapping approach due to its flexibility, we rather comply with the structure-mapping approach, since we only retrieve existing data that is very likely not stored in terms of the XML document model. Further on, the structure-mapping approach promises performance advantages over the model-mapping approach, since it is able to utilize classical optimization techniques.

In contradiction to our approach, the XPath support in XRel is limited to what is called *XPathCore* in [YASU01]. There is no support for ancestor axes (**parent**, **ancestor**, **ancestor-or-self**), the self axis or preceding and following axes (**preceding**, **preceding-sibling**, **following**, **following-sibling**). It is described as

"...basically the intersection of the nonterminal symbol ***PathExpr*** *in XPath 1.0 [W3C99] and the nonterminal symbol* ***PathExpr*** *in Quilt [ChRF00]"*

This very restricted set of axes allows the usage of a table storing materialized string representations for every existing path expression of the stored documents.

Those strings can be searched via SQL-92 compliant LIKE (see [Date93]) for the matching path expression of a posed query. A great advantage of this approach is that queries making heavy use of the **`descendant-or-self`** axis (via **`//`**) do not require join operations proportional to the length of the path.

For our approach, this is not applicable, since only providing a view, we can not take the responsibility of keeping such a table consistent with the rest of the database without heavily interfering with the original applications via triggers. The problem is that a parent node might be deleted, while its child is rerouted to an other parent node. See also Section 5.4.3.1 on this.

4.4.2 Xalan (Apache)

The Java Development Kit (JDK) 1.4 comes with a build in XSLT/XPath processor which is the one from the Apache Software Foundation (http://www.apache.org). This product that is also available as a separate download as well as for C++ (see http://xml.apache.org/) offers a way to access relational database data within an XSLT style sheet. But this is not done via a view, it is realized utilizing the concept of extension functions (see [Kay01], Chapter 8). The procedure for the user is shown in Figure 4-8.

```
<?xml version="1.0"?>
<xsl:stylesheet   xmlns:xsl=http://www.w3.org/1999/XSL/Transform
                  version="1.0"
                  xmlns:sql="org.apache.xalan.lib.sql.XConnection"
                  extension-element-prefixes="sql">
...
    <xsl:param name="query" select="'SELECT * FROM import1'"/>
...
    <!-- 1. Make the connection -->
    <xsl:variable name="products"
                  select = "sql:new('org.enhydra.instantdb.jdbc.idbDriver',
                          'jdbc:idb:D:\instantdb\Examples\sample.prp')"/>
...
    <!--2. Execute the query -->
    <xsl:variable name="table" select='sql:query($products, $query)'/>
...
    <!-- Get column-label attribute from each column-header-->
    <xsl:for-each select="$table/sql/metadata/column-header">
        <xsl:value-of select="@column-label"/>
    </xsl:for-each>
...
    <xsl:apply-templates select="$table/sql/row-set/row"/>
...
    <!-- 3. Close the connection -->
    <xsl:value-of select="sql:close($products)"/>
...
</xsl:stylesheet>
```

Figure 4-8 *Obtaining relational data in Xalan*

4.4.3 Oracle

There is no direct way in Oracle 9i Database to access relational data via XPath. Oracles solution for using relational table data in XML-based components is called XSQL. Standard SQL query results are converted to XML output via the XML-SQL Utility (XSU), nesting is achieved through *structured columns*[15]. Thus obtained results may then be parsed and processed with the accompanying (or any other) XSLTransformer using standard XPath queries.

4.4.4 ROLEX

ROLEX (see [BoKN01]) addresses issues required to support high-performance navigation of XML views of relational data. Their idea of providing a virtual XML document through standard interfaces like DOM and SAX over relational database table data is close to our concept of the virtual document.

ROLEX has no real support for querying. Instead, the defined views may be *parameterizes* as it is called in [BoKN01]. But their main focus is rather the caching of heavily used data in between the relational tables and the virtual document they provide, to allow quick navigation through the tree. For these purposes they rely on the DataBlitz ™ Main-Memory Database System (see [BBG+99]). Its trigger mechanisms are used to keep materialized portions of the virtual document up-to-date, or to invalidate sections of the materialized tree. This is indeed an improvement in comparison with the caching of materialized XML documents in a business application. The techniques described in conjunction with ROLEX, especially in the paper that will be published on VLDB in August 2002, [BGK+02], might be worth considering for the improvement of our own virtual document.

4.5 *Summary*

Summing up, we can state that the combination of an XPath querying in conjunction with a virtual document is unique until today. An even better digest might be given by the table from Figure 4-9.

Approach	**Query Language**	**XML View Features**				**XPath Features**												
		External Difinition	**Access to relational Table Data**	**Document Order**	**Navigation**	**Axes**									**Relative Queries**	**Cross Product Queries**	**Support for UNION ("|")**	**Result**
						anchestor (or-self)	attribute	child	descendant (or-self	following (sibling)	namespace	parent	preceding (sibling)	self				
MS SQL Server	XPath	yes	yes	no	no	n	y	y	n	n	n	y	n	y	no	no	no	Text
X2S	**XPath**	**yes**	**yes**	**no**	**yes**	**y**	**y**	**y**	**y**	**n**	**n**	**y**	**n**	**y**	**yes**	**yes**	**yes**	**Pointer**
XRel	XPathCore	yes	no	yes	no	n	y	y	n	n	n	n	n	n	no		no	Text
Jain et al.	XSLT	yes	yes	no	no	n	n	y	n	n	n	y	n	y	no		yes	Text
XPERANTO	XQuery	yes	yes		no										no			Text
Oracle 9i	SQL	no	yes	-	no	-	-	-	-	-	-	-	-	-	-	-	-	(Text)
Xalan	SQL	no	yes	-	no	-	-	-	-	-	-	-	-	-	-	-	-	(Text)
ROLEX	(none)	yes	yes		yes	-	-	-	-	-	-	-	-	-	-	-	-	(Pointer)

"-" = not applicable, [blank] = undocumented

Figure 4-9 *Summary of Comparison of Approaches*

[15] for details see http://otn.oracle.com/docs/tech/xml/xdk_java/doc_library/Production9i/doc/java/xsql/readme.html#ID3260

5 Design & Realization

We propose a proxy-based solution to achieve the goal. The core algorithm is relatively close to the naïve solution, thus easy to understand and to implement. To archive this, we need a powerful object model providing the required operations. The important operations will be presented in detail, the remaining are obvious and easy to implement. Thus for the conciseness, we are not naming them explicitly, here.

5.1 *Virtual DOM*

The approach directly connects two existing, fairly well know and approved worlds: relational databases and XML technology. Each of both has a standard representation within the Java framework which we exemplarily choose here. Other frameworks provide similar constructs. The connection between Java and database systems is established by a technology named JDBC (Java Database Connectivity). XML on the other hand may be processed by JAXP (Java API for XML Processing). That API contains two access methods: through SAX (Simple API for XML) or through DOM (Document Object Model). While the first is an event driven interface providing a mechanism for "callback" notifications to application's code, the latter is the W3C DOM Working Group's set of interfaces to access a more or less static in memory object representation of an XML documents elements. We choose to utilize DOM since its vocabulary of concepts is closer to the classical notion of database views.

In DOM every attribute, every element, all text nodes and all items that may else be contained in an XML document are represented by an individual object of the type 'node'. These nodes are connected to each other by functions allowing basic navigation up and down the object tree. As a result to this design, a naïve implementation requires the whole document to fit into a computers main memory to be processed efficiently.

As mentioned earlier, a possible way to fulfill our task is to build the document by coping the content of our view from the database into an XML document. Afterwards, we could process the XPath queries with optimized and proven existing tools. For a naïve DOM implementation, we had to copy all the databases content into one computer's memory (see Figure 5-1). This would mean a heavy system load, especially for large systems with several gigabytes of data. But since we provide an XML view onto a relational database, we must be able to map all the content of that database into our document. Thus we need a work-around to only keep a few elements in memory but allowing unhindered navigation and of course querying throughout the whole content, nonetheless.

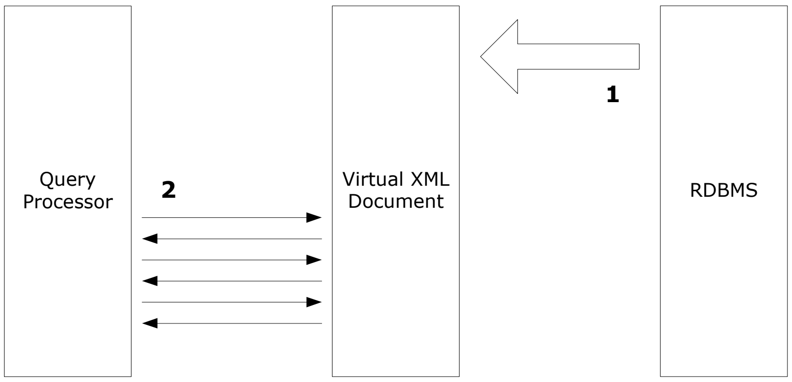


Figure 5-1 *Naïve approach*

We call the concept to circumvent this drawback *virtual DOM.* The idea is to keep a template document in memory. It only represents the structure of our document. Its elements are treated as proxies for zero, one, or many instances, depending on the data stored in the relational system. They can also be considered as schema elements or as 'classes' of different instances, depending on the point of view.

Each node within the template document stores a 'pointer' into the relational database[16]. But following such a pointer does not lead to one distinct piece of information, it selects a set (containing 0, 1, or multiple elements) of information items. Obtaining these from the database, we can create instances of those classes. These instances, in turn are the nodes of the virtual DOM and a 1:1 representation of database information items.

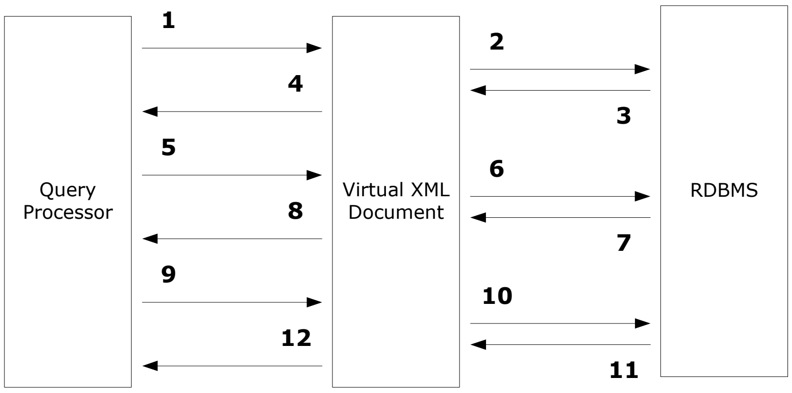


Figure 5-2 *Virtual document as proxy for database access functions, i.e. navigation*

[16] A pointer in this case refers to a certain column and is representable by the columns name and the containing tables name

Now we could already process XPath queries against such this document (see Figure 5-2). This would mean to obtain every node from the database that an XPath query touches. But most of those nodes are discarded since they literally are a step on the path to the result, only. To save resources, we only want to output data from the database we probably will need for further processing. The result nodes are part of it, nodes that only occurred in the XPath query to express conditions are not. Thus, we only want to create instances for the result nodes. The other nodes should be processed via their proxies and within the relational database system (see Figure 5-3). This mechanism can only be provided for queries, of course. A following navigation through the virtual document by the standard org.w3c.dom navigation functions is only realizable via the proxy method illustrated in Figure 5-2.

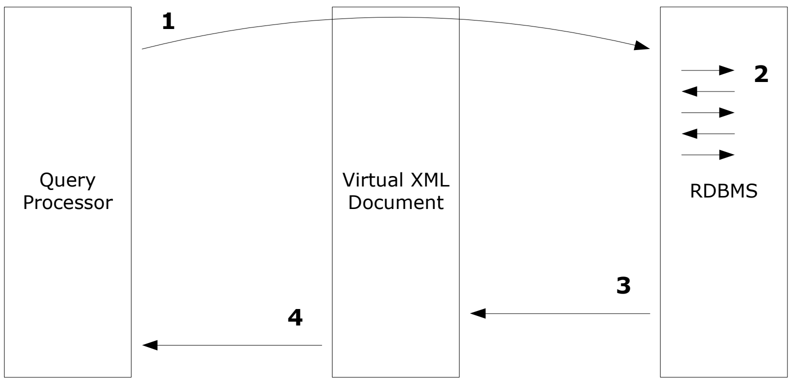


Figure 5-3 *Query processing within the frontiers of the RDBMS*

More formally spoken, Figure 5-2 represents method A in Figure 5-4, Figure 5-3 represents method B, whereas DB and DB’ mark up the states of the original (relational) database and VDB and VDB’ mark up the states of the view (the virtual document here). For a read-only approach, ‘state’ refers to what information item(s) we are currently looking at. An operation o of a user, this can only be a navigation in read-only case, must be translated into an operation p on the original database, so that o(v(DB))=v(p(DB)), to achieve reasonable performance boosts from enabling the underlying database engine to use its optimizing techniques.

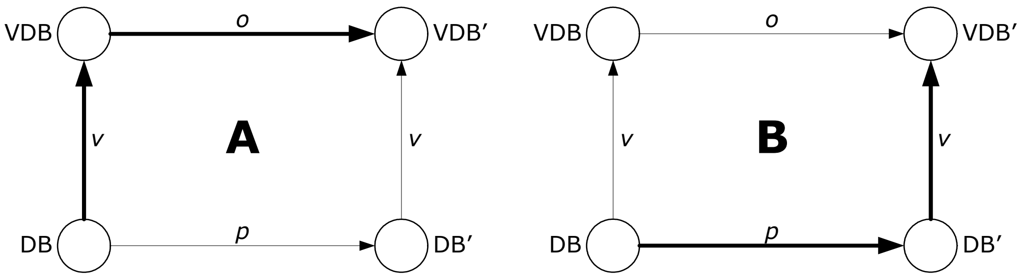


Figure 5-4 *Operating on the view and operating on the model*

Apart from the traffic savings, the memory savings within the client computer evaluating the XPath expression are substantial. While keeping all existing instances containing a pointer[17] to their content would mean a compression ration of 1:100[18], keeping only the classes means a compression ratio of 1:1,000 or more[19]. If the data is of an other type than VARCHAR, maybe INT or DATE, the overhead for a pointer in every instance may even lead to negative savings or in other words increased memory consumption.

The third drawback of keeping the instances is the problem of keeping them consistent with the database. But this has already been discussed in section 3.4.

5.2 *Extended Example*

In this section, we extend the example from section 1.2. Now we can additionally save products of our companies and takers (customers) that may order them. The resulting database structure can be seen in Figure 5-5, Figure 5-6 shows some sample table data. The schema of the desired virtual document is shown in Figure 5-7, the resulting virtual document in Figure 5-8.

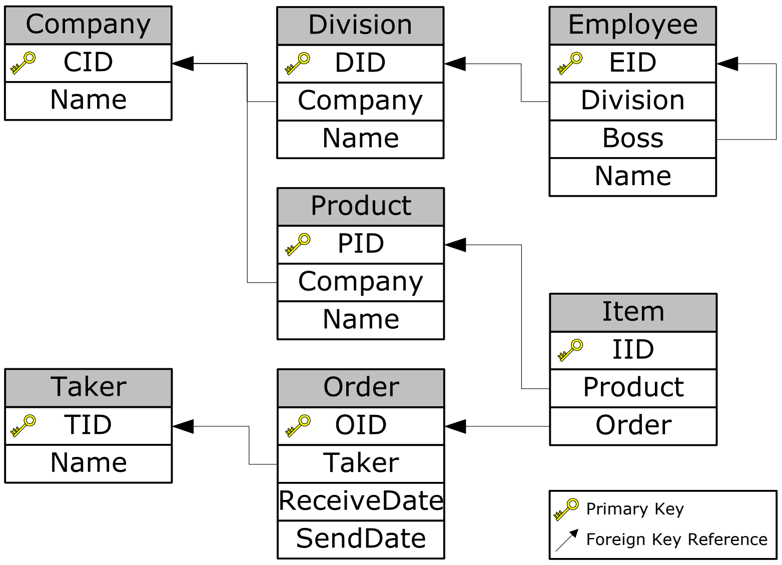


Figure 5-5 *The Extended Examples relational schema*

[17] A pointer in this case is a set of columns that form a key to a certain table, a set of assigned values, identifying a certain row and a column selector, selecting the distinct value from the row. Such a pointer can be used to create a simple **SELECT FROM WHERE** statement to obtain the desired data from the database.
[18] Assuming the pointer to point to VARCHAR(200) data fields
[19] Assuming the content to be VARCHAR(200) data fields and a representation of only 10 instances (i.e. 10 rows) per class.

T_Company	
ID	**Name**
1	FZI
2	Siemens

T_Division		
ID	**Company**	**Name**
1	1	DBS
2	1	PROST
3	1	SWT
4	2	A&D AS IT
5	2	A&D AS PAS

T_Employee			
ID	**Division**	**Boss**	**Name**
1	1		P.C. Lockemann
2	1	1	Wassili Kazakos
3	1	2	Alexey Valikov
4	1	2	Andreas Schmidt
5	2		Gerhard Goos
6	2	5	Benedikt Schulz
7	2	6	Thomas Genßler
8	3		Walter F. Tichy
9	3	8	Andreas Judt
10	3	9	Alexander Christoph
11	3	9	James J. Hunt
12	4		Alfred Schmit

Figure 5-6 *Some sample table data*

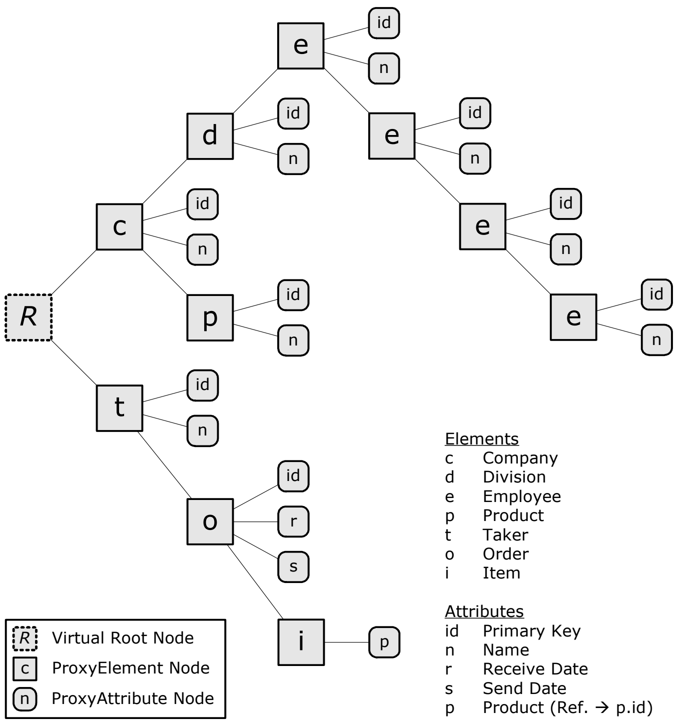


Figure 5-7 *The document structure of the Extended Example*

```
<company id="1" name="FZI">
    <division id="1" name="DBS">
        <employee id="1" name="P.C. Lockemann">
            <employee id="2" boss="1" name="Wassili Kazakos"/>
                <employee id="3" boss="2" name="Alexey Valikov" />
                <employee id="4" boss="2" name="Andreas Schmidt" />
            </employee>
        </employee>
        <employee id="2" boss="1" name="Wassili Kazakos">
            <employee id="3" boss="2" name="Alexey Valikov" />
            <employee id="4" boss="2" name="Andreas Schmidt" />
        </employee>
        <employee id="3" boss="2" name="Alexey Valikov" />
        <employee id="4" boss="2" name="Andreas Schmidt" />
    </division>
    <division id="2" name="PROST">
        ...
    </division>
    ...
    <product id="1" name="NOKIS"/>
</company>
<company id="2" name="Siemens">
    <division id="5" name="A&D AS IT">
        <employee id="16" name="Alfred Schmit"/>
    </division>
    ...
</company>
<taker id="1" name="unknown organization">
```

```
    <order id="1" receivedate="October 2001" senddate="somewhen in 2002"
        <item product="1"/>
    </order>
</taker>
...
```

Figure 5-8 *XML representation of the Extended Example*

In the example in Figure 5-8, several persons occur multiple times in the virtual document. This behavior is by design, since we do not want to restrict the user in the way he treats recursive data. The drawback is, that the user will not be able to select one distinct node from the virtual document by searching only for the key attributes' values. Instead we will receive a list of nodes. At first look, this may seem to be a great limitation, but indeed, it is not. All these nodes will have identical data and within their structural position, they are unique and thus distinguishable all over the virtual document. Further on, we can take the advantage that it can be left to the user whether he likes to iterate through the nodes flatly (employees by division) or whether he wants to search the hierarchical structure (employee hierarchy) deeply. We do not need to force the user to decide which way he probably prefers and to provide a condition that will block the other one. Both ways are possible without mutual interference. The flat enumeration can be achieved by a simple query like Figure 5-9 (just without performing a deep search afterwards), the hierarchical structures root nodes can be obtained with a query like Figure 5-10.

```
/company/division[@name='DBS']/employee
```

Figure 5-9 *XPath query obtaining a flat enumeration of employees*

```
/company/division[@name='DBS']/employee[./employee]
```

Figure 5-10 *XPath query obtaining only the 'bosses'*

5.3 *The Basic Object Model*

As already mentioned in Chapter 1, both XPath and SQL are descriptive query languages. Consequently, if we use a subtle mapping that is close to a 1:1 relation of the objects of both domains, the translation process is straightforward. Thus we first try to find a representative for each XML term in the relational domain and vice versa.

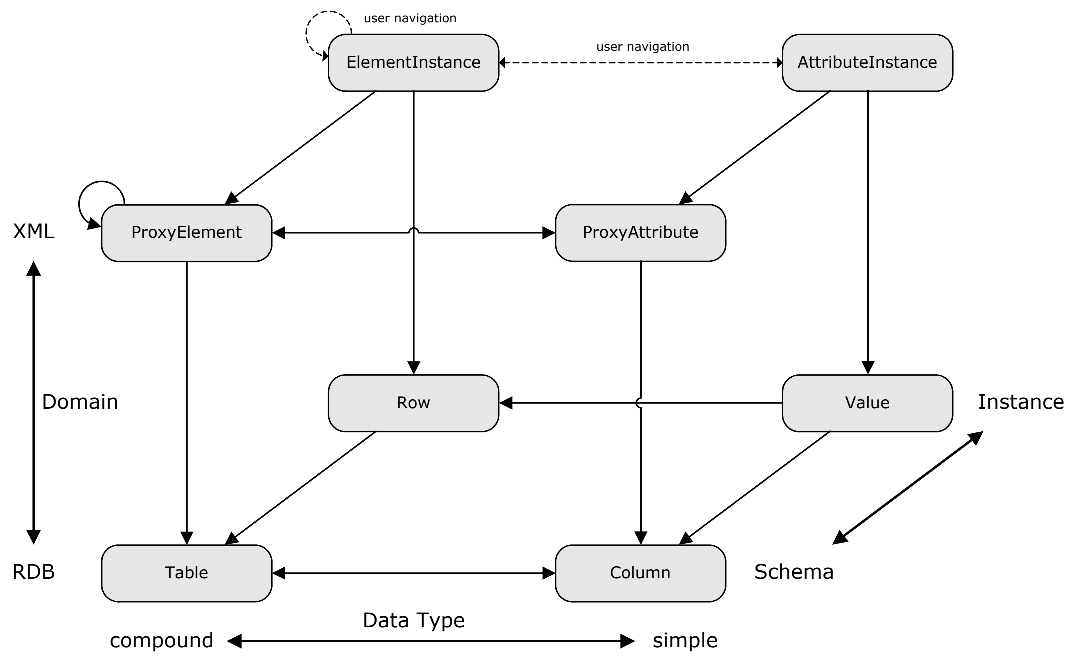


Figure 5-11 *The Basic Object Model*

In the relational world, four basic entities can be found. These are the table, its rows and columns, and the values. Values can be found in each row at a certain position (column), thus rows (or 'tuples') are the container elements for values. Tables are used to group tuples of the same type.

XML documents on the other hand are built from elements and attributes.[20] They store the data. Though they bring their own structuring, there is a meta structure, indeed. It is called *Schema*.[21] It describes at which position new 'records' (attributes or elements) may be inserted or where existing ones might be found.

Now, comparing both domains, we see that both contain two data containers, one for simple, noncomplex data-typed values, and one complex, containing the simple ones. Further on, both contain a schema, describing where to add or where to find data of a certain category. Thus, searching for a 1:1 mapping, we can find a sensible interpretation for each object as shown in Figure 5-11[22].

Fortunately, the user sees hardly anything of this. Once he has provided the Setup (see section 5.4), he only sees a large virtual document as demanded in section 5.1. He simply accesses DOM nodes and navigates through the document via their functions.

[20] Of course, also other constructs exist, but to keep things simple we stick to this statement.

[21] An explicit Schema is not required. If no explicit Schema is provided, the implicit Schema, a subset of the explicit Schema, can be extracted from the document.

[22] You may ask yourself why the XML Schema objects are called 'proxy'. This is because the proposed algorithm works on these (not on the instances) and calling them SchemaElement and SchemaAttributes or likewise made things very confusing.

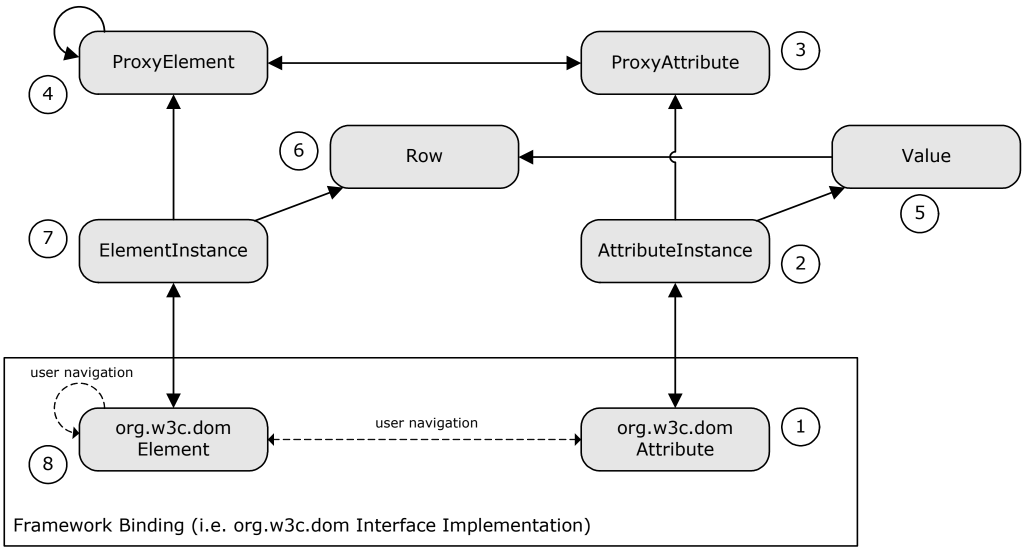


Figure 5-12 *Navigation from an attribute to its containing element*

Let's say an XPath query returned an attribute node. Now, if we want to retrieve the containing element node, we would call the **`getElement()`** function in an org.w3c.dom implementation. Since the document is virtual, neither the attribute node has a direct reference to its containing element node, nor is this element node even materialized due to the virtuality of the whole document. But for the sake of object orientation, the following process can be hidden from the user ash shown in Figure 5-12:

Starting from the attribute node (1), we can find the corresponding **`AttributeInstance`** object (2).[23] The **`AttributeInstance`** object itself contains two pointers, one to its Schema element, the **`ProxyAttribute`** (3) and one to the Value object (5), this **`AttributeInstance`** represents. From the **`ProxyAttribute`**, we can find out the containing **`ProxyElement`** (4), and the Value object contains a reference to the Row (6) it can be found in. Knowing the Row and the **`ProxyElement`**, we can create an **`ElementInstance`** object (7), which corresponds to the element node (8) that 'contains' the original attribute node and thus will be returned as the function's result.

[23] Both, attribute node and **`AttributeInstance`** object could be represented by one object. The separation is done to encapsulate the object model of this approach from a certain DOM API.

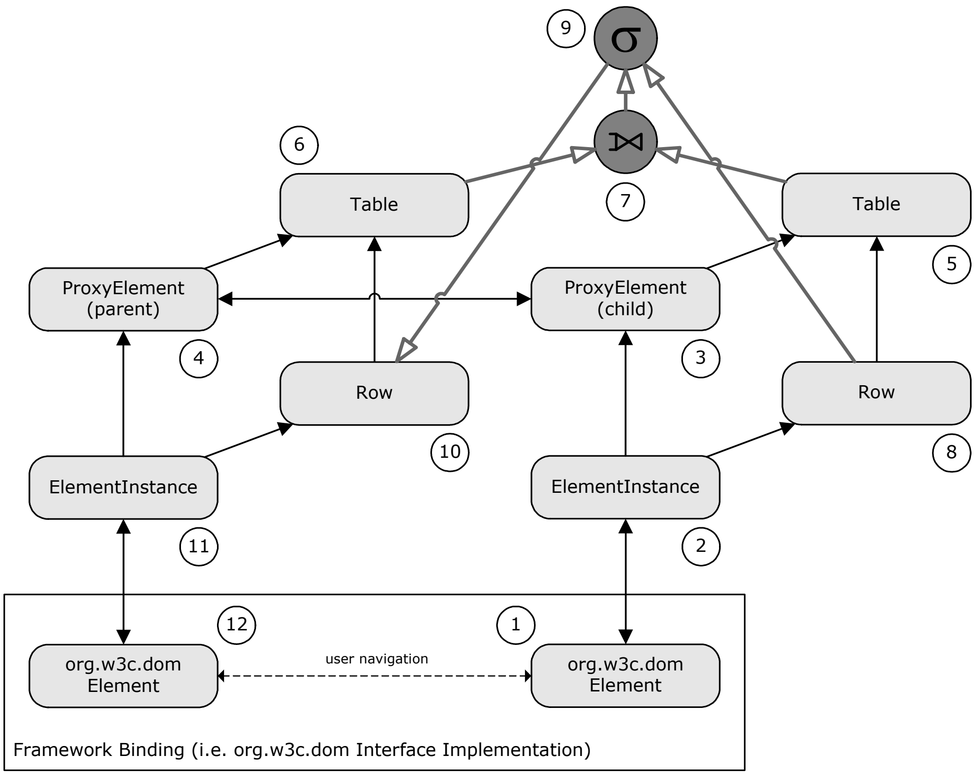


Figure 5-13 *Navigation from an element to its containing parent element*

The navigation from a child element node to its parent element node (shown in Figure 5-13) is a bit more complex since it requires a database access. The origin element (1) has a corresponding **`ElementInstance`** (2). This **`ElementInstance`** has a certain **`ProxyElement`** (3) as Schema, for which indeed we can find a parent **`ProxyElement`** (4). Both, parent and child represent each one **`Table`** element (5 and 6). The Join on these two tables (7) represents the parent child relation of the **`ProxyElement`**s. Searching for a certain parents data, we only select those result rows (9) that comply with the **`Row`** condition (8) of our origin **`ElementInstance`**. The resulting rows deliver the data to create the Row objects (10) that we need to create the **`ElementInstance`**s (11) in conjunction with the parent **`ProxyElement`** (4). These **`ElementInstance`**s correspond to the element nodes (12) to return as result.

Now, having spoken about the implementation of the **`getParent()`** function of an element, why do we talk about result nodes (plural)? If the join (7) is not an asymmetric join using key and foreign key but a symmetric join using just two columns required to be equal, the selection (9) may return more than one result row. In this case, the parent node is unfortunately not clear without ambiguity. That is why we demand the Setup to only use key and foreign key to nest the defined elements. A concrete implementation must be able to cope with this situation, nonetheless.

5.4 *Demanded Translation*

This section should clarify, how we expect the different constructs of XPath queries to be translated. This section appears in this chapter and not in Chapter 3, since the translation may differ from approach to approach (see Chapter 4). Often, there are several syntactically valid ways to express the same semantic query.

5.4.1 Translating Elements and Columns

A query for the names of all companies would look like Figure 5-14 in XPath against our virtual document. As we represent tables by elements and columns by attributes, it is not very surprising that the corresponding SQL statement should look like Figure 5-15.

```
/company/@name
```

Figure 5-14 *XPath: all companies names*

```
SELECT Company.Name FROM Company
```

Figure 5-15 *SQL: all companies names*

5.4.2 Translating Conditions

Conditions increase the selectiveness of queries and thus are a main source for performance improvement by executing the query within the database system instead of outside on the virtual document data structure. The query for the ID of the company with a name equal to 'FZI' is an example here. Figure 5-16 shows this query in XPath syntax. The translated query in SQL should look like Figure 5-17.

```
/company[@name='FZI']/@id
```

Figure 5-16 *XPath: the id of the company called 'FZI'*

```
SELECT Company.ID FROM Company
WHERE Company.Name='FZI'
```

Figure 5-17 *SQL: the id of the company called 'FZI'*

5.4.3 Translating Nested Elements

Translation of nesting is probably one of the trickier tasks. Due to the paradigm gaps between the relational model and the semi-structured model, it can not be done without some metadata (see section 3.1.1). Consequently, every translation bears on the metadata provided during the translation. It is only valid in conjunction with the provided setup.

5.4.3.1 *First Approach*

A query like the one in Figure 5-18 returns the names of all divisions. More precisely, the names of all divisions that belong to any company are queried. If our database also let us store divisions of clubs or universities, there would be a different result from the query in Figure 5-18 to the query in Figure 5-19, indeed. Consequently, our translation needs a join as shown in Figure 5-20. Regarding the equality of the results of the queries in Figure 5-18 and Figure 5-19 respectively of the queries in Figure 5-20 and Figure 5-21 for our very special case, automatically exchanging them means an optimization step requiring even more metadata in the setup. Such optimizations are beyond the scope of this thesis.

```
/company/division/@name
```

Figure 5-18 *XPath: the names of all divisions of all companies*

```
//division/@name
```

Figure 5-19 *XPath: the names of all divisions*

```
SELECT Division.Name FROM (Company INNER JOIN Division)
```

Figure 5-20 *SQL: the names of all divisions of all companies*

```
SELECT Division.Name FROM Division
```

Figure 5-21 *SQL: the names of all divisions*

Figure 5-20 contains a short form of the inner join. The join condition `ON Division.Company=Company.CID` has been left out, since the human reader who knows the database schema from Figure 5-5 can easily supplement it notionally. Especially for the following more complex expressions we will use this concise notation.

5.4.3.2 *Correctly Translating Nested Elements*

The next example query will show us which join type we have to choose to correctly translate nesting. Assume, we want the names of all companies of which either themselves, any of their divisions or any of their employees has the name 'Knox'. The corresponding XPath query is shown in Figure 5-22. A possible translation is shown in Figure 5-23.

```
/company[.//@name='Knox']/@name
```

Figure 5-22 *XPath: the names of all companies where either the company itself, any of its division, or any of its employees has the name 'Knox'*

```
SELECT Company.Name
FROM ((Company INNER JOIN Division) INNER JOIN Employee)
WHERE Company.Name = 'Knox'
OR Division.Name = 'Knox'
OR Employee.Name = 'Knox'
```

Figure 5-23 *SQL: the names of all companies where either the company itself, any of its division, or any of its employees has the name 'Knox'*

But: What if a firm named 'Knox' has neither any employees nor any divisions stored in our database? The XPath query would return it, so we must adapt the SQL query to return the company if either its name is 'Knox', or if it has a division and that divisions name is 'Knox', or if it has a division and this division has an employee and this employees name is 'Knox'. This relation is expressed by the `LEFT OUTER JOIN`. A corresponding SQL query is shown in Figure 5-24.

```
SELECT Company.Name
FROM ((Company LEFT OUTER JOIN Division) LEFT OUTER JOIN Employee)
WHERE Company.Name = 'Knox'
OR Division.Name = 'Knox'
OR Employee.Name = 'Knox'
```

Figure 5-24 *SQL: the names of all companies where either the company itself, any of its division (if it has any), or any of its employees (if it has any) has the name 'Knox'*

As a result to these considerations we note that we can build a simulation of the virtual document within our database by `LEFT OUTER JOIN`ing the necessary tables in the `FROM` clause. This is what is called *Redundant Relation Approach* in [SSB+00].

5.4.4 Translating Existence

Another invidious feature (for implementers) of the XPath query language is that a query expression does not have to begin somewhere down at the root and go straight up to a leaf of the queried document. It may also be formulated upside down or even jump higgledy-piggledy through the nodes of the object tree.

An example is shown in Figure 5-25. This query is not to be mixed up with the query of Figure 5-14, since we select only names of companies here that have at least one division with at least one employee (at least in our database).

```
/company/division/employee/../../@name
```

Figure 5-25 *XPath: the names of all companies for which at least one division as well as at least one employee is stored*

The XPath standard defines to evaluate such location paths step by step.[24] Every step selects the nodes on the corresponding axis and filters the nodes by a name test (or node test) afterwards. Finally, the conditions are applied, removing all nodes not complying with them. The resulting nodes are the local root nodes for the next step.

Regarding the whole location path expression, per step from the very first until the progenitor one a node had to exist and fulfill the predicates to allow the current node to be in the result set. Thus, when working on the proxies of the template document, we need an additional existence test on every node our location path touches besides the explicit predicates.

We will translate the test for existence of elements by a **keycolumn NOT NULL** construct in our SQL queries[25], since its results are equal to those of an existence test and it costs nearly nothing. This is important since we do not want to search for any other condition or a constellation of other conditions that implicitly contains the desired existence test. And if the test is cheap, we can safely connect our existence test to *every* node without fearing a major performance penalty. The resulting SQL query is shown in Figure 5-26.

```
SELECT Company.Name
FROM ((Company LEFT OUTER JOIN Division) LEFT OUTER JOIN Employee)
WHERE Division.ID NOT NULL
AND Employee NOT NULL
```

Figure 5-26 *SQL: the names of all companies for which at least one division as well as at least one employee is stored*

5.4.5 Translating Cross Products

Until now, we were able to create SQL statements that only require one occurrence per table in the **FROM** clause. But there are also cases, where this simply does not work, for instance for cross product queries. Assume, we want to process a query like the one in Figure 5-27. This query does not only select customer names who have been sent an order on the same date as it has been received (like the wrong translation in Figure 5-28). Instead, this query selects the names of all customers who have been send an order on a day we received any (other) order from them. The corresponding SQL query looks like Figure 5-28.

```
/taker[order/@sendDate=order/@receiveDate]/@name
```

Figure 5-27 *XPath: cross product example*

[24] Indeed, 'step' is the term used in the XPath standard for a segment of a location path expression.

[25] The existence test for an attribute node is obviously **correspondingcolumn NOT NULL**

```
SELECT Taker.Name
FROM Taker LEFT OUTER JOIN Order
WHERE Order.sendDate=Order.receiveDate
```

Figure 5-28 *SQL: wrong translation of the cross product example*

```
SELECT Taker.Name
FROM ((Taker LEFT OUTER JOIN Order o1) LEFT OUTER JOIN Order o2)
WHERE o1.sendDate=o2.receiveDate
```

Figure 5-29 *SQL: correct translation of the cross product example*

Now we could imagine that if a node occurs twice in the XPath query expression, its corresponding table also had to occur twice in the `FROM` clause of the translated SQL statement. But this is not true, as the query from Figure 5-30 (it contains the parent of `item` twice) demonstrates.

```
//item[../@receiveDate=../@sendDate]
```

Figure 5-30 *XPath: common parent*

This query indeed selects all items, that have been send on the same date as the order has been received. If a certain `item` instance is checked whether or not it is in the result set, its predicate is evaluated. The two operands within this predicate both refer to the parent of the current instance. This is in both cases the same node. In a translation of this query, the order table may thus occur only once in the `FROM` clause. The resulting SQL query is shown in Figure 5-31.

```
SELECT Item.IID
FROM (Taker LEFT OUTER JOIN Order) LEFT OUTER JOIN Item
WHERE Order.receiveDate=Order.sendDate
```

Figure 5-31 *SQL: common parent*

These cross product queries and absolute location paths (predicates containing absolute location paths form a special kind of cross product queries) are the reason, why we cannot use the `ProxyNode`s as operands to our `Expression`s, even though we process the result of our query on their structure. The concept to solve this problem is presented in Section 5.7.

5.5 *The Setup*

The mapping between the relational elements and the elements of the virtual document has to be externally defined. The structure of the virtual document as well as the connections between the tables and columns and elements and attributes have to be established. This metadata is stored in and provided through what we call the *Setup*.

5.5.1 Basic Considerations

The Setup needs to contain detailed information about the relational database schema, at least for the cantle that should be mapped. As well, the Setup contains all information needed for generating an XML Schema Document (XSD) describing the virtual document. If all available data for both domains is combined in one Setup, this document can be used to extract the relational as well as the XML schema. This is illustrated in Figure 5-32.

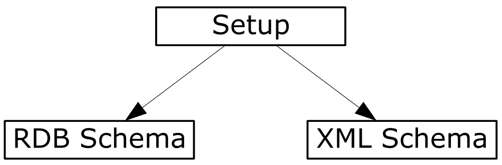


Figure 5-32 *Extracting Schemas from the Setup*

On the other hand, with the aid of some additional data, the Setup can be created from the database schema automatically. The missing XML schema can then be extracted. Also, Figure 5-33 shows the other way round: The generation of a Setup from an existing XML Schema. In this case, the database schema is extractable.

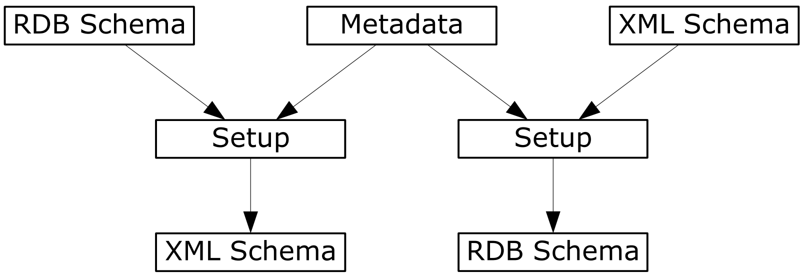


Figure 5-33 *Providing one Schema, creating the Setup and extracting the other one*

5.5.2 Requirements

In detail the Setup has to provide

- Elements, their attributes, and their structure
- Tables and their columns
- A primary key for every table
- A foreign key (see Section 5.5.3) for every table represented by an element that is a child of an other element. The foreign key must reference a key of the table corresponding to the parent element.
- The mapping from elements to tables and from columns to attributes[26]

[26] We require this to be consistent, i.e. an attribute does not refer to a column that belongs to a different table than the one represented by the containing element

- The data types of the columns have to be provided to adapt the query syntax if necessary (especially for type casts).

We confine ourselves to these basic functions here. Apart from them, one could think of conditions that could automatically be applied to table queries (like **`Boss IS NULL`**) or always joined tables, for instance. We regard this as unnecessarily complex to implement since it can very easily be realized by dint of (relational) views. Whether our 'tables' refer to true tables or just to views does not matter in this read-only approach.

5.5.3 Asymmetrical (n:1) Relations

With the introduction of the Extended Example in Section 5.2, we meet a problem with the modeling of an **`order`**'s **`item`**s. There are two possibilities how the content could be provided. They are both shown in Figure 5-34.

Version A is the one we used in Section 5.2 without dwelling on it. An **`order`** contains **`item`** elements which themselves contain a **`product`** attribute. The value of this attribute is the string used in some **`product`**'s id attribute. The functionality is equal to an ID/IDREF pair, but since we do not require any elements to be unique within in the virtual document (see Section 5.2), we cannot use mapped attributes as an ID attribute that may be referenced by an other node via IDREF.

Version B indeed makes use of the fact that no element within the virtual document has to be unique. It first maps the product nodes corresponding to the item's product attribute beyond the order nodes. Then it maps the companies that produced the products beyond those product nodes. The resulting queries (see Figure 5-35) would look a lot more pleasing than the ones of version A (see Figure 5-36).

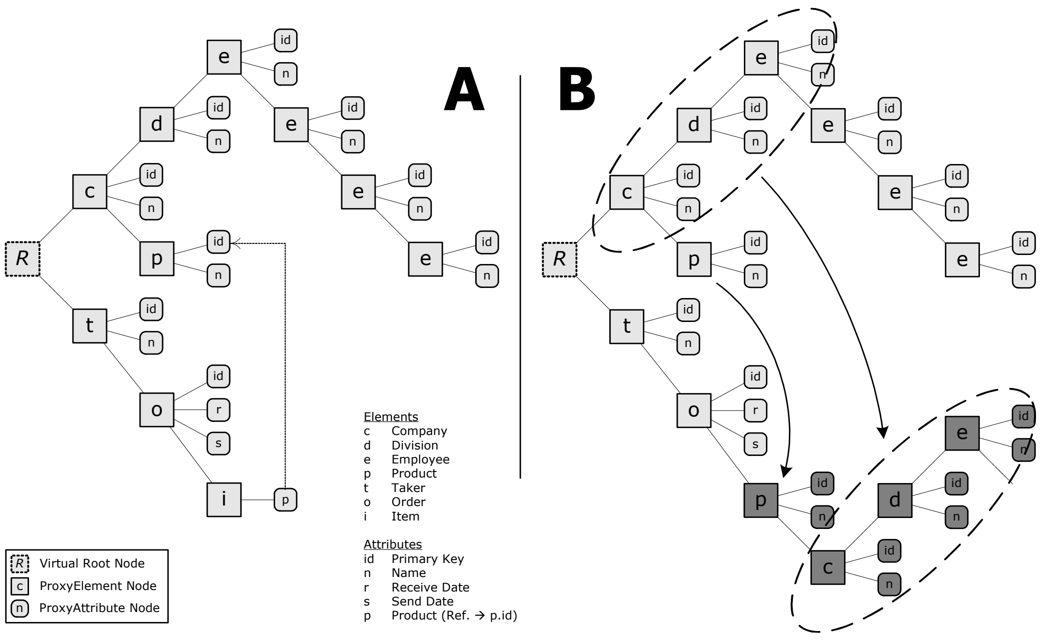


Figure 5-34 *Linking (A) vs. Inlining (B)*

```
/taker[@name='Otto Meier']/order/product/company/@name
```

Figure 5-35 *XPath query for the inline draft (B)*

```
/company[product/@id=/taker[@name='Otto Meier']/order/item/@product]/@name
```

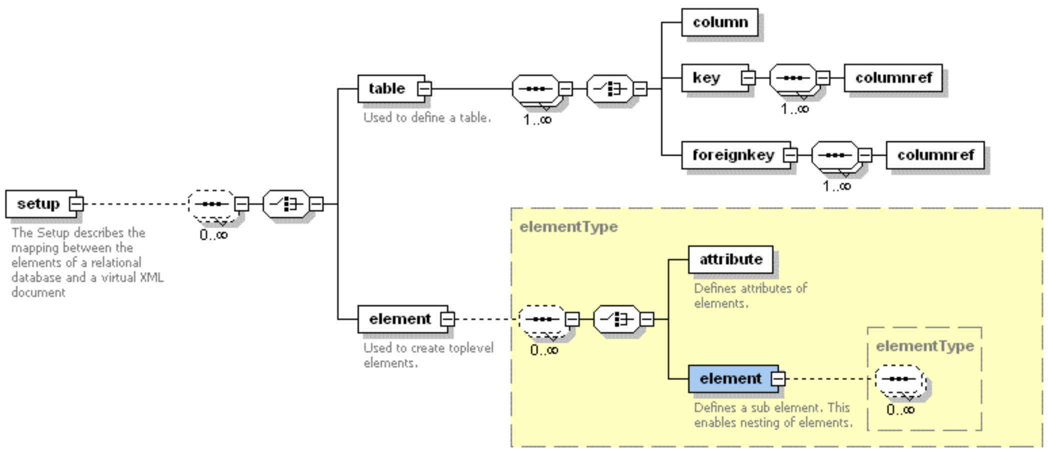
Figure 5-36 *XPath query for the linked draft (A)*

The reason for not choosing the much more elegant inlining is not the fear of the heavy redundancy in the virtual document. It is also not difficult to find out the node types of the ancestors of companies selected via a query like the one in Figure 5-35. But is impossible to determine one distinct parent instance (that would be of the type **product**) for a **company** instance that offers multiple products. We already hinted at this problem talking about the **getParent()** implementation in Section 5.3.

Consequently, we may state, that only key/foreign-key relations (or likewise asymmetrical (n:1) relations) may be mapped to nested elements. All other kinds of relations have to be emulated in the XPath query like the equi-join from the query in Figure 5-36. This is not very pleasant but inevitable, since this dues to a conceptual peculiarity.

5.5.4 Example

According to Section 5.5.1, there are multiple ways to obtain a Setup starting from the manual creation by program code or a definition via an XML document up to an automatic generation from a database schema. An example XML Schema according to Figure 5-37 can be found in Appendix A, the setup document for the Extended Example from section 5.2 can be found in Appendix B.



setup
The Setup describes the mapping between the elements of a relational database and a virtual XML document
0..∞
table
Used to define a table.
1..∞
column
key
1..∞
columnref
foreignkey
1..∞
columnref
element
Used to create toplevel elements.
0..∞
elementType
attribute
Defines attributes of elements.
element
Defines a sub element. This enables nesting of elements.
elementType
0..∞

Figure 5-37 *An example XML Schema for the Setup*

5.6 *Expressions*

Declarative query languages select certain information items without stating how to obtain them. To achieve this goal, they provide a mechanism to select classes of information items coupled with a condition the instances of this classes have to fulfill to be in the result set. The condition is a Boolean expression within which the instances are used as operands. If the expression evaluates to true for a certain instance, this instance will be in the result set.

Certainly, the declaration of conditions cannot use the instances themselves as operands, since they are as likely as not unknown to the enquirer. Instead, the classes are used as proxies for their instances as operands in the condition.[27] The query execution engine must be able to resolve these expressions to evaluate the conditions.

SQL as declarative query language uses a **SELECT-FROM** clause to describe the class of the resulting information items and a **WHERE** clause defines the condition. XPath queries on the other hand are also structured alike. Unfortunately that is not as obvious, a tribute to their recursive tree design. The location paths select the classes of nodes to return, the predicates of the path steps from the condition[28]. But if a path is a sub expression of an other path, for instance in a step's predicate, the resulting nodes are not returned to the user. Instead, they are used as operands

[27] This is comparable to the use of residue classes in modulo arithmetic expressions.
[28] Every condition on the path must have been fulfilled in order to return a node

within the super expression. Often, implicit casts especially to the data type Boolean are used here.

If now we created a binary operation named 'SELECT_IF' (σ) with an operand expressing the class of information items to output and an other operand defining the condition an instance from that class has to fulfill to be output, we could express queries of both languages in one single expression. This expression only consisted of operands and operations, resulting in no additional constructs like `SELECT-FROM` and no implicit 'reuse-or-output' semantics any more.

The goal is to try to make the concept 'expression' center of the object model for a common query representation. Every translation process needs such a model to perform major or minor adoptions[29]. Further on, there are several dialects of SQL. An abstract query representation allows us to simply use of different serializations to support a wide range of relational database management systems instead of recreating the whole query translator again every time.

5.6.1 Matrix Operands

The object model of the expressions is probably even more important to the translation algorithm than the one from section 5.3. But it is by far more obvious and thus easier to understand. The only thing we really have to take into account is that XPath allows operands in its expressions that are one order of magnitude more powerful than the ones allowed in SQL. Both allow scalar operands (like constants) and vector operands representing a class of information items (the 'normal' operands like column names in SQL expressions). XPath in addition allows a kind of *matrix operands*: sets of classes of information items.

Figure 5-38 shows a query that in conjunction with our example (see Section 1.2) contains such a matrix operand. The `.//@name` operand of the comparison within the condition refers to the company's name attributes (`/company/@name`), the division's name attributes (`/company/division/@name`), and the employee's name attributes (`/company/division/employee/@name`). All these are classes that themselves represent a set of instances.

```
/company[.//@name='Knox']/@id
```

Figure 5-38 *An XPath query with a matrix operand*

[29] For instance overcome paradigm gaps (section 3.1) or to customization of the query for a special Setup as described in section 5.4.3.

Obviously, the resolution of matrix operands is a point where the translation process has to make the query specific to one particular Setup. For a different Setup, the `.//@name` would have referred to different nodes than the above for sure.

5.6.2 Unnesting Matrix Operands

Since SQL does not support matrix operands, they need to be broken down to vector operands to issue a valid SQL query. The SQL query processor in turn will then take care of those and break them down to scalar operands by replacing them with the respective instances to evaluate the condition for us.

For simplicity, we are going to abstract to 'classes' and 'instances' in the following, knowing that the classes represent the matrix operands and the instance represent the plural operands.

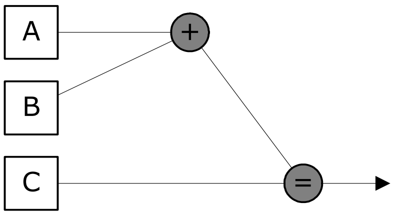


Figure 5-39 *Example Expression containing 3 Matrix Operands, none resolved*

Imagine, we have a condition like the one in Figure 5-39. Assuming that the class **`A`** is a proxy for two possible instances: **`a1`** and **`a2`**. The condition is evaluated to true if and only if either **`a1+B=C`** or **`a2+B=C`**. This is shown in Figure 5-40. The algorithm therefore is of the complexity O(n).

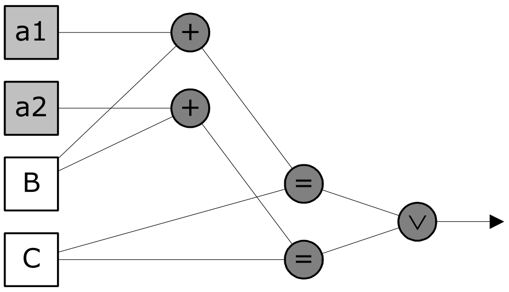


Figure 5-40 *Example Expression containing 3 Matrix Operands, first resolved*

The next step is to resolve **`B`**. **`B`** again consists of to instances **`b1`** and **`b2`**. Consequently we must extend the expression to **`a1+b1=C`** ∨ **`a2+b1=C`** ∨ **`a1+b2=C`** ∨ **`a2+b2=C`**. This is shown in Figure 5-41. Obviously, the algorithm is already O(n^2).

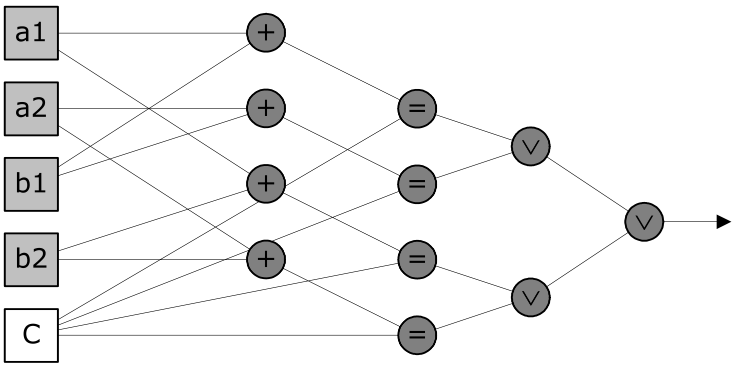


Figure 5-41 *Example Expression containing 3 Matrix Operands, second resolved*

Finally we are going to break **C** down. Again, the expression has to be evaluated for every possible instance. Again, the condition is fulfilled if any of the **x+y=z** terms may be evaluated to true. Figure 5-42 shows the resulting expression of the complexity of O(n^3).

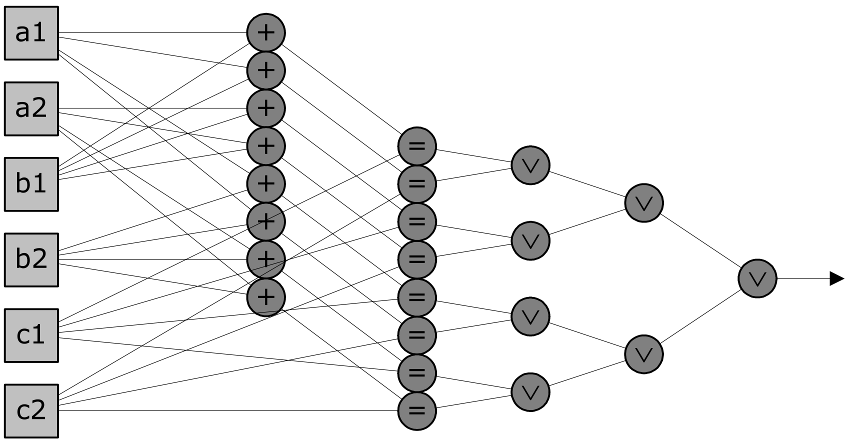


Figure 5-42 *Example Expression containing 3 Matrix Operands, all resolved*

Unfortunately we can not make use of the fact that only one of the comparisons has to be true by performing a lazy evaluation. This dues to the fact that for the translation process the instances are vectorial values of which the translator does not know any scalar values to truly evaluate them. This optimization can only be done by the SQL query processor after we passed him the expression of the complexity of O(n^s) whereas s is the number of matrix operands within this expression.

The naïve approach as hinted in Figure 5-42 copies the whole expression for every instance that should be inserted. Then it replaces the matrix operand by a distinct instance within each copy. The last step is to connect the resulting expression by OR. To save a little effort, we could reuse equal terms as shown in Figure 5-43, thus not requiring to save all O(n^s*m) operation nodes (m the number of operations in the original condition), or copy them if an other matrix operand would have been to resolved. The resulting condition string will contain all O(n^s) terms each consisting of O(m) operations, though. We can formulate a rule of thumb for the order of resolution of matrix operands: the deeper, the earlier.

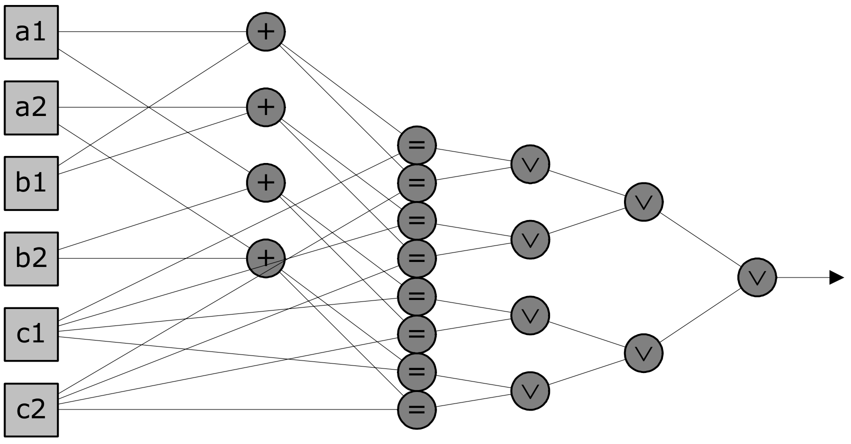


Figure 5-43 *Example Expression containing 3 Matrix Operands, all resolved, using common sub expressions*

5.6.3 The Expression Object Model

As already mentioned, the Expression Object Model is pretty straight forward. Due to the recursive structure of XPath expressions, an expression may be an operand to an operation in a superior expression. Thus, there actually is only an **`Operand`** object in the object model and no **`Expression`** object at all. The terms 'expression' and 'operand' may be used as synonyms within the Expression Object Model.

The root object of this object model is the **`Operand`**. Due to the recursive nature, everything may be an operand to any other thing. There are two types of **`Operand`**s: **`InnerOperand`**s and **`OuterOperand`**s. The **`InnerOperand`**s are those containing other **`Operand`**s, **`Operation`**s, and **`OperandSet`**s, particularly. They may receive reorganizing commands through the **`optimize()`** function and must provide a **`copyAndReplace()`** function for the resolution of the matrix operands (see section 5.6.2).

`OuterOperand`s on the other hand are **`Operand`**s that either do not need further processing or where the translation process is just not able to provide it. In particular, these are **`Constant`**s (like **`NULL`**, **`TRUE`**, **`FALSE`**, **`NAN`**), **`Variable`**s (like 'FZI' or '5'), and **`ProxyNode`**s. **`ProxyNode`** is the super class of **`ProxyElement`** and **`ProxyAttribute`**, known from the object model of section 5.3. (This is a bit inaccurate as we will see in Section 5.4.5. But until we have not explained the problem presented there and also not the concepts from Section 5.7 intended to overcome them, it does not make sense to already introduce them here. This is also the reason, why the corresponding node is marked by diagonal lines in the UML diagram from Figure 5-44 representing the Expression Object Model.)

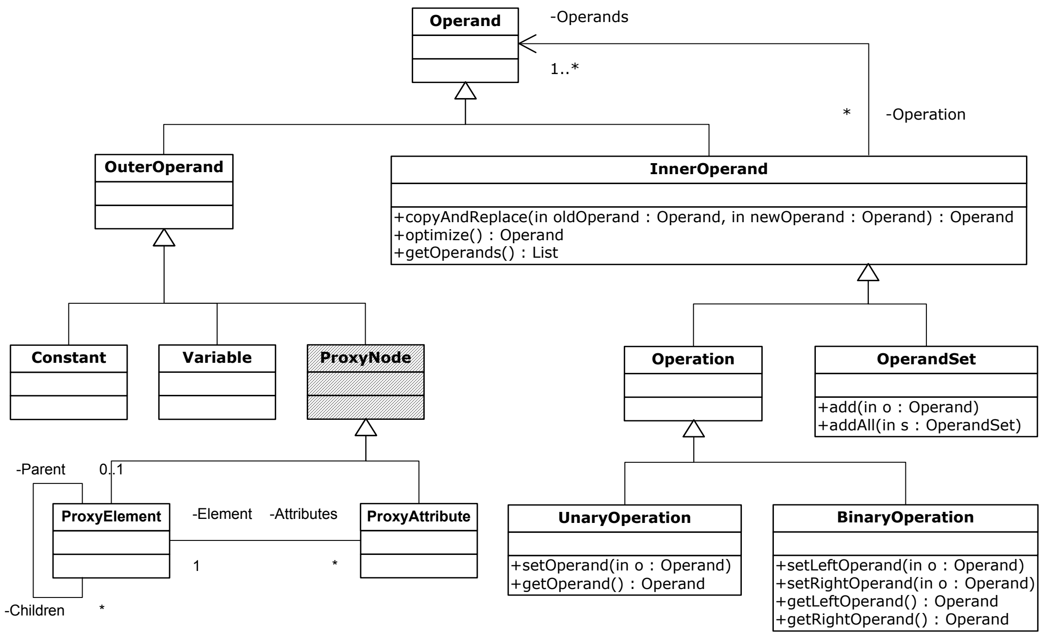


Figure 5-44 *UML diagram for the Expression Object Model*

The **copyAndReplace()** function does what it's name suggests. First, it copies the current **Operand**. If none of its operands is the provided **oldOperand**, it sets the operands of the copy to the results of the **copyAndReplace()** function call of every **Operand** of the original **InnerOperand**. If one of the original **Operand**s is the searched one, the copy's operand will be set to the provided **newOperand**. Whether or not the other operands are recursively copied or just adopted depends on the replacement strategy. If we want to utilize the optimization mentioned in section 5.6.2, we just need to adopt the rest of the operands.

The **optimize()** function's implementation heavily depends on the very special type of **InnerOperand**. But there is one thing in common: If an **InnerOperand** receives an **optimize()** function call, it should return an optimized equivalent to itself. This may either be the same instance with just a couple of optimized operands (recursiveness!), or it may also be a completely new operand. Two examples can be seen in Figure 5-45 and Figure 5-46.

```
optimize() for ADD, a binary Operation:

Operand left = getLeftOperand();
if (left instanceof InnerOperand)
{
    left = ((InnerOperand) left).optimize();
    setLeftOperand(left);
}
Operand right = getRightOperand();
if (right instanceof InnerOperand)
{
    right = ((InnerOperand) right).optimize();
    setRightOperand(right);
}
if ((left instanceof FALSE) or (right instanceof FALSE))
{
    return Constant.FALSE;
}
if (left instanceof TRUE)
```

```
{
    return right;
}
if (right instanceof TRUE)
{
    return left;
}
// no further optimization
return this;
```

Figure 5-45 **`optimize()`** *implementation for the AND operation*

```
optimize() for an OperandSet:

For Each o In getOperands()
{
    if (o instanceof InnerOperand)
    {
        Operand optimized = ((InnerOperand)o).optimize();
        this.replace(o, optimized);
    }
}
// remove NULLs, optimize() might have returned NULLs
For Each o In getOperands()
{
    if (o instanceof NULL)
    {
        removeOperand(o);
    }
}
// optimizations count==0 or count==1
if (getOperands().count() == 0)
{
    // do not keep an empty set
    return Constant.NULL;
}
if (getOperands().count() == 1)
{
    // do not keep set if only one operand
    return getOperands().first();
}
// no further optimization
return this;
```

Figure 5-46 **`optimize()`** *implementation for an Operand Set*

5.6.4 Double Tree Illustration

In this section, we will concentrate on the quirk that XPath expressions cope with two trees, in fact. First, there is the tree for the expressions itself with its operations and their operands. Second, the operands refer to the elements of the XML document tree.

For the graphical representation of expressions within this thesis, we choose two trees facing at each other. On the one hand, there is the operation tree in which a child of a node represents an operand to an operation. Such a tree has already been shown in Figure 5-43, for instance. It needs to be read from its root. The second tree, on the other hand, represents the template document. According to the Document Object Model (DOM), this may also be represented by a tree. The template document also needs to be read from its root.

Both trees are connected through pointers from the operation tree side to the template tree side. These pointers illustrate that the corresponding node shall be used as an operand to that operation. Further on, if constants are used within one expression (for example for comparison with a string or a number), these will be represented by just one single node, that has no further connections. An example can be seen in Figure 5-47 in which the SELECT_IF operation (see introduction of Section 5.6) is presented by the node 'σ'. It shows a query for an employee's name whose ID equals 5.

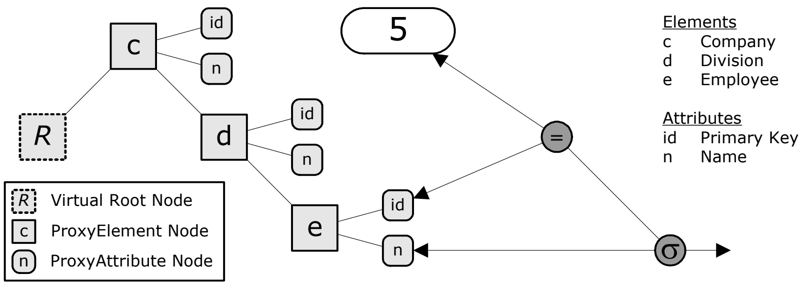


Figure 5-47 *Double Tree Representation of* **`/c/d/e[@id=5]/@name`**

Actually, there is some additional hidden semantic in this diagram: The query only makes sense if we require the equality operation's 'id' operand and the SELECT_IF operation's 'name' operand to refer to the same employee instance. Thus, if multiple operands within the template tree are referenced, we require instances to cover at least the minimal spanning sub tree containing all of the referenced operands. 'Instances' in this case refers to fragments of the virtual document, for example one single virtual node or a set of virtual nodes forming the desired tree structure. A more clarifying example for the resulting structural demands is shown in Figure 5-48. The query outputs the names of all employees of the company named 'FZI'. If we did not require a relationship between the company and the employee here, the query would return all employee's names if any company in the database had the name 'FZI'.

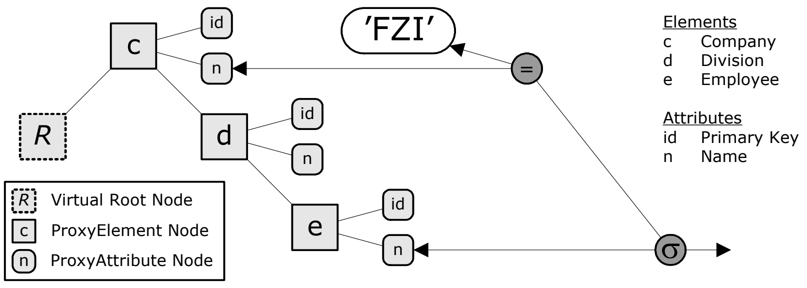


Figure 5-48 Double Tree Representation of **`/c[@name='FZI']/d/e/@name`**

5.7 *The Path Tracing Graph (PTG)*

As we have already learned from Section 5.4.5, we might require to map an element multiple times within our query. In this section, we want to concretize the cases when this is necessary and introduce a new data structure managing the references to the **ProxyNode** objects. We call this data structure *path tracing graph* (PTG), since we build it by recording the traces of our process through the nodes of the template tree. The PTG will also ensure that we only use a minimal spanning tree of the information items we require to access within the query.

5.7.1 Clear Position within the Template Document

Assuming we currently have a particular node v somewhere in the middle of the template tree. The particularity of the node may for instance due to the fact that the user passed our query processor one certain node instance as context node for the path processing. Such an instance obviously has one defined representative in the template tree. The particularity may also due to constraints imposed to it by the processing of previous steps. In this case, there may be multiple nodes meeting these constraints. If so, we simply treat them one by one. Thus we can assume, that for every processing state, we can define one distinct position within the template document.

5.7.2 Moving around in the Template Document

There are two ways to navigate to an other node starting from v. The first is to further process the current location path expression, the second is to step into the predicates. This movements is what we are going to record with the Path Tracing Graph.

For the PTG, it does not matter, which way actually has been chosen for navigation. The axis of the next step (may this either be the next step in the current location path or the first step in one of its predicates operands) is relevant, and whether or not the next step is relative to v at all is of even greater importance. Namely for predicates it is possible to replace the current context node by the root node, since they form a new location path which may be absolute. Nonetheless, this shall not hinder us not to distinguish between the next step of the current location path and the first step of the predicates operands (at least for the utilization trace) – the next step of the current location path just will never be absolute.

5.7.3 An Example for Building the Path Tracing Graph

Let v be the taker (t) node within the template document of our Extended Example from Section 5.2. We are now going to process the query from Figure 5-27 in detail. Stepping into the predicate, we have to translate two terms to evaluate the comparison. The first term is the **`order/@sendDate`** which consists of two relative steps. The first step (S1.1) selects a child node with the name 'order', the second one (S1.2) selects attributes of such nodes with the name 'sendDate'. Walking this path, we build our Path Trace Graph step by step. This is shown in Figure 5-49.

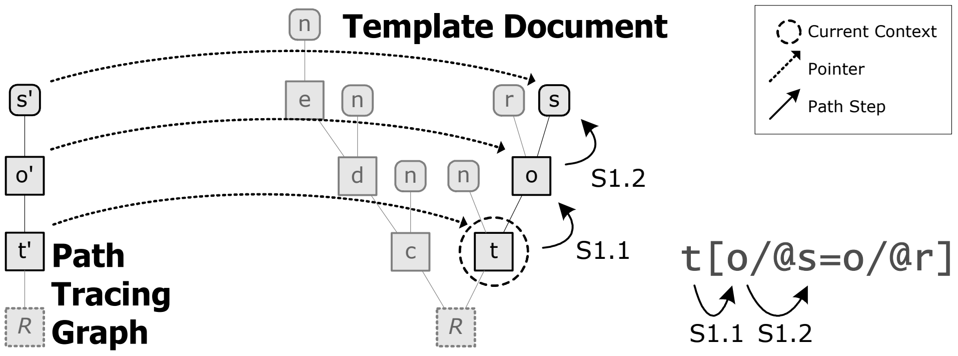


Figure 5-49 *Building the Path Tracing Graph (first term)*

The second term, **`order/@receiveDate`**, has the same first step. But we may, according to Section 5.4.5, not use the same order nodes for further processing. Instead, we have to create a second child node (o") to our context node (t') within our PTG, pointing to the same schema node (o) as the already existing child node (o'), but expressing the independence of the node instances in both sets. Then, we can correctly express our condition as shown in Figure 5-50.

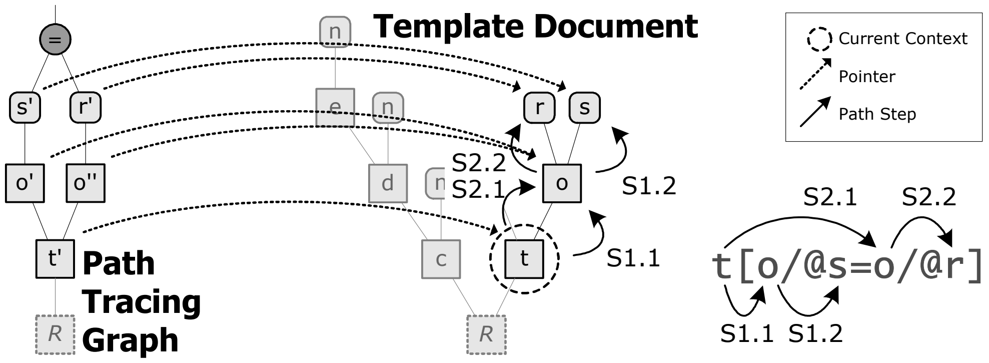


Figure 5-50 *Building the Path Tracing Graph (second term)*

The reason why we need a second branch within our PTG is the resolution of plural operands within the query executor. A **`ProxyNode`** o, respectively its corresponding table, represents *all* its possible instances, but within the XPath language as well as in SQL, as an operand to an expression, o represents *only one* instance at a time (but each during the whole processing).

5.7.4 Navigation along the Axes

This section shall clarify how the Path Tracing Graph reacts to the occurrence of new nodes relatively above or relatively below the position of the current context node. Our considerations may be grouped into four categories: ascendant nodes (`parent`, `ascendant`), descendant nodes (`child`, `descendant`), attributes (`attribute`) and the containment of the self axis (`self`, `ancestor-or-self`, `descendant-or-self`).

5.7.4.1 Navigation to Ascendant Nodes

In Section 5.4.5, also a second problem has been addressed: the navigation to ascendant nodes. For the navigation in the direction of the root node, it is unnecessary to create multiple branches. Multiple references from a certain node within a tree to one particular of its ancestors (either always the parent node or always the grandparent node) will always return one single instance – and that instance is always the same. Concluding, ascendant nodes do not cause the PTG to create new branches, they just reuse their former instance.

5.7.4.2 Navigation to Descendant Nodes

As we have seen in Section 5.7.3, nodes that are descendant to the current context node cause the PTG to create a new branch starting at the current context node and ending at the descendant node. It is important to mention that also the branch on which a node lies is part of the context information for the steps relying on that node. These nodes traces must 'grow' on the same branch as their origin. This can also be read out of Figure 5-50.

5.7.4.3 Navigation along the Self Axis

Since the self axis is defined to select a single node (see [Kay01], p. 365), the origin node itself, it acts like an ascendant node to the Path Tracing Graph (see Section 5.7.4) and will not create new branches. This is along with the ancestor-or-self and the descendant-or-self axes only true for this special node on the self axis (if any). The rest of the nodes (if any) must be treated according to Section 5.7.4.1 or to Section 5.7.4.2, respectively.

5.7.4.4 Navigation to Attributes

Even though the attributes are always depicted as descending nodes in the figures of this thesis, they do not cause the PTG to create new branches. The attribute axis is always applied to its origin element, only. Since an element can only have one value for an attribute (or none), the attribute is unique from the point of view of the origin node (which is our current context node). So every reference to the class of this attribute that passes the current context node, will always return the same instance. Consequently, the PTG will not branch for attributes. It may have branched to reach the context node though, see Figure 5-50.

5.7.5 Transformation of the PTG to SQL

The Path Tracing Graph forms a minimal spanning tree of all operands used in the superior expression. In relational terms, it can thus be regarded as a kind of universal table, at least for the fragment of the databases mini world[30], our query covers. If the PTG is serialized with each of its nodes receiving a unique name within the `FROM` clause of the SQL queries we produce, we can simply use these operands in the `WHERE` clause. In this case we do not even need to think about the paradigm gap described in Section 3.1.1 anymore.

To achieve this goal, we explicitly (most other approaches use implicit joins somewhere in the `WHERE` clause) build a `LEFT OUTER JOIN` tree from the PTG in the `FROM` clause. For every parent-child edge in the tree, we add a `LEFT OUTER JOIN child` to what we already have (also containing the parent). The (always virtual) root node must be omitted, instead we use the first level nodes as 'seeds' for our growing join trees. The first level nodes then coexist without mutual interference, thus constituting a cross product.

The statement on the existence without mutual interference is only true within the `FROM` clause, of course. Multiple first level nodes indicate multiple absolute location paths. This is only possible within expressions or predicates, thus the two 'independent' trees will in any case be connected by some operation within the `WHERE` clause of the generated SQL statement.

An example of a serialized PTG can be seen in the following example. The query `/company[.//employee[@name='Kazakos']/employee]/@name` obtains all company's names that have an employee with the name 'Kazakos' who is the boss of an other employee. The `FROM` clause of the corresponding SQL statement in Figure 5-51 shows the transformed PTG. If we would output the PTG itself, the results looked like Figure 5-52, or even closer to the tree-like structure of XML documents in the nested illustration of Figure 5-53.

```
SELECT c.Name
FROM COMPANY c
LEFT OUTER JOIN DIVISION d ON T_DIVISION.Company = T_COMPANY.ID
LEFT OUTER JOIN EMPLOYEE e1 ON e1.Division = d.ID
LEFT OUTER JOIN EMPLOYEE e2 ON e2.Boss = e1.ID
LEFT OUTER JOIN EMPLOYEE e3 ON e3.Boss = e2.ID
WHERE (e1.Name='Kazakos' and e2.id NOT NULL)
OR (e2.Name='Kazakos' and e3.id NOT NULL)
```

Figure 5-51 *SQL query with serialized PTG*

[30] for the definition of the term *mini world* see [LaLo95], Section 2.2.1

/c/@name	/c/d/@name	/c/d/e/@name	/c/d/e/e/@name	/c/d/e/e/e/@name
FZI	DBS	P.C. Lockemann	Wassili Kazakos	Alexey Valikov
FZI	DBS	P.C. Lockemann	Wassili Kazakos	Andreas Schmidt
FZI	DBS	Wassili Kazakos	Alexey Valikov	
FZI	DBS	Wassili Kazakos	Andreas Schmidt	
FZI	DBS	Alexey Valikov		
FZI	DBS	Andreas Schmidt		
FZI	PROST	Gerhard Goos	Benedikt Schulz	Thomas Genßler
FZI	PROST	Benedikt Schulz	Thomas Genßler	
FZI	PROST	Thomas Genßler		
FZI	SWT	Walter F. Tichy	Andreas Judt	Alexander Christoph
FZI	SWT	Walter F. Tichy	Andreas Judt	James J. Hunt
FZI	SWT	Andreas Judt	Alexander Christoph	
FZI	SWT	Andreas Judt	James J. Hunt	
FZI	SWT	Alexander Christoph		
FZI	SWT	James J. Hunt		
Siemens	A&D AS IT	Alfred Schmit		
Siemens	A&D AS PAS			

Figure 5-52 *The 'universal table' build for the example query*

/c/@name	/c/d/@name	/c/d/e/@name	/c/d/e/e/@name	/c/d/e/e/e/@name
FZI	DBS	P.C. Lockemann	Wassili Kazakos	Alexey Valikov
				Andreas Schmidt
		Wassili Kazakos	Alexey Valikov	
			Andreas Schmidt	
		Alexey Valikov		
		Andreas Schmidt		
	PROST	Gerhard Goos	Benedikt Schulz	Thomas Genßler
		Benedikt Schulz	Thomas Genßler	
		Thomas Genßler		
	SWT	Walter F. Tichy	Andreas Judt	Alexander Christoph
				James J. Hunt
		Andreas Judt	Alexander Christoph	
			James J. Hunt	
		Alexander Christoph		
		James J. Hunt		
Siemens	A&D AS IT	Alfred Schmit		
	A&D AS PAS			

Figure 5-53 *The 'universal table' with visualized nesting*

5.8 *The Core Algorithm*

The algorithm splits into three parts. The first part is to build the expression objects from the XPath query string. This step includes parsing, of course. The second step is to serialize the expression to one or many SQL query strings that may be executed against the database. And last, but not least, we have to create instance nodes from the resulting rows.

5.8.1 Building the Expression

Note: This Section presents the core algorithm. It really is not difficult to understand but it is a bit bulky to explain. Thus it will be developed step by *step to relieve the understanding. We also suggest to recall Section 5.3, especially to Figure 5-11.*

XPath queries are written by the enquirer in string form. Thus we will receive a string that first needs to be parsed for being processed any further. Such parsing can be implemented using JavaCC or likewise tools. We will not go into details of this task and expect the user to obtain a sensible object representation of the string query by himself. This object model just has to provide apparent functions like access to the location path steps, to the axes, to the predicates, and so on.

Assuming we have parsed the XPath query string, we can start to build the Expression. Lately, we want to obtain several node instances that fulfill the query conditions. We do this by simulating a naïve XPath processing approach analogous to the XPath definition. We start at the context node and process the path expression step by step, calculating intermediate results (without materializing the instances, of course). The intermediate results are used as context nodes for the next step. If no context node has been provided, it defaults to the root node. The nodes that remain from the last step are to be returned as the result.

5.8.1.1 Processing a Step

To process one step, we need to do the following: Starting at a certain node, our context node, we first select all the nodes on the axis provided by the current step into a set. Afterwards, we discard all nodes that do not match the node test (which may either be a name test or a node type since we do not support processing instructions) from this set. We cannot do this on the instances since we do not want to materialize them, at least if they are not the result of the whole XPath query. But we are able to work on the template document, since if **`ProxyNode`** meets these requirements (axis and node test), all the instances it represents also meet them. They all have the same name respectively the same node type like the proxy node and they are structurally at the same position as the proxy node, thus also fulfill the axis requirement. This is hinted in Figure 5-54.

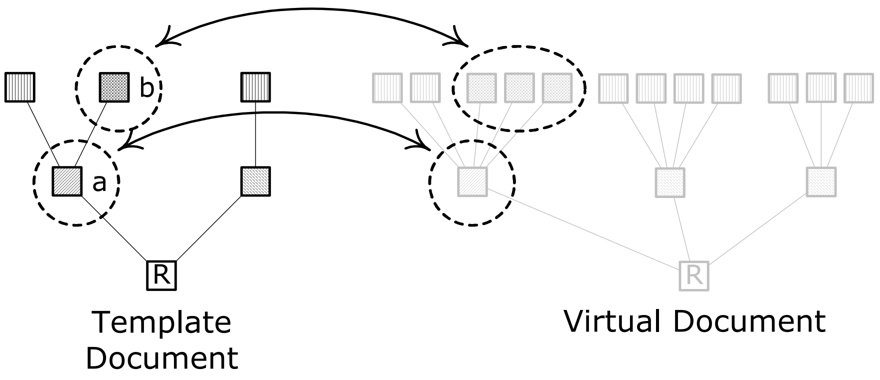


Figure 5-54 *Working on the* **`ProxyNode`***s*

The next thing to do is to apply the predicates. Since we operate on proxies and do not know any instances, we cannot evaluate them. We cannot select those that are kept and those that are removed from the result set. But we can at least describe which instances represented by the proxies would remain in the set by concatenating a condition to each proxy node. For now, we expect to have a function capable of processing a predicate for us and return a Boolean expression. This expression is what we call a *condition*.

A node may have to fulfill several predicates to be in the result set. Thus, the conditions have to be combined to one condition by an **AND** chain (or tree, but the chain is simpler to implement). Summing up, our algorithm looks like Figure 5-55 so far.

```
FUNCTION processStep (
    IN (NodeOperand contextNode, Operand contextCond);
    IN Axis currentAxis;
    IN NodeTest currentNodeTest;
    OUT Set[(NodeOperand contextNode, Operand condition)];
)
 // apply axis
Set nodesOnAxis = contextNode.getProxyNodesOnAxis(currentAxis);
// apply nodetest
Set filteredNodes = nodesOnAxis.filterNodesThatMatch(currentNodeTest);
Set resultSet = new Set[(contextNode,condition)];
// apply predicates
For Each Node n in filteredNodes
{
    Operand condition = Constants.TRUE; // no predicates means no restriction
    For Each Predicate currPredicate in currentPathStep.getPredicates()
    {
        XPathExpression expr = currPredicate.getExpression();
        Operand predicate = processPredicate(currentNode, expr);
        condition = new And(condition, predicate);
    }
    Tuple t = (contextNode,condition);
    t.setContextNode(currContextNode)
    t.setCondition(condition);
    resultSet.add(t);
}
return resultSet;
END FUNCTION
```

Figure 5-55 *Process one step (first approach)*

Something that we have to take into account now is the need for path tracing (see Section 5.7). Since we cannot operate on instances, we need to work on proxies.

But according to the insights of Section 5.7 we cannot use the `ProxyNode` objects themselves as operands to the expressions, since we might lose the knowledge about the independence of multiple node sets, if both are represented by only one `ProxyNode`. Consequently, we use the nodes of the Path Tracing Graph as operands for the expression. These nodes point to the `ProxyNode` they belong to, thus transitively to the instances. They may be considered as proxies for proxies or, for the core algorithm, simply as proxies for the corresponding instances.

Further on, the PTG attends to procure us with the necessary data. Its serialization creates variables we simply can use within the whole expression. All we need to do here is to provide the necessary information to the PTG module to enable it to build its internal data structure. This information is the context node, the axis and the next node to process.

A thing we have to heed is the dependence of a step's result nodes on the context node for a proxy algorithm. This is not yet implemented correctly. The above will not work correctly for a certain node within the document.

Assume we start at a certain virtual node which is mapped to `ProxyNode` a within the template document. The next step looks like `child::*`. Starting from `ProxyNode` a, selecting all children with any name, we meet `ProxyNode` b. This node, according to our definition of a `ProxyNode`, represents all instances that are children of any instances represented by `ProxyNode` a. This is shown in Figure 5-56. This is incorrect, the proper selection would comprehend the children of the initial virtual node only.

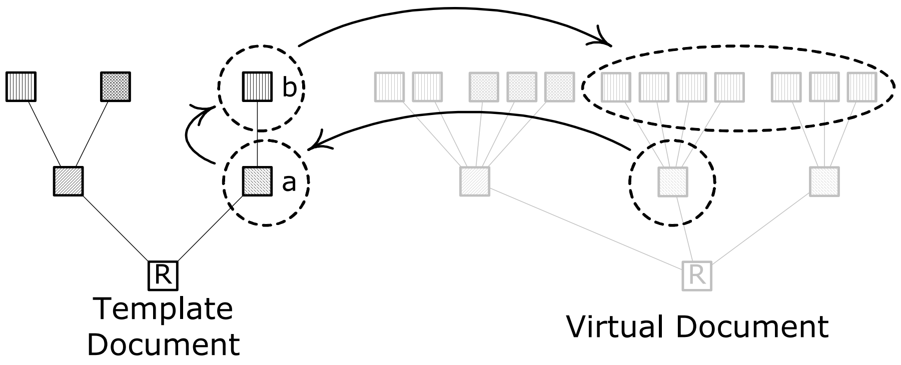


Figure 5-56 *Navigation on template violating the structural relation to progenitor*

To achieve the correct selection, our navigation would have to impose a restriction to the set of instances represented by `ProxyNode` b. Such a restriction may only be applied in the current context, not in general.[31] It has to limit the set to only con-

[31] An other reference to the `ProxyNode` may require its set of instances in an unrestricted manner.

tain nodes that have the desired structural relation to the selected ancestor[32] node as shown in Figure 5-57.

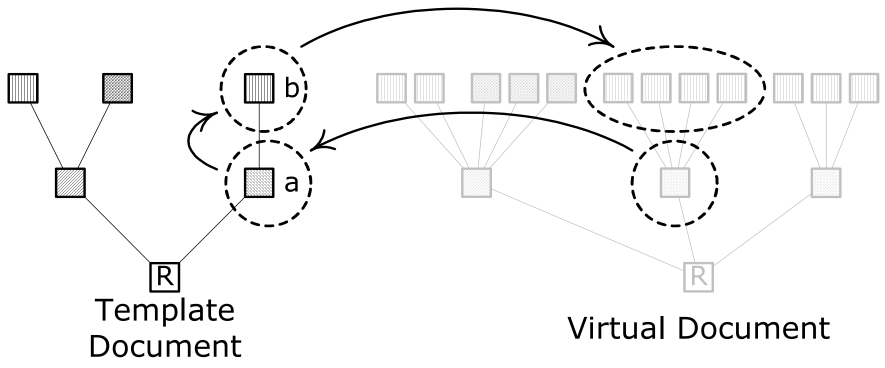


Figure 5-57 *Navigation on template regarding the structural relation to progenitor*

According to Section 5.4.4, we need connect an existence test to *every* node. Thus we can also expect the condition of the progenitor node to contain it. To archive the desired restriction, it is sufficient to append the condition of the progenitor node to the condition of each of the current result nodes. Afterwards, we can discard[33] the progenitor node, since we do not need it any more.

For a complete location path, the process looks as follows: Every certain step takes one proxy node in conjunction with a condition as context and returns its result proxy nodes also each with an attached condition. Every result node then is once context node for the processing of the next step. All the thus emerging new result nodes together form the new result set, the old result nodes are discarded. Only their conditions live on in the conditions of the new result nodes, and so on. The result nodes of the last step form the basis for the building process of Section 5.8.4, their conditions for the serialization process from Section 5.8.3.

One word at the attached conditions: They are a passing through parameter bound to the context node, not bound to the operands. Or, in other words, they need to be passed by value, not by reference. If they are bound to the operands, the processing of a query like the one from Figure 5-58 fails. May the parent p of node x have a certain condition p_p. The condition of x will then become $p_x{:=}p_p \wedge \exists(p) \wedge \text{true}$. The first term comes from the previous deliberations. We require the second term according to Section 5.4.4. The third term impersonates the predicates of the path step `child::x`. Since there are none, they are in any case fulfilled. Now we return to the parent node. The condition of p becomes $p_p{:=}p_x \wedge \exists(x) \wedge \text{true}$. If we pass a reference

[32] The term 'ancestor' does not refer to the ancestor axis of XPath here, but to the ancestor step's result nodes of our path expression. For clearness, we will use the term 'progenitor', instead. In this case, the progenitor node is the parent.

[33] Of course, the **ProxyNode**s are kept within the Setup, the traces are kept within the PTG and the operands stay within the expression. We are just done with it in the algorithm here and do not need it for further processing.

to p_x which itself contains a reference to p_p, we will have a hard time serializing the resulting expression $p_p=p_p \wedge \exists(p) \wedge true \wedge \exists(x) \wedge true$. With a reference by value semantic, we obtain $p_p=p_p^{(old)} \wedge \exists(p) \wedge true \wedge \exists(x) \wedge true$. Thus the existence of p still relies on the existence of p (hen-egg-problem), but this is due to our way of translating it (see Section 5.4.4) no problem. The corresponding SQL clause will look like Figure 5-59.

```
x/.. ( ≡ child::x/parent::node() )
```

Figure 5-58 *XPath query that will fail in pass by reference semantics*

```
... WHERE pp(old) AND p.KeyCol NOT NULL AND x.KeyCol NOT NULL
```

Figure 5-59 *Translation of the pass by ref. vs. pass by val. semantics*

The algorithm finally looks like Figure 5-60.

```
FUNCTION processStep (
     IN (NodeOperand contextNode, Operand contextCond);
     IN Axis currentAxis;
     IN NodeTest currentNodeTest;
     OUT Set[(NodeOperand contextNode, Operand condition)];
)
 // apply axis
Set nodesOnAxis = contextNode.getProxyNodesOnAxis(currentAxis);
// apply nodetest
Set filteredNodes = nodesOnAxis.filterNodesThatMatch(currentNodeTest);
Set resultSet = new Set[(contextNode,condition)];
// apply predicates
For Each Node n in filteredNodes
{
     // inclide path tracing
     NodeOperand currentNode = PTG.getNode(n, contextNode, currentAxis);
     // every node has the condition ‘must exist’ (see Section 5.4.4)
     Operand condition = new EX(contextNode);
     For Each Predicate currPredicate in currentPathStep.getPredicates()
     {
          XPathExpression expr = currPredicate.getExpression();
          Operand predicate = processPredicate((contextNode,contextCond),expr);
          condition = new And(condition, predicate);
     }
     // include context information
     condition = new And(contextNode.getCondition(),condition);
     Tuple t = (contextNode,condition);
     t.setContextNode(currContextNode)
     t.setCondition(condition);
     resultSet.add(t);
}
return resultSet;
END FUNCTION
```

Figure 5-60 *Pseudo Code for Function processStep*

5.8.1.2 *Processing a Predicate*

A predicate is according to its definition again an XPath expression and can thus be processed equally. Yet, we have to watch the result type(s), which in contradiction to the result instances we necessarily can determine. Basically, a predicate may either be a Boolean expression or a numeric expression. If the value of a predicate is a number, it is treated like a numeric predicate; if it is of any other type, it is converted to Boolean using the `boolean()` function (see [Kay01] p. 452-454) and is treated as a Boolean predicate. Since we do not support lateral navigation in this approach (see Section 3.1.2), we also apply the **boolean()** function to numeric values.

So, if the predicate is a Boolean expression, we just have to translate it and use it as condition. If the XPath expression returns a non complex value, it must be converted to Boolean. If it returns nodes, the expression will be an (implicit) existence test. The results of the expression are used as operands to an **EX()** (exists) operation to explicitly express this test. This operation in turn is the condition we are binding to the proxy.

```
FUNCTION processPredicate(
    IN (NodeOperand contextNode, Operand contextCond);
    IN XPathExpression expr;
    OUT Operand;
)
Operand condition;
if (expr instanceof LocationPath)
{
    // this predicate will select multiple nodes
    Set s = new Set[(NodeOperand contextNode, Operand contextCond)];
    s = processLocationPath((contextNode,contextCond),expr);
    // we do not need the nodes. we only need to know if any
    // of them exists
    condition = Constants.FALSE; // none of them exists per se
    For Each Tuple t in s
    {
        // a node exists if its contextCond may be evaluated to true
        condition = new OR(condition, t.getContextCond());
    }
}
elseif (expr instanceof NaryExpression)
{
    // this is an n-ary expression and will thus return a noncomplex value
    Operand o = processNaryExpression((contextNode,contextCond),expr);
    // the expression must evaluate to true
    condition = castToBooleanIfNecessary(o);
}
else
{
    // something else??
    throw new Exception();
}
return condition
END FUNCTION
```

Figure 5-61 *Pseudo Code for Function processPredicate*

5.8.1.3 Processing an N-Ary Expression et al.

The processing of an n-ary expression as well as the other functions are pretty straight forward. But for the sake of completeness, they will at least be mentioned in Appendix D.

5.8.2 Transforming the Expression

When we are done with the building process, we need to apply some minor transformations. Most important, we have to resolve the matrix operands (see Section 5.6.2). Then we could also apply a few optimizations, for instance to remove unnecessary **TRUE AND ...** constructs (see Section 5.6.3). But we do not attempt to optimize the whole expression, since the general problem, called *query minimization* is NP-complete (see [ChMe77]).

5.8.3 Serializing the Expression to SQL Queries

Since an XPath expression may select multiple elements from the Schema tree, it is not possible with our mapping (see Section 5.3) to obtain the necessary data with one single **SELECT** statement. Indeed, multiple elements representing multiple tables could be gathered using a union join which in fact is also not really different from a **UNION ALL** statement (see [Date93] p. 239). Thus for the simplicity, we assume that every result Schema element creates one separate SQL query, ignoring performance issues that may or may not result from issuing just one query utilizing **UNION**. (Microsoft's SQL Server 2000 for instance does not seem to optimize **UNION** statements at all.)

To create the SQL statement for a certain Schema element, we first serialize the PTG for the **FROM** clause (see Section 5.7.5). This makes available a unique variable name for every position within the virtual document we create with this step within the database.

Since our conditions contain references to the nodes of the PTG, we can easily obtain the unique variable names they received and use them within the serialization of the condition. Thus we can easily take over the condition as it is just using the terms of SQL serializing it.

5.8.4 Creating Virtual Document Node Instances

According to the object model (see Figure 5-11), we need to create a **Row** object for each resulting row of each accruing SQL query, no matter whether the desired result node is an **ElementNode** or an **AttributeNode**. Since the **Row** object represents a unique table row in our database, we have to save the necessary key column values within that **Row** object. Thus the **SELECT** part of the serialized Expression (see Section 5.8.3) has to reflect this.

More interesting than the true creation of the Virtual Document Nodes (it can easily be derived from the description of the navigation from a child to its parent element from Section 5.3) is the point of time of creation. Either it uses *early evaluation*, creating them right after the querying and filling them into a bird-brained set which is returned to the user. An other possibility would be the *lazy evaluation*, equipping the set with the result (schema-) node objects and the conditions. Such a set would materialize the virtual (instance-) nodes it contains not until they are really accessed by either counting them, iterating through then, accessing their values, et cetera.

Assume we want to process an XSLT style sheet. Since their `template` definitions (see [Kai01], p. 312-323) are normally nested, the reuse of the current XPath query result is pretty likely. The first rule to apply in such a document is mostly `<xsl:template match=”/” />` and just used to call other templates. For a naïve XSLT processor implementation we would in any case already have an enormous advantage over approaches that serialize the whole result document. Nevertheless we would strongly recommend to use lazy evaluation, to not let steps access the database, that are only used for structuring purposes and do not use any data from the virtual document.

5.9 *Realization*

We have implemented an early prototype of the XPath-to-SQL translation process in Java. For the parsing of XPath into an advantageous object model (see 5.8.1), we used SixPath[34]. This open source library (Mozilla Public License) consists of a parser for XPath expressions and a set of classes which are used to represent their structures. The parser creates an object representation of the posed XPath expression which we can easily be read out.

```
Q1: /company/division
Q2: /company/division//@name
Q3: //@name
Q4: //employee[@name='Angel']/ancestor::company//@name
Q5: //employee[@name='Lord']/ancestor::company//employee[@name='Devil']/ancestor::company//@name
```

Figure 5-62 *Five test queries for our prototype*

The prototype produced the demanded translations for the test queries (five of those are shown in Figure 5-62). Figure 5-63 shows a table containing the time the translation of 100.000 XPath queries (20.000 times translating the 5 queries of Figure 5-62, including serialization to SQL) required on JDK 1.4 on an AMD Athlon 1,33 GHz processor. This shows us, the translation process is not only simple to understand but also ‘simple to process’. A production version is likely to be even faster, since the prototype was only a proof of concepts.

[34] see http://sourceforge.net/projects/sixpath/

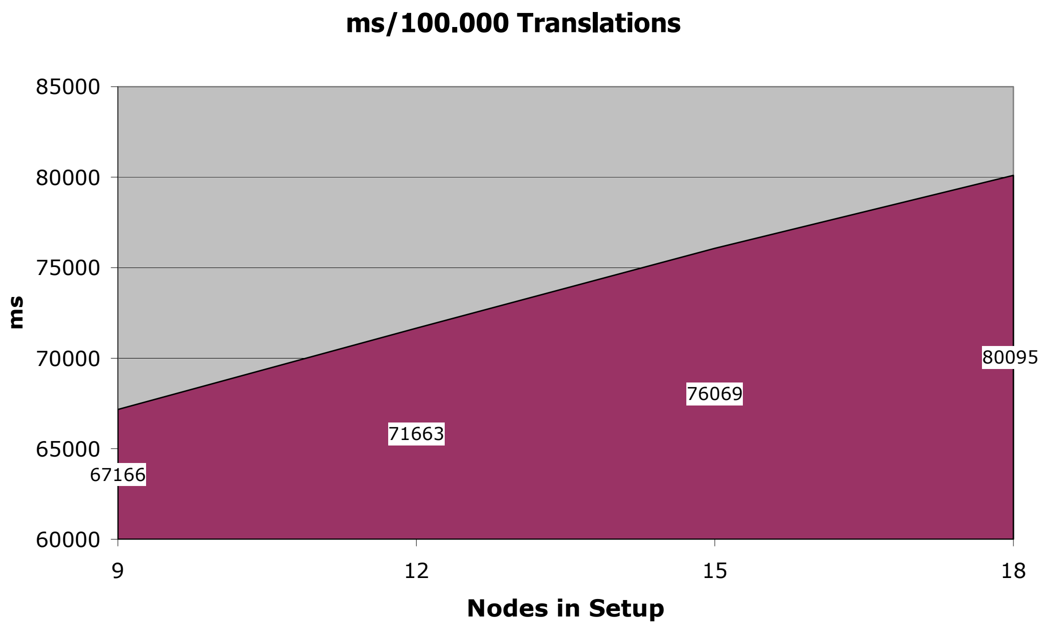


Figure 5-63 *Translation speed for 9/12/15/18 Nodes in Setup*

6 Conclusion and Future Work

We have presented a way to query relational database table data by XPath through a view simulating a virtual document. In contradiction to any other approach, we use an object model to only return the results instead of deep-materializing the whole selected document fragment, tagging it and returning it in text form. This enables relative queries by providing a certain context node with the XPath query string. The object model further on allows basic navigation according to the functions of the W3C Document Object Model.

As we already have mentioned in Section 5.4.3.2, we use the Redundant Relation Approach (see [SSB+00]) for the internal simulation of the virtual document. We do not expect this approach to behave that bad in our case, since we do not use it for outputting purposes, thus requiring a lot of tuples to really be materialized, but for querying purposes. Branches from the main location path are always used to narrow down the results, they con only be created through predicates. Thus the *fan out* as it is called in that paper should not really be an issue here. But for sure it would be interesting to try to find other ways of serializing the PTG to SQL and compare their speed.

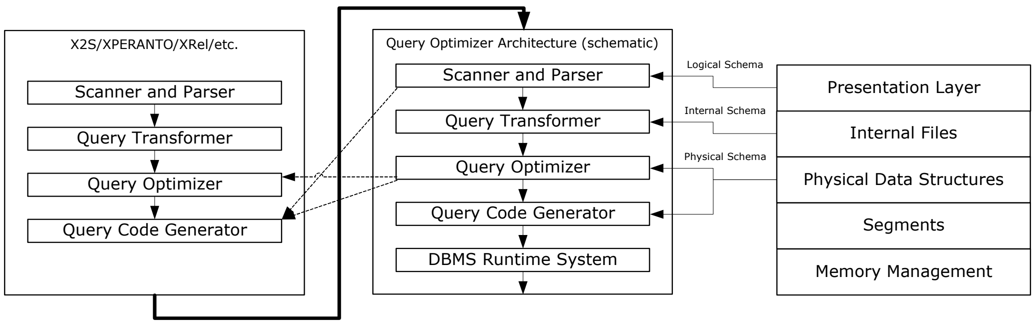


Figure 6-1 *Current Approaches*

An other point worth researching would be to evaluate (existing) query optimization techniques for the applicability in conjunction with both, processing XPath queries and materializing their results. [JaMS02] and [Shan01] do already parts of this work, but they stick to SQL interface, inducing special optimizations of the query through the relational database systems query optimizer by special SQL statements (as shown in Figure 6-1). A reason for this is that general purpose optimizers, like the ones in (commercial) relational database management systems, do not and sometimes even *can not* apply optimizations that would be valid for the special case of processing an XPath (or likewise) query (see [JaMS02]).

The question is, whether there may be an even greater potential for optimization (also regarding those requiring special metadata as described in Section 5.4.3) not using the detour via SQL, or if the advantage of these optimizations in point of fact can as well be achieved through 'induction'. In the latter case, one would surely not want to abandon the platform independence of the current approaches. An open source RDBMS like PostgreSQL or MySQL could allow to attach a special purpose optimizer for determining processing plans for XPath queries, using only primitives of a lower system level to evaluate these different techniques. A good starting point for this research could be [FeMS01].

Besides a reliable and optimized implementation, two things which are missing in this approach are the support for ordering (**`preceding`**, **`preceding-sibling`**, **`following`**, **`following-sibling`** axes) and read-write support of the result object model. Imaginable extensions to this work may further be an interactive browser, allowing a user to navigate the view of the database and an XSLT processor, making use of the provided access to relational data.

Appendix A

This is an Example XSD (XML Schema Definition) defining the element structure of a Setup XML document as described in section 5.5. Even though it is relatively strict in matters of valid keys and key references, it still has two weaknesses: (a) It can not ensure that the attributes columns belong to the containing elements table and (b) it does not allow ‘endless’ recursions. Thus a user defining a setup according to this XSD has to define a maximum level of recursion and an implementation using such a setup will ignore any database data that would require a deeper recursion. A delineation of the file is shown in below.

It shall formally be pointed out, that this schema is only an example. Many other schemas are possible without interfering with the rest of this approach.

Schema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:element name="setup">
        <xs:complexType>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:choice>
                    <xs:element name="table">
                        <xs:annotation>
                            <xs:documentation>Used to define a table.</xs:documentation>
                        </xs:annotation>
                        <xs:complexType>
                            <xs:sequence maxOccurs="unbounded">
                                <xs:choice>
                                    <xs:element name="column">
                                        <xs:complexType>
                                            <xs:attribute name="name" type="xs:string"
                                                use="required"/>
                                            <xs:attribute name="CID" type="xs:ID"
                                                use="required"/>
                                        </xs:complexType>
                                    </xs:element>
                                    <xs:element name="key">
                                        <xs:complexType>
                                            <xs:sequence maxOccurs="unbounded">
                                                <xs:element name="columnref">
                                                    <xs:complexType>
                                                        <xs:attribute name="refer"
                                                            type="xs:IDREF"
                                                            use="required"/>
                                                    </xs:complexType>
                                                </xs:element>
                                            </xs:sequence>
                                            <xs:attribute name="KID" type="xs:ID"
                                                use="required"/>
                                        </xs:complexType>
                                    </xs:element>
                                    <xs:element name="foreignkey">
                                        <xs:complexType>
                                            <xs:sequence maxOccurs="unbounded">
                                                <xs:element name="columnref">
                                                    <xs:complexType>
                                                        <xs:attribute name="refer"
                                                            type="xs:IDREF"
```

```
                                             use="required"/>
                                       </xs:complexType>
                                    </xs:element>
                                 </xs:sequence>
                                 <xs:attribute name="keyref" type="xs:IDREF"
                                    use="required"/>
                              </xs:complexType>
                           </xs:element>
                        </xs:choice>
                     </xs:sequence>
                     <xs:attribute name="TID" type="xs:ID" use="required"/>
                     <xs:attribute name="name" type="xs:string" use="required"/>
                  </xs:complexType>
                  <xs:keyref name="keycolumnref" refer="localcolumnid">
                     <xs:selector xpath="key/columnref"/>
                     <xs:field xpath="@refer"/>
                  </xs:keyref>
                  <xs:keyref name="foreignkeycolumnref" refer="localcolumnid">
                     <xs:selector xpath="foreignkey/columnref"/>
                     <xs:field xpath="@refer"/>
                  </xs:keyref>
                  <xs:key name="localcolumnid">
                     <xs:selector xpath="column"/>
                     <xs:field xpath="@CID"/>
                  </xs:key>
               </xs:element>
               <xs:element name="element" type="elementType"/>
            </xs:choice>
         </xs:sequence>
      </xs:complexType>
      <xs:key name="tablekey">
         <xs:selector xpath="table/key"/>
         <xs:field xpath="@KID"/>
      </xs:key>
      <xs:keyref name="foreignkeyref" refer="tablekey">
         <xs:selector xpath="table/foreignkey"/>
         <xs:field xpath="@keyref"/>
      </xs:keyref>
      <xs:key name="tableid">
         <xs:selector xpath="table"/>
         <xs:field xpath="@TID"/>
      </xs:key>
      <xs:key name="columnid">
         <xs:selector xpath="table/column"/>
         <xs:field xpath="@CID"/>
      </xs:key>
      <xs:keyref name="elementtableref" refer="tableid">
         <xs:selector xpath=".//element"/>
         <xs:field xpath="@table"/>
      </xs:keyref>
      <xs:keyref name="attributecolumnref" refer="columnid">
         <xs:selector xpath=".//attribute"/>
         <xs:field xpath="@column"/>
      </xs:keyref>
   </xs:element>
   <xs:complexType name="elementType">
      <xs:annotation>
         <xs:documentation>The structure of an element definition</xs:documentation>
      </xs:annotation>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:choice>
            <xs:element name="attribute">
               <xs:complexType>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                  <xs:attribute name="column" type="xs:IDREF" use="required"/>
               </xs:complexType>
            </xs:element>
            <xs:element name="element" type="elementType">
            </xs:element>
         </xs:choice>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="table" type="xs:IDREF" use="required"/>
   </xs:complexType>
</xs:schema>
```

Appendix B

The following XML file describes the Setup (see Section 5.5) for the Extended Example (see Section 5.2) according to the Example XSD of Appendix A.

ExtExample.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<setup
    xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
    xsi:noNamespaceSchemaLocation="schema.xsd">
    <table TID="t01" name="COMPANY">
        <key KID="k0101">
            <columnref refer="c0101"/>
        </key>
        <column name="CID" CID="c0101"/>
        <column name="Name" CID="c0102"/>
    </table>
    <table TID="t02" name="DIVISION">
        <key KID="k0201">
            <columnref refer="c0201"/>
        </key>
        <foreignkey keyref="k0101">
            <columnref refer="c0202"/>
        </foreignkey>
        <column name="DID" CID="c0201"/>
        <column name="Company" CID="c0202"/>
        <column name="Name" CID="c0203"/>
    </table>
    <table TID="t03" name="EMPLOYEE">
        <key KID="k0301">
            <columnref refer="c0301"/>
        </key>
        <foreignkey keyref="k0201">
            <columnref refer="c0302"/>
        </foreignkey>
        <foreignkey keyref="k0301">
            <columnref refer="c0303"/>
        </foreignkey>
        <column name="EID" CID="c0301"/>
        <column name="Division" CID="c0302"/>
        <column name="Boss" CID="c0303"/>
        <column name="Name" CID="c0304"/>
    </table>
    <table TID="t04" name="PRODUCT">
        <key KID="k0401">
            <columnref refer="c0401"/>
        </key>
        <foreignkey keyref="k0101">
            <columnref refer="c0402"/>
        </foreignkey>
        <column name="PID" CID="c0401"/>
        <column name="Company" CID="c0402"/>
        <column name="Name" CID="c0403"/>
    </table>
    <table TID="t05" name="TAKER">
        <key KID="k0501">
            <columnref refer="c0501"/>
        </key>
        <column name="TID" CID="c0501"/>
        <column name="Name" CID="c0502"/>
    </table>
    <table TID="t06" name="ORDER">
        <key KID="k0601">
            <columnref refer="c0601"/>
        </key>
        <foreignkey keyref="k0501">
            <columnref refer="c0602"/>
        </foreignkey>
        <column name="OID" CID="c0601"/>
        <column name="Taker" CID="c0602"/>
        <column name="receiveDate" CID="c0603"/>
        <column name="sendDate" CID="c0604"/>
    </table>
    <table TID="t07" name="ITEM">
        <key KID="k0701">
            <columnref refer="c0701"/>
        </key>
        <foreignkey keyref="k0601">
            <columnref refer="c0702"/>
        </foreignkey>
        <foreignkey keyref="k0401">
            <columnref refer="c0703"/>
        </foreignkey>
        <column name="IID" CID="c0701"/>
        <column name="Order" CID="c0702"/>
        <column name="Product" CID="c0703"/>
    </table>
    <element name="company" table="t01">
        <attribute name="id" column="c0101"/>
        <attribute name="name" column="c0102"/>
        <element name="division" table="t03">
            <attribute name="id" column="c0301"/>
            <attribute name="name" column="c0302"/>
            <element name="employee" table="t04">
                <attribute name="id" column="c0401"/>
                <attribute name="name" column="c0402"/>
                <element name="employee" table="t04">
                    <attribute name="id" column="c0401"/>
                    <attribute name="name" column="c0402"/>
                    <element name="employee" table="t04">
                        <attribute name="id" column="c0401"/>
                        <attribute name="name" column="c0402"/>
                        <element name="employee" table="t04">
                            <attribute name="id" column="c0401"/>
                            <attribute name="name" column="c0402"/>
                        </element>
                    </element>
                </element>
            </element>
        </element>
    </element>
    <element name="taker" table="t05">
        <attribute name="id" column="c0501"/>
        <attribute name="name" column="c0502"/>
        <element name="order" table="t06">
            <attribute name="id" column="c0601"/>
            <attribute name="receiveDate" column="c0603"/>
            <attribute name="sendDate" column="c0604"/>
            <element name="item" table="t07">
            <attribute name="id" column="c0701"/>
                <attribute name="product" column="c0703"/>
            </element>
        </element>
    </element>
</setup>
```

Appendix C

The following shows the intermediate SQL statement for the query **`/company[@name='FZI']`** applied to the view created through the annotated XML-Data Reduced Schema from Figure 4-6 for the introductory example from Section 1.2. It can be obtained using Microsoft Query Profiler when issuing the XPath query against the Microsoft SQL Server 2000.

```
select   1 as TAG,
         0 as parent,
         _Q1.A0 as [company!1!name],
         _Q1.C0_ID as [company!1!ID!hide],
         NULL as [division!2!name],
         NULL as [division!2!ID!hide],
         NULL as [employee!3!name]
from     (
                 select   _QB0.Name AS A0,
                          _QB0.ID AS C0_ID,
                          _QB0.Name AS C0_Name
                 from     Company _QB0
         ) _Q1
where    (
             (CONVERT(nvarchar(4000),_Q1.A0,126)=N'FZI')
             and _Q1.A0 IS NOT NULL
         )

union all

select   2,                                -- TAG
         1,                                -- parent
         NULL,                             -- [company!1!name]
         _Q1.C0_ID,                        -- [company!1!ID!hide]
         _Q2.A4,_Q2.C1_ID,                 -- [division!2!name]
         NULL,                             -- [division!2!ID!hide]
         NULL                              -- [employee!3!name]
from     (
                 select   _QB0.Name AS A4,
                          _QB0.ID AS C1_ID,
                          _QB0.Name AS C1_Name,
                          _QB0.Company AS C1_Company
                 from     Division _QB0
         ) _Q2,
         (
                 select   _QB0.Name AS A0,
                          _QB0.ID AS C0_ID,
                          _QB0.Name AS C0_Name
                 from     Company _QB0
         ) _Q1
where    (
             (CONVERT(nvarchar(4000),_Q1.A0,126)=N'FZI')
             and _Q1.A0 IS NOT NULL
         )
         and _Q1.C0_ID=_Q2.C1_Company

union all

select   3,                                -- TAG
         2,                                -- parent
         NULL,                             -- [company!1!name]
         _Q1.C0_ID,                        -- [company!1!ID!hide]
         NULL,                             -- [division!2!name]
         _Q2.C1_ID,                        -- [division!2!ID!hide]
         _Q3.A5                            -- [employee!3!name]
from     (
                 select   _QB0.Name AS A5,
                          _QB0.Name AS C3_Name,
                          _QB0.Division AS C3_Division
                 from     Employee _QB0
         ) _Q3,
         (
                 select   _QB0.Name AS A4,
                          _QB0.ID AS C1_ID,
                                 _QB0.Name AS C1_Name,
                                 _QB0.Company AS C1_Company
                         from    Division _QB0
         ) _Q2,
         (
                         select  _QB0.Name AS A0,
                                 _QB0.ID AS C0_ID,
                                 _QB0.Name AS C0_Name
                         from    Company _QB0
         ) _Q1
where    (
             (CONVERT(nvarchar(4000),_Q1.A0,126)=N'FZI')
             and _Q1.A0 IS NOT NULL
         )
         and _Q2.C1_ID=_Q3.C3_Division
         and _Q1.C0_ID=_Q2.C1_Company

order by 4,6,2
```

Appendix D

Here, the more or less obvious parts of the core algorithm for building the expression (see Section 5.8.1.3) shall be annotated for the sake of completeness.

```
FUNCTION processNaryExpr(
    IN (NodeOperand contextNode, Operand contextCond);
    IN XPathExpression expr;
    OUT Operand;
)
    int termcount = expr.getTermCount();
    if (termcount<2)
    {
        // how comes?
        throw new Exception();
    }
    Expression term = expr.getTerm(0);
    OperandSet operandset = processExpr((contextNode, contextCond), term);
    Operand operand = operandset;
    for (int i=1; i<termcount; i++)
    {
        term = expr.getTerm(i);
        BinaryOperation operation = expr.getOperator().getInstance();
        operation.setLeftOperand(operand);
        operandset = processExpr((contextNode, contextCond), term);
        operation.setRightOperand(operandset);
        operand = operation;
    }
    return operand;
END FUNCTION

FUNCTION processExpr(
    IN (NodeOperand contextNode, Operand contextCond);
    IN XPathExpression expr;
    OUT Operand;
)
    if (expr instanceof XPathFilterExpr)
    {   // not yet implemented
        throw new Exception();
    }
    if (expr instanceof XPathFunctionCall)
    {   // not yet implemented
        throw new Exception();
    }
    if (expr instanceof XPathLiteral)
    {
        Variable result = new Variable();
        result.setContent(expr.getValue());
        result.setDataType(DataType.String);
        return result;
    }
    if (expr instanceof XPathLocationPath)
    {
        return processLocationPath((contextNode, contextCond), expr);
    }
    if (expr instanceof XPathNaryExpr)
    {
        return processNaryExpr((contextNode, contextCond), expr);
    }
    if (expr instanceof XPathNumber)
    {
        Variable result = new Variable();
        result.setContent(expr.getValue());
        result.setDataType(DataType.Number);
        return result;
    }
    if (expr instanceof XPathVariable)
    {   // not yet implemented
        throw new Exception();
    }
    // come here?
    throw new Exception();
}
```

Appendix E

This is the grammar in BNF for the Rule Language of the approach of Jain et al. It has been obtained from http://www.cs.washington.edu/homes/ratul/dbproject/grammar.html in July 2002. It is below-mentioned to ensure its availability for the case the web site moves. The parts concerning XPath are marked in dark gray.

```
Program ::= ( ( Head "=" Body ";" ) )* <EOF>
Head ::= ( ( Id ) "(" ( Tag ) ( "," ( PList ) )? ")" )
Body ::= ( SimpleBody | ConditionalBody )
SimpleBody ::= ( FCall | ReturnCall )
FCall ::= ( ( Id ) "(" ( XPath ) ( "," ( PList ) )? ")" )
ReturnCall ::= ( <RETURN> "(" ( PList ) ")" )
ConditionalBody ::= <IF> "(" ( Condition ) ")" ( SimpleBody ) ( <ELSE> Body )?
Condition ::= ( ( OrCondition )  | ( AndCondition )  | ( SimpleCondition ) )
SimpleCondition ::= ( ( RelationalExpr ) | "(" ( Condition ) ")" )
OrCondition ::= ( SimpleCondition ) "||" ( SimpleCondition )
AndCondition ::= ( SimpleCondition ) "&&" ( SimpleCondition )
NotCondition ::= "!" ( SimpleCondition )
RelationalExpr ::= ( ( IdOrPhiOrConst ) "==" ( IdOrPhiOrConst )
                      | ( IdOrPhiOrConst ) "!=" ( IdOrPhiOrConst )
                      | ( IdOrPhiOrConst ) "<" ( IdOrPhiOrConst )
                      | ( IdOrPhiOrConst ) "<=" ( IdOrPhiOrConst )
                      | ( IdOrPhiOrConst ) ">" ( IdOrPhiOrConst )
                      | ( IdOrPhiOrConst ) ">=" ( IdOrPhiOrConst ) )
XPath ::= ( ( "/" ) | ( "/" ( RelativePath ) )
             | ( RelativePath ) | ( XPath "|" XPath ))
RelativePath ::= ( ( XPUnit ) ( "/" ( XPUnit ) )* )
XPUnit ::= ( "." | ".." | Id | "*" )
PList ::= ( Id ( "," ( Id ) )* )
Tag ::= ( ( Id ) | "*" )
IdOrPhiOrConst ::= ( PhiVal | PhiTag | IdVal | IdTag | Constant )
PhiVal ::= <PHI_VAL>
PhiTag ::= <PHI_TAG>
IdVal ::= <IDENTIFIER> ".val"
IdTag ::= <IDENTIFIER> ".tag"
Constant ::= <CONSTANT>
Id ::= <IDENTIFIER>
```

Bibliography

[BBG+99] Datablitz storage manager: Main memory database performance for critical applications, Baulier, J., Bohannon, P., Gogate, S., Gupta, C., Haldar, S., Joshi, S., Khivesera, A., Korth, H. F., McIlroy, P., Miller, J., Narayan, P. P. S., Nemeth, M., Rastogi, R., Seshadri, S., Silberschatz, A., Sudarshan, S., Wilder, M., Wei, C., In Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1999

[BGK+02] Optimizing View Queries in ROLEX to Support Navigable Result Trees, Bohannon, P. Ganguly, S., Korth, H. F., Narayan, P. P. S., Shenoy, P., In Proceedings of 28th International Conference on Very Large Data Bases, August 2002, Hong Kong, China

[BoKN01] The Table and the Tree: On-Line Access to Relational Data through Virtual XML Documents, Bohannon, P., Korth, H. F., Narayan, P. P. S., Proceedings of the Fourth International Workshop on the Web and Databases (WebDB'2001), Santa Barbara, CA, May 24-25, 2001

[Brad01] The XML companion, 3rd edition, Bradley, N., Addisson-Wesley, 2001

[CFI+00] XPERANTO: Publishing Object-Relational Data as XML, Carey, M., Florescu, D., Ives, Z., Lu, Y., Shanmugasundaram, J., Shekita, E., Subramanian, S., In Proceedings of WebDB, Dallas, TX, May 2000

[CKS+00] XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents, Carey, M., Kiernan, J., Shanmugasundaram, J., Shekita, E., Subramanian, S., In Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

[ChF+01] XQuery: A query language for XML, Chamberlin, D., Florescu, D., et al, J. R. (2001). In W3C Working Draft, http://www.w3.org/TR/xquery.

[ChMe77] Optimal implementation of conjunctive queries in relational data bases, Chandra, A., Merlin, P., In Procedings of the 9th ACM Symposium on Theory of Computing, pages 77-90, Boulder, Colorado, Mai 1977

[ChRF00] Quilt: An XML query language for heterogeneous data sources, Chamberlin, D., Robie, J., and Florescu, D., In Proceedings of the International Workshop on Web and Databases (WebDB '2000), Springer Verlag, 2000

[Date93] A guide to the SQL Standard : a users guide (covers "SQL2"), C. J. Date with Hugh Darwen, 3rd ed., Addisson-Wesley, 1993, ISBN 0-201-55822-X

[DeFF99] XML-QL: A query language for XML, Deutsch, A., Fernandez, M., and Florescu, D., In Proceedings of the 8th International World Wide Web Conference, Toronto, May 1999

[FeMS01] Efficient Evaluation of XML Middleware Queries, Fernandez, M., Morishima, A., Suciuy, D., ACM SIGMOD 2001 May 2124, Santa Barbara, California, USA, Copyright 2001 ACM 1581133324/01/05

[FFL+02] XTABLES: Bridging Relational Technology and XML, Fan, C., Funderburk, J., Lam, H., Kiernan, J., Shekita, E., Shanmugasundaram, J., March 2002, (not yet published, see http://xperanto.dfw.ibm.com/demo/papers/xtables.pdf)

[GrWe94] LAN Times Guide To SQL, Groff, J. R., Weinberg, P. N., Osborne McGrawhill, 1994 ISBN 0-07-882026-X

[HaMe01] XML in a Nutshell. A Desktop Quick Reference, Harold, E. R., Means, W. S., O'Reilly, January 2001

[Isem01] Microsoft SQL Server 2000 Reference Library, SQL Server 2000 Architecture and XML/Internet Support (Volume 1), David Iseminger, Microsoft Press, 2001, ISBN 0-7356-1280-3, Chapter 13 “Accessing SQL Server Using HTTP”, Chapter 14 “Creating XML Views Using XDR Schemas”, Chapter 15 “Using XPath Queries”

[JaMS02] Translating XSLT Programs to Efficient SQL Queries, Jain, S., Mahajan, R., Suciu, D., WWW2002, May 2002, Honolulu, Hawaii, USA, ACM 1581134495/02/0005, Project Homepage: http://www.cs.washington.edu/homes/ratul/dbproject/

[KaST02] Datenbanken und XML, Kazakos, W., Schmidt, A., Tomczyk, P., Springer Verlag Berlin Heidelberg, 2002, ISBN 3-540-41956-X

[Kay01] XSLT Programmers Reference, 2nd Edition, Michael Kay, Wrox Press, 2001, ISBN 1-861005-06-7

[LaLo95] Datenbankeinsatz, Stefan M. Lang, Peter C. Lockemann, Springer, 1995, ISBN 3-540-58558-3, Chapter 22.3.1 “Syntaktische Grundform des Anfragemodells.“

[Malc01] Programming Microsoft SQL Server 2000 With XML, Graeme Malcom, Microsoft Press, 2001, ISBN0-7356-1369-9, Chapter 2 “Retrieving XML Data Using Transact-SQL”, Chapter 3 “Using ADO for XML Data Access”, Chapter 4 “Using HTTP for Data Access”, and Chapter 5 “Using XML Templates to Retrieve Data over HTTP”

[MSDN01] XML and Internet Support, Using EXPLICIT Mode, MSDN, http://msdn.microsoft.com/library/en-us/xmlsql/ac_openxml_4y91.asp

[Shan01] Bridging Relational Technology and XML, A dissertation submitted in partial fulfillment of the requirements of the degree of Doctor of Philosophy (Computer Science), Shanmugasundaram, J., University of Wisconsin, Madison, 2001

[SKS+01] Querying XML Views of Relational Data, Shanmugasundaram, J., Kiernan, J., Shekita, E., Fan, C., Funderburk, J., In Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

[SSB+00] Efficiently Publishing Relational Data as XML Documents, Shanmugasundaram, J., Shekita, E., Barr, R., Careyq, M., Lindsay, B., Pirahesh, H., Reinwald, B., In Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

[Vali02] Технология XSLT, Алексей Валиков, БХВ-Петербург, 2002, ISBN 5-94157-129-1, Chapter 6 "XPath-выражения".

[W3C98a] Extensible Markup Language (XML) 1.0, W3C Recommendation 10-February-1998, http://www.w3.org/TR/1998/REC-xml-19980210

[W3C98b] Document Object Model (DOM) Level 1 Specification, , Version 1.0, W3C Recommendation 1 October, 1998, http://www.w3.org/TR/REC-DOM-Level-1/

[W3C99] XML Path Language (XPath) version 1.0 http://www.w3c.org/TR/xpath

[YASU01] XRel: a path-based approach to storage and retrieval of XML documents using relational databases, Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, Shunsuke Uemura, ACM Transactions on Internet Technology (TOIT), Volume 1, Issue 1 (August 2001), ACM Press, 2001, ISSN:1533-5399, http://portal.acm.org/citation.cfm?doid=383034.383038