Towards Unifying Selection Mechanisms for DB- and IR-Systems

Klaus Benecke
Otto-von-Guericke University Magdeburg
39016 Magdeburg, Germany
benecke@iws.cs.uni.magdeburg.de

Abstract

With the help of an algebraic specification language and the functional programming language OCAML [Chailloux et al., 2000] we introduced a new understanding of XML. For example, in our abstract XML specification we distinguish in the data structure and not only in the DTD between a tuple and a collection. Further all intermediate results of the generating operations are considered as XML documents. Because of this understanding of XML, we could define and implement new, powerful, and easy to handle operations for XML documents. For example, we have a restructuring operation, by which an XML document can be transferred to another one only by giving the DTD of the desired document. The paper presents a description of a complex selection operation with simple syntax, which can be used in database and information retrieval environments. It will be evident that some well known commuting rules from other data models fail, but nevertheless a first outlook for query optimization strategies is given. It will be clear that OttoQL (our language based on the discussed operations) differs from XQuery very significantly.

1 Introduction

Our end-user computer language OttoQL (OsTäfisch Table Oriented) [Benecke and Schnabel, 2009] was originally designed as a database language for non-first-normal-form relations. With the introduction of XML it was generalized to (XML) documents. The highlights of OttoQL are:

1. It is based on an algebraic specification language, which distinguishes, for example, between tuples and collections.
2. The program logic of the kernel is very simple; the operations are applied one after the other.
3. There are complex, powerful operations on structured data, which are easy to handle.

This paper covers only a part of the possibilities of OttoQL, namely - the select operation. The input of an OttoQL program can be either an XML document with DTD or a tab file. Both objects are internally represented by an (abstract) OCAML term. Such terms are transformed into new terms by operations of the program. The resulting term of the program can be represented as tab file, table, XML document, HTML document, or as an OCAML term. The XML document pupils.xml in Figure 1 contains data about three pupils. It can be represented in the following way in form of a tab file (Table 1).

The computer internal representation of this tab file may also include the “invisible” above tags PUPILS, PUPIL, and SUBJECTTUP. The use of these tags in the tabular representation would damage the tabular view. We will use this representation in this paper because it makes our understanding of XML more visible. Tuples are arranged horizontally and elements of collections (except at deepest level, if this level contains only one elementary field) vertically. The second section describes the specification of schemes of tabments and the specification of tabments. The ext operation, by which computations can be realized, is then briefly given in section 3. In the following section 4 the selection (mit-part) is introduced by examples and partially by corresponding XQuery programs. Selections are done in general by several mit-parts step by step. In the fifth section the essential part of the selection is reduced mainly to the ext operation. In section 6, the failure of well known rules for query optimization is shown. Furthermore, definitions are presented, which are already implicitly contained in the definition of section 5. Then improved commuting rules, in part of a new kind, are presented. These rules are a basis for query optimization strategies (section 7). Section 8 compares shortly our approach with others and in section 9 a summary of our data model is given.

2 XML IN OCAML

In this section we present our understanding of XML in the syntax of OCAML. An XML document is also called tabment (TABle+docuMENT).

```
type coll_sym = Set | Bag | List | Any
   | S0 ;; (* collection types: M, B, L, A, ? *)

(*collection types: M, B, L, A, ? *)

type name = string;;
   (*column names *)

(*column names *)

type scheme =
   (* schemes of documents *)
   Empty_s
   (* empty scheme *)
   | Inj of name
   (* each name is a scheme *)
   | Coll_s of coll_sym*scheme
   (*schemes for collections*)
```

Table 1: pupils.xml as tab file pupils.tab
Figure 1: Sample file pupils.xml

| Tuple_s of scheme list (* schemes for tuples *) |
| Alternate_s of scheme list:: (* schemes for choice *) |

type value = (* disjoint union of elementary types *)
Bar (* a dash; only for school of interest *)
Int_v of big_int (* each big integer is a value *)
Float_v of float
Bool_v of bool
String_v of string::;

type tabment = (* type for tables resp. documents *)
Empty_t (* empty tabment; error value *)
El_tab of value (* an elementary value is a tabment *)
Tuple_t of tabment list (* tuple of tabments *)
Coll_t of (coll_sym * scheme) * (tabment list) (* collection of tabments *)
Tag0 of name * tabment (* a tabment is enclosed by a name *)
Alternate_t of (scheme list) * tabment:: (* the type of the tabment is changed to a choice type *)

Examples: The string "Hallo" ("XML document" without root) can be represented by the OCAML term

El_tab(String_v "Hallo")

and the XML document

<X><A>a</A><A>b</A></X>

can be represented for example by

Tag0("X",Tuple_t{Tag0("A",El_tab(String_v "a"));Tag0("A",El_tab(String_v "b")))

or by

Tag0("X",Coll_t((List, Inj "A"), [Tag0("A",El_tab(String_v "a"));Tag0("A",El_tab(String_v "b"))])).

The third pupil of the above XML file has the following description:

Tag0 ("PUPIL", Tuple_t [Tag0 ("NAME", El_tab (String_v ("Mayer"))));Tag0 ("FIRST", El_tab (String_v ("Fritz"))));Coll_t ((Set,Inj "SUBJECTTUP"),[Tag0 ("SUBJECTTUP", Tuple_t [Tag0 ("SUBJECT", El_tab (String_v ("Maths"))));Coll_t ((Set,Inj "MARK"), [ ])]))]

We summarize the differences between the common understanding XML documents and the specified tabments:

1. The specification does not distinguish between XML-attributes and XML-elements; an attribute is signaled by a preceding "@".
2. Unlike to XML a tabment need not to have a root tag.
3. In the tabment, and not only in the scheme specification, a tuple of several elements is distinguished from a collection of these elements. This is an advantage, for the specification and implementation of our powerful tabment operations (restructuring stroke, selection, extension ext, vertical, ...).
Unlike some other languages a selection can be placed at level \texttt{mit} (\texttt{level} and means "without"). A "where" from SQL) or a \texttt{level} \texttt{mit} \texttt{SUBJECT} \texttt{MARK}=3 selects SUBJECTTUP elements (all, in which a mark 3 exists) and \texttt{MARK}:: MARK>3 selects only MARKs. Sometimes no outer tag like \texttt{PUPIL} or SUBJECTTUP is present. Therefore, we also permit column names from inside the tuple as level selectors (topmost level of the corresponding collection).

Program 2: Selection at top level (Give all pupils, who have a German entry:)

\begin{verbatim}
> aus doc("pupils.xml")
> mit PUPIL:: SUBJECT="German"
\end{verbatim}

The condition selects from the collection of all pupils and not from the collections of SUBJECTTUP elements. However, with each qualified pupil all its subordinated SUBJECTTUP- and MARK-values appear. Thus we get the following result:

\begin{verbatim}
<<L(NAME, FIRST, L(SUBJECT, L(MARK)))::
  Schulz Michael German 1 4 4
  Meier Hans              4 4 4
  Mayer Fritz             4 4 4
\end{verbatim}

We recognize that the condition selects all pupils, who have at least one German entry. However, all subjects of those pupils are in the result. For people, who do not know the SUBJECT tag we plan to implement also a keyword search of type \texttt{PUPIL} : "German" or simply "German". If we want only all German entries then this can be easily expressed:

Program 3: Selection at second level: Find for each pupil (more general: with a certain property) all German data:

\begin{verbatim}
> aus doc("pupils.xml")
> mit SUBJECT:: SUBJECT="German"
\end{verbatim}

The condition selects only MARKs. Sometimes no outer tag like \texttt{PUPIL} or SUBJECTTUP is present. Therefore, we also permit column names from inside the tuple as level selectors (topmost level of the corresponding collection).

\begin{verbatim}
<<L(NAME, FIRST, L(SUBJECT, L(MARK)))::
  Meier Hans              4 4 4
  Schulz Michael German   4 4 4
\end{verbatim}

We recognize that the condition selects all pupils, who have at least one German entry. However, all subjects of those pupils are in the result. For people, who do not know the SUBJECT tag we plan to implement also a keyword search of type \texttt{PUPIL} : "German" or simply "German". If we want only all German entries then this can be easily expressed:
If we apply both conditions, one after the other, then we get all German entries and, in addition, only the pupils, who have a German entry.

**Program 4:** Selection at two different levels:

```plaintext
aus doc("pupils.xml")
mit NAME::SUBJECT="German"
mit SUBJEC::SUBJECT="German"
```

**Result:**

```
<<L(NAME, FIRST, L(SUBJECT, L(MARK)))::
Schulz Michael German 1 4 4
4 4 4>>
```

If we consider the Boolean expression

```
MARK>3
```

then one can select in three collection types. In place of the following three conditions

```plaintext
mit NAME::MARK>3 #pupils with a mark>3
mit SUBJECT::MARK>3
mit MARK::MARK>3 #marks greater than 3
```

we can write shorter

```plaintext
mit NAME, SUBJECT, MARK::MARK>3
```

or even shorter as in

**Program 5:** Relational selection with structured output:

```plaintext
aus doc("pupils.xml")
mit MARK>3 # or: mit MARK::MARK>3
```

**Result:**

```
<<L(NAME, FIRST, L(SUBJECT, L(MARK)))::
Schulz Michael German 4 4 4 4 4
Maths 4 >>
```

The condition MARK: MARK > 3 expresses that the Boolean expression MARK > 3 is applied to all collections, which include MARK (are higher than MARK). For the formulation of Program 5 in XQuery we need 3 nested FLOWR constructs.

**Program 6:** Give all pupils (with all data), who have a mark 1 in Maths:

```plaintext
aus doc("pupils.xml")
mit NAME:: SUBJECT="Maths" and MARK=1
```

This query is equivalent to each of the following two XQuery-programs, which require SUBJECTUP-tags:

```xml
<PUPILS>{
  doc("pupils.xml")//
  SUBJECTUP[SUBJECT="Maths"
and MARK=1]/..
}</PUPILS>
<PUPILS>{
  doc("pupils.xml")//
  PUPIL[SUBJECTUP[SUBJECT="Maths"
and MARK=1]]}
}</PUPILS>
```

But Program 6 is not equivalent to the simpler XQuery program:

```xml
<PUPILS>{
  doc("pupils.xml")//
  PUPIL[.//SUBJECT="Maths"
and ./MARK=1]}
}</PUPILS>
```

The difference between *OttoQL* and *XPath* is that in the last *XPath* program SUBJECT and MARK are independent of each other, but in *OttoQL* both values are considered to adhere to each other. Nevertheless, it is no problem to express the last *XPath* program by *OttoQL*.

**Program 7:** Two independent conditions, which refer to the same collection type: Give all pupils, who have a Maths-entry, and who have a one in any subject:

```plaintext
aus doc("pupils.xml")
mit NAME:: SUBJEC="Maths"
mit NAME:: MARK=1
```

**Program 8:** Disjunction of two conditions:

```plaintext
aus doc("pupils.xml")
mit NAME:: SUBJECT="Maths" or MARK=1
```

Here, also the last pupil (Mayer Fritz) appears in the result, although a corresponding MARK-value does not exist. The same holds even if both names are not on a hierarchical path.

**Program 9a:** A condition, which contains two names, which are not on a hierarchical path, can be meaningful:

```
<< M(X, M(Y), M(Z))::
1 2 3
4 5 6
7 >>
mit X:: Y=2 or Z=6
```

Here, the input tabment is also the output tabment.

**Program 9b:** A similar condition applied to all three levels:

```
<< M(X, M(Y), M(Z))::
1 2 3
4 2 6
5 7
8 9
9 2 >>
mit Y in M[2; 5] or Z in M[6; 7]
```

**Result:**

```
<< M(X, M(Y), M(Z))::
1 2
4 2 6
5 7
9 2 >>
```

We note that this understanding of or differs from a pure relational environment, where a structure of type *M(X,Y,Z)* is given. In this case the last element could not be in the result, because it would not appear in the given table. The first result line had to be replaced by "1 2 3" and instead of the second element 8 (X,Y,Z)-tuples (the "8" and "9" included) would appear in the result. If we would restructure these tuples to *X, M(Y)* and *M(Z)* contrary to *Y* and *Z* are on a hierarchical path of the given scheme as the following graph of Figure 3 shows. For the next program a simple relation with 2 optional columns is given:
courses.xml: L(COURSE, HOURS?, PROF?)

**Program 10:** Give all course tuples, which have an HOURS-entry greater 20:

```xml
aus doc("courses.xml")
mit COURSE::HOURS>20 # or mit HOURS>20
```

Because the corresponding XML document contains no tuple tags, the solution in XQuery looks relatively complicated:

```xml
<results>
{for $h in doc("courses.xml")//HOURS
  where $h > 20
  return
  <$h/preceding-sibling::COURSE[1]($h)
  {if local-name($h/following-sibling::PROF[1])="PROF"
  then $h/following-sibling::PROF[1]
  else ()}
</tup>}
</results>
```

### 5 SELECTION MORE PRECISE

The selection is based essentially on the extension operation `ext` and the `forget` operation. In short, by `ext n e sns t (n:=e at sns)` besides each name from `sns`, the value of `e` tagged by `n` is inserted. By Program 8 and Program 9b it becomes clear why it is not sufficient to extend at one name only. An extension

```xml
ext %:=SUBJECT="Maths" or MARK=1 && at SUBJECT
```

would generate only empty %-values (undefined) if we have a subject unequal to Maths, because the MARK-values are invisible at SUBJECT-level. An extension

```xml
ext %:=SUBJECT="Maths" or MARK=1 && at MARK
```

at the other hand could not carry a truth value, if the collection of MARK-values is empty. But the following ext operation realizes the desired extensions:

```xml
ext %:=SUBJECT="Maths" or MARK=1 && at SUBJECT,MARK
```

In the same way the corresponding extension for Program 9b is:

```xml
ext %:=Y in M[2;5] or Z in M[6;7] && at Y,Z
```

By such %-extensions the given table is extended at each corresponding position by one of the corresponding three truth values:

- TRUE = Tag0("%", El_tab(Bool_v true))
- FALSE = Tag0("%", El_tab(Bool_v false))
- UNDEFINED = Tag0("%", Empty_t)

Table 2: Tabment T0

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>&gt;&gt;</td>
</tr>
</tbody>
</table>

6 TOWARDS OPTIMIZATION RULES

Unfortunately, well known commuting rules from the relational and other data models do not hold in our data model in general. First, we have a look at Table 2 (T0), on which the failing of certain well-known commuting rules for selection can be demonstrated. In the following σ is the selection operation, which corresponds to one mit part.

**Counter examples** for commuting rules of selection:

(a) $\sigma_{X::Y=4}(\sigma_{Y::Z=3}(T0)) \neq \sigma_{Y::Z=3}(\sigma_{X::Y=4}(T0))$

The left hand side is an empty tabment, unlike the right hand side. The reason is that the condition $Y::Z=3$ selects elements of the fix level (see below) of $X::Y=4$ or in other words $Y::Z=3$ refers (see below) to a fix level $(YZ)$ of the quantified condition $X::Y=4$. We shall see that we can commute both conditions above in another sense. $X::Y=4$ can absorb $Y::Z=3$.

(b) $\sigma_{Y::pos(Y)=1}(\sigma_{X::Y=4}(T0))$

$\neq \sigma_{Y::pos(Y)=1}(\sigma_{X::Y=4}(T0))$

The left hand side contains the subtuple $<Y::4>>$, whereas the $L(YZ)$-collection of the right hand side is empty. The reason is again that the condition $Y::pos(Y)=1$ is the set difference. The left hand side is $\emptyset$, unlike the right hand side contains $<<Z::5>>$, whereas the $L(YZ)$-collection of the right hand side is empty. The reason is again that the condition $X::Y=4$ selects the fix level of the position selecting condition $Y::pos(Y)=1$.

(c) $\sigma_{Y::Z=3}(\sigma_{X::L(Z)[-1]=5}(T0))$

$\neq \sigma_{X::L(Z)[-1]=5}(\sigma_{Y::Z=3}(T0))$

Here, we have again a position selecting and a content selecting condition. $L(Z)[-1]$ describes the Z-component of the last element of the list $L(YZ)$. The result of the left hand side contains an inner singleton and the result of the right hand side is empty. Here, $X::L(Z)[-1]=5$ refers to $(X,L(YZ))$ and has the fix level $(YZ)$.

(d) $\sigma_{X::Y=2}(T0) \neq T0$ except $\sigma_{X::not(Y=2)}(T0)$

Here, except is the set difference. The left hand side is $T0$ and the right hand side the empty set of type $M(X,L(YZ))$. This rule is not of importance for OttoQL, because our set theoretical operations like except are defined only for flat tabments.

To work with optimization rules we need precise definitions. We want to illustrate the level of a DTD by examples of Table 3:

level(H, dd1) = H
level(A, dd1) = A, B, L(M(C, D))
level(C, dd1) = (C, D)
A scheme \( sname1::cond1 \) is called *simple* if \( cond1 \) is a well defined Boolean valued expression and if it contains no positional attribute and each slashed name is of elementary type (TEXT, \( \ldots \)) and each deepest slashed name from \( cond1 \) is at the same level as \( sname1 \). In this case, the condition "contains" no (implicit) existential quantifier.

A condition is called *relational* if it is of type \( sname1::cond1 \) and \( sname1::cond1 \) is simple. A condition \( cond \) without level specification is therefore always equivalent to a relational condition if it contains only elementary slashed names. Therefore, we will call it relational, too.

In the above given tabment \( T0 \), \( Y_Z = 3 \) and \( X_X = 1 \) are relational conditions, contrary to \( X_Y = 4 \). Again, with respect to \( T0 \), \( Y_Z = 3 \) is an abbreviation of \( X_Z = 3 \) followed by \( Y_Z = 3 \). A condition can have zero, one, or several fix levels. The simple condition \( Y_Z = 3 \) (consider tabment \( T0 \)) refers to \( \text{level}(Y) = \text{level}(Z) = Y_Z \) has no fix level. Generally, simple conditions have no fix levels. The "quantified" condition \( X_Y = 4 \) refers to \( (X_L Y_Z) \) that means it selects \( (X_L Y_Z) \)-elements and has the fix level \( Y_Z \). That means the truth value of the condition depends on the \( L(Y_Z) \)-collection. If \( (Y_Z) \)-elements are eliminated by another condition then the truth value for evaluating a level \( X \)-element may change.

A scheme \( lev \) is a fix level of a condition \( sname1::cond1 \) if one of the following cases is satisfied:

1. \( cond1 \) contains an attribute \( C(sname) \), or \( pos(sname) \) with \( \text{lev} = \text{level}(sname) \), or \( att(if) \) where \( att \) is of collection type and \( \text{lev} = \text{level}(\text{type}(att)) \).

2. \( cond1 \) contains an attribute \( att \) such that \( lev \) is superordinated to \( att \) (or at same level) and \( sname1 \) superordinated to \( lev \) and \( lev = \) unequal to \( \text{level}(sname1) \).

3. \( cond1 \) contains a collection name attribute \( cn \) and \( cn \) is of type \( C(lev) \).

We present some illustrating examples with respect to the above DTD \( dd1 \):

1. \( C:: M(G) = M[I:2J] \) has fix level \( (G) \), but not \( (C,D) \) \( (1) \)
2. \( C:: pos(G) < 40 \) has fix level \( (G) \), but not \( (C,D) \) \( (1) \)
3. \( C:: M(G)[3] = 2 \) has fix level \( (G) \), but not \( (C,D) \) \( (1) \)
4. \( A:: G = I \) has fix the levels: \( (G) \) and \( (C,D) \), but not \( (A?,B1,M(C,D)) \) \( (2) \)
5. \( A:: 1 \) in \( M(G) \) has fix level \( (C,D) \) \( (1) \)

6. \( D = L[I:2J] \) has fix level \( H \). \( (3) \)

The "counter examples" presented after each of the following conjectures are not real counter examples. They are to demonstrate that we cannot omit corresponding presuppositions.

**Conjecture 1 (sel–sel1):**

If \( sn1::cl \) does not select in a fix level of \( sn2::c2 \) and \( sn2::c2 \) does not select in a fix level of \( sn1::cl \) then the following holds:

\[
\sigma_{sn2::c2}(\sigma_{sn1::cl}(tab)) = \sigma_{sn1::cl}(\sigma_{sn2::c2}(tab))
\]

**Counter example (e):**

\[
\begin{align*}
\sigma_{Z::Z=3}(\sigma_{Y::Y=M(Z)}[T1]) \\
\sigma_{Y::Y=M(Z)}(\sigma_{Z::Z=3}(T1))
\end{align*}
\]

The right and left hand sides are equal to the following tabments, respectively:

\[
\begin{align*}
\langle M \{ X, M(Y), M(Z) \} : & : \quad 1 \quad 2 \quad 3 \quad \rangle \\
\langle M \{ X, M(Y), M(Z) \} : & : \quad 1 \quad 2 \quad 2 \quad \rangle
\end{align*}
\]

Here, \( Z::Z=3 \) selects in a fix level \( Z \) of \( Y::Y \) in \( M(Z) \).

**Conjecture 2 (sel–sel2):**

If \( sn1::cl \) and \( sn2::c2 \) are relational and all occurrences of attributes from \( sn1 \), \( sn2 \), \( cl \), and \( c2 \) are on one hierarchical path then the following holds:

\[
\sigma_{sn2::c2}(\sigma_{sn1::cl}(tab)) = \sigma_{sn1::cl}(\sigma_{sn2::c2}(tab))
\]

**Constructed counter example (f):**

\[
\begin{align*}
\sigma_{X::X=2}(\sigma_{X::X=M(Z)}[T2]) \\
\sigma_{X::X=2}(\sigma_{X::X=M(Z)}[T2])
\end{align*}
\]

The left hand side is the empty table and the right hand side is:

\[
\begin{align*}
\langle M \{ X, M(Y), M(Z) \} : & : \quad 1 \quad 2 \quad 2 \quad \rangle
\end{align*}
\]

**Non hierarchical path counter example of ordinary type (g):**

\[
\sigma_{Z::Z=4}(\sigma_{Y::Y=M(Z)}[T3]) \\
\neq \sigma_{Y::Y=M(Z)}(\sigma_{Z::Z=4}(T3))
\]

Here the left hand side is again empty and the right hand side is equal to:

\[
\begin{align*}
\langle M \{ X, M(Y), M(Z) \} : & : \quad 1 \quad 2 \quad \rangle
\end{align*}
\]

Here, \( Z \) and \( Y \) are not on a hierarchical path and \( Z::Z=4 \) or \( Y::Y=4 \) is not relational. But, we remark that the following equation holds, because both sides are empty.

\[
\sigma_{Z::Z=4}(\sigma_{Y::Y=M(Z)}[T3]) = \sigma_{Y::Y=M(Z)}(\sigma_{Z::Z=4}(T3))
\]

\[
\begin{align*}
\langle M \{ X, M(Y), M(Z) \} : & : \quad 1 \quad 2 \quad 3 \quad 4 \quad \rangle
\end{align*}
\]
Conjecture 3 (sel–intersect):
If \( t \) is a set or bag and \( sn1::c1 \) and \( sn2::c2 \) refer to the
outmost level and \( sn1::c1 \) is not position selecting then the
following holds:
\[
\sigma_{sn2::c2}(\{\sigma_{sn1::c1}(t)\}) = \sigma_{sn1::c1}(t) \text{ intersect} \sigma_{sn2::c2}(t)
\]

Conjecture 4 (sel–conjunction 1):
If the condition \( sn::c1 \) does not select in a fix level of \( sn::c2 \) and
\( sn::c2 \) does not select in a fix level of \( sn::c1 \) then the
following holds:
\[
\sigma_{sn::c1 and c2}(tab) = \sigma_{sn::c1}(\{\sigma_{sn::c2}(tab)\})
\]

Counter example (h1):
\[
\begin{align*}
\sigma_{Y::Y=2}(\{\sigma_{Y::4inL(Y)}(T0)\}) & \neq \sigma_{Y::4inL(Y)}(\{\sigma_{Y::Y=2}(T0)\}) \\
& \neq \sigma_{Y::4inL(Y)=andY=2(T0)}
\end{align*}
\]
The first and third expression is equal to the first following
tabment and the second to the second following:
\[
\begin{align*}
\sigma & (X, L( Y, Z)) : <> \\
\sigma & (X, L( Y, Z)) : >>
\end{align*}
\]

Counter example (h2):
\[
\begin{align*}
\sigma_{X::Y=2}(\{\sigma_{X::Z=5}(T0)\}) & \neq \sigma_{X::Y=2}(\{\sigma_{X::Z=5}(T0)\}) \\
& \neq \sigma_{X::Y=2 andZ=5(T0)}
\end{align*}
\]
The result of the two first expressions is \( T0 \) and the result of the third is empty.

Conjecture 5 (sel–conjunction 2):
If \( sn::c1 \) and \( sn::c2 \) are relational and all occurrences of
attributes from \( sn, c1, \) and \( c2 \) are on one hierarchical path
then the following holds:
\[
\sigma_{sn::c1 and c2}(tab) = \sigma_{sn::c1}(\{\sigma_{sn::c2}(tab)\})
\]

Conjecture 6 (sel–conjunction 3):
If \( sn::c1 \) is not position selecting and \( sn \) determines the
outmost level of a given set, bag, or list \( tab \) then the following holds:
\[
\sigma_{sn::c1 and c2}(tab) \in2 \sigma_{sn::c1}(\{\sigma_{sn::c2}(tab)\})
\]

Here, \( \in2 \) is the set (bag) (list) theoretic inclusion.

Counter example (inequality) (i):
\[
\sigma_{X::Y=2 andZ=5(T0)} \neq \sigma_{X::Y=2}(\{\sigma_{X::Z=5}(T0)\})
\]

Here, the left hand side is the empty tabment and the right
hand side results in \( T0 \).

Conjecture 7 (absorb–sel):
If \( sn1::c1 \) is a simple condition and \( c2 \) contains only
slashed names as attributes and \( sn1::c1 \) refers to a fix level
of \( sn2::c2 \) then the following holds:
\[
\sigma_{sn2::c2}(\{\sigma_{sn1::c1}(tab)\}) = \sigma_{sn1::c1}(\{\sigma_{sn2::c1 and c2}(tab)\})
\]

Counter example (absorb–sel) (j):
\[
\begin{align*}
\sigma_{X::4inM(Y)}(\{\sigma_{Y::Y=2}(T0)\}) & \neq \sigma_{Y::Y=2}(\{\sigma_{X::4inM(Y)=andY=2(T0)}\})
\end{align*}
\]

Here, the left hand side is empty and the right hand side
equal to the following tabment:
\[
\begin{align*}
\sigma & (X, M( Y, Z)) : <> \\
\sigma & (X, M( Y, Z)) : >>
\end{align*}
\]

Conjecture 8 (::–condition-to–::–condition):
Assume \( sn1::c1 \) and \( sn2::c2 \) are relational conditions and
\( sn2 \) is deeper than \( sn1 \) then the following holds:
\[
\sigma_{sn2::c2}(\{\sigma_{sn1::c1}(tab)\}) = \sigma_{sn2::c2}(\{\sigma_{sn1::c1}(tab)\})
\]

8 Related work
XQuery [Boag et al., 2007] is a very powerful, well understood
computer language for XML files and collections
of XML files. But, XQuery seems to be more complicated
than SQL. Therefore, we do not believe that in future the
number of XQuery users will exceed the number of SQL
users. We believe that OttoQL is more easy to use for
a broad class of queries than XQuery and even SQL.
We trace this back mainly to our simple syntax. Our semantic
is more complicated than the semantic of SQL.
In XAL [Frasincar et al., 2002] and most other languages
and algebras the select operation is in general commutative.
[Li et al., 2004] is an interesting article with similar aims
as ours. Here, it is tried to generalize XQuery in a way that
the user has not to know the structure (DTD) of the
given documents in detail. Or in other words that one query can
be applied to several XML documents of a similar structure.
They use three queries and two DTD’s to illustrate
their theory. Here, the formulation of the second of three
queries of [Li et al., 2004] in OttoQL follows:

A: BIBLIOGRAPHY = BIB*
B: BIBLIOGRAPHY = BIB*
BOOK = TITLE, AUTHOR*
ARTICLE = TITLE, AUTHOR*
BOOK = (YEAR, BOOK*, ARTICLE*)
ARTICLE = YEAR, TITLE, AUTHOR*
BOOK = TITLE, AUTHOR*
ARTICLE = YEAR, TITLE, AUTHOR*
Query 2: Find additional authors of the publications, of which Mary is an author.

```
aus doc("bib.xml")
m HarryaTitle::AUTHOR="Mary"
ohne AUTHOR::AUTHOR="Mary"
gib M(AUTHOR)
```

In [Li et al., 2004] a MCLAS (Meaningful Lowest Common Ancestor Structure) function is used to express these queries. As in our examples, the formulation of these queries does not require knowledge about the exact structure of the document and the tags BIBLIOGRAPHY, BIB, BOOK, and ARTICLE are not needed, too. But in [Li et al., 2004], contrary to our approach, these tags are needed to find the ancestors.

In [Bast and Weber, 2007] IR goes one step into DB-integration. Here, Bast and Weber start with an efficient algorithm and data structure and then they develop a user interface. In OttoQL we started to develop a user language, example by example, and now we try to develop a theory and an efficient implementation. The CompleteSearch Engine of [Bast and Weber, 2007] is a full text retrieval system, which uses tags and the join. The system does not support aggregations and does not allow restructuring. We present the query: Which German chancellors had an audience with the pope?

```
{ mit "german" and "chancellor"
gib M(politician)
intersect
{ mit "audience" and "pope"
gib M(politician)}
```

9 SUMMARY OF OttoQL

We summarize the interesting features of our model:

1. Our understanding of XML is based on independent, abstract, generating operations for tuples, collections, choice, elementary values, and tags. Therefore, we could define and widely implement powerful and easy to use operations. The use of these generating operations seems to be the reason that our OCAML programs for selection, stroke, ... are relatively short. So, we think that short OCAML programs are good specifications of our operations.

2. The operation stroke (gib part) allows a restructuring of arbitrary XML documents, where only the DTD of the desired XML document is given. It is a procedural and axiomatic generalization of the restructuring operation of [Abiteboul and Bidot, 1986]. Additional to the restrict operation, stroke allows to realize aggregations on arbitrary levels, simultaneously. A non-first-normal-form predecessor version of stroke is described in [Benecke, 1991]. One of the best examples of OttoQL is query 1.1.9.4 Q4 of [D. Chamberlain et. al., 2000]. The presented XQuery query needs more than 20 lines. With the help of OttoQL it can be realized in the following way:

```
aus doc("bib.xml")
gib rs rs=M(r) r=authorLTR(title)
```

3. The select operation is very powerful, but nevertheless syntactically easy to use. It is based on the ext operation. Therefore it can be widely applied also independently from the given structure (DTD). This will be an advantage, if we apply a select operation on a collection of XML documents of different types. Especially tuple tags and collections tags are not necessary to formulate corresponding selections.

4. Because of new optimization rules, as yet unproven, new optimization strategies have to be developed.

Acknowledgments

I would like to thank Dmitri Schamschurko and Milan Gnjatovic for the Caml-light implementation of our forerunner nested relational algebra version of OttoQL. Further thanks are directed to Martin Schnabel and Xuefeng Li, who improved our ongoing implementation in several important points and to Rolf Aretz, who implements now our h2o-file concept [Benecke, 2008], which is intended to replace very large XML files. Finally, I thank the anonymous referees for their valuable remarks.

References


