Does the School Need a Tabular Computer Language?

Klaus Benecke, Andreas Hauptmann
Faculty of Computer Science
Otto-von-Guericke-University, Magdeburg, Germany

Abstract

In this paper it is argued that the school needs a universal computer language for all classes and all subjects. This language should be applicable on tables, documents, and sets of documents. Further, it should allow graphical representations and should be built on concepts, which are as simple as possible. If corresponding operations have no simple definition, then they should be useable at least in a simple way. We believe that the corresponding operations of this language should be taught not only in higher schools, but also in basic ones. As single data operations they should be teachable not only with computer but also on a sheet of paper. If we look carefully to the examples of the paper we will observe that our language OttoQL can even be used in very early classes of the primary school. We believe that XML with the understanding of OttoQL is a good starting point for such a universal computer language. OttoQL is implemented over XML-files and tables and has some efficient operations, which are easy to use, as we believe, and a very promising file concept (h2o). Nevertheless, we believe that OttoQL and its efficiency can be improved by further concepts.

Keywords: programming language for school, XML, table, document, query, end-user

1. Introduction

A computer language for broad user classes and the school should satisfy the following requirements:

1. It should have a simple unified syntax for a lot of applications
2. It should have a precise semantic, which is based on simple algorithms or on simple to use concepts
3. It should allow queries on tables (databases) and documents
4. It should allow queries on collections of documents (IR-systems)
5. It should allow queries on an Inter- and on Intranets of (*XML*-) documents
6. It should allow computations by naive algorithms
7. It would be nice, if it allows to express corresponding laws for taxes etc.
8. It should be useable by people with little interest for mathematics and computer science
9. It should offer sophisticated concepts for broad application classes such that it suits also for users with great interest for mathematics and computer science
10. It should integrate single data and mass data operations
11. It would be nice, if it can exploit graphical features on the base of tables
12. It should be efficiently implementable
13. It should be possible to optimize at least the kernel of the language

Most people, who need information from a computer, use Google or other search engines. They specify one or more words and get in general thousand or million answers. There are weak possibilities to formulate the query more precise, to reduce the answer set.

Most people, who need information from a computer, which are not from the Internet, get it by a program written by computer scientists. Sometimes they set some parameter, but the input data of the program and the computer program itself is invisible for them. Therefore, it is difficult for them to interpret the resulting data correctly. Assume the computer replies the number 2400 as the average salary of metal workers. If the user does not know, whether the column SALARY allows NULLs, or / and whether self-employed business man are included in the file or in the program, then the user can not interpret the result correctly, and he cannot modify the program in order to get the wanted number.

Most people, who solve computational problems with computers, use *EXCEL* or similar spreadsheet programs. *EXCEL* and most other programs are special purpose programs. That means for each problem class (computing with spreadsheets, or writing a document, or making a simple graphic, etc.) the user has to learn a new program with new concepts, menus, and syntax, etc.. Once more, a user, who has not designed a spreadsheet, can hardly interpret the resulting data, because it is not visible, in which order the result is computed. Further, a spreadsheet may contain thousands of expressions, which may be created by copy and paste. In *EXCEL*, expressions can be copied to a whole row or column, for example. Therefore, it seems that *EXCEL* uses mass data

operations. However, a criminal can modify one or more cells in the copied row or column after it and nobody can observe it. This is not possible in a program which uses several mass data operations. Here, you can only change a whole (mass data) formula and this is not hidden like in a spreadsheet.

In this paper we present the main features of our end-user language *OttoQL* [5] with the help of different problems, which do not require deep mathematical feeling. The today's implementation of *OttoQL* is written in *OCAML* and realized for *XML*-documents and tables (TAB-files). We have our own understanding of *XML*-documents, which we call tabments (TABle+docuMENT). Our tabments need a *DTD* (Document Type Definition). In tabments we distinguish not only in the *DTD* but also in primary data between collections and tuples. Further, a root is not necessarily needed for a tabment.

We introduce *OttoQL* by examples, which are also presented in [5] and can be tested there.

2. OttoQL queries on tables

Program 1: BMI.otto

Find the average BMI and the average per age of all people over 20; sort by age and within each age group by BMI.

```
<<L(NAME,   LENGTH, L(AGE,   WEIGHT))::
    Klaus   1.68      18     61
                      30     65
                      56     80
    Rolf    1.78      40     72
    Kathi   1.70      18     55
                      40     70
    Walleri 1.00       3     16
    Viktor  1.61      13     51
    Bert    1.72      18     66
                      30     70 >>
mit NAME: AGE>20
ext BMI:=(WEIGHT div LENGTH**2)
gib BMIAVG,M(AGE,BMIAVG,B(BMI,NAME)) &&
    BMIAVG:=avg(BMI)
round 2
```

"&&" connects two lines to a logical unit.

Result as a table:

```
<<BMIAVG,M(AGE,BMIAVG, B(BMI,  NAME))::
  23.12    18  20.98     19.03 Kathi
                         21.61 Klaus
                         22.31 Bert
           30  23.34     23.03 Klaus
                         23.66 Bert
           40  23.47     22.72 Rolf
                        24.22 Kathi
           56  28.34     28.34 Klaus>>
```

Result as an (XML) document:

```
<root>
  <BMIAVG>23.12</BMIAVG>
  <AGE>18</AGE>
  <BMIAVG>20.98</BMIAVG>
  <BMI>19.03</BMI>
  <NAME>Kathi</NAME>
  <BMI>21.61</BMI>
  <NAME>Klaus</NAME>
  <BMI>22.31</BMI>
  <NAME>Bert</NAME>
  <AGE>30</AGE>
  <BMIAVG>23.34</BMIAVG>
  <BMI>23.03</BMI>
  <NAME>Klaus</NAME>
  <BMI>23.66</BMI>
  <NAME>Bert</NAME>
  <AGE>40</AGE>
  <BMIAVG>23.47</BMIAVG>
  <BMI>22.72</BMI>
  <NAME>Rolf</NAME>
  <BMI>24.22</BMI>
  <NAME>Kathi</NAME>
  <AGE>56</AGE>
  <BMIAVG>28.34</BMIAVG>
  <BMI>28.34</BMI>
  <NAME>Klaus</NAME>
</root>
```

The given table consists of 6 tuples, whereby the first tuple has 3 subtuples. On the given table a condition is applied, which selects only complete tuples. Here, Walleri and Viktor are eliminated. If we apply the condition

```
mit AGE>20
```

to the given table, then not only the two children but also the first subtuple of Klaus, Kathi, and Bert is eliminated. This condition selects persons and (AGE, WEIGHT)-pairs. Because we can also use "ohne" (without) instead of "mit" (with), we need no direct "for all quantifier" in conditions. Thereafter the extension follows:

```
ext BMI:=(WEIGHT div LENGTH**2)
```

It computes for each (WEIGHT, LENGTH)-pair a BMI, such that a table of type L(NAME, LENGTH, L(AGE, WEIGHT, BMI)) results. The special feature of this extension is that we can use WEIGHT and LENGTH in one expression without further specification. This is because both column names are on one hierarchical path. That means that the system multiplies for example not only the weight 61 from Klaus with 1.68, but also the weight 65; although 1.68 does not appear in this row. This correspondence information is described in the first line (the structure of the table) .

The most powerful operation of our data model is stroke (gib). “gib” is German and means “give me”. It allows a restructuring of tables and arbitrary *XML*-documents only by specifying the scheme or *DTD* of the wanted document. In *OttoQL* "M" abbreviates set (German: Menge), "B" Bag, and "L" List. Sets and bags are sorted and lists maintain the given ordering. In sets duplicates in elementary fields (here AGE) are eliminated. By “M-“, “B-“, “L-“ the result is sorted descending.

The above *XML*-document cannot conveniently be used by other systems, because tuple- and subtuple tags are missing. This can be improved by the following gib-part.

```
gib TOTALFILE TOTALFILE=BMIAVG, &&
```

```
                  M(AGEGROUP) &&
    AGEGROUP=AGE,BMIAVG,B(PAIR) &&
    PAIR=BMI,NAME &&
    BMIAVG:=avg(BMI)
```

Finally "round" takes as first input the result of "gib" and the second argument is 2. It rounds all floats of the considered tabment. Non-floats remain unchanged.

Program 2a: bottle_and_cork_a.otto

A bottle and a cork cost altogether 1 mark ten. The bottle is 1 mark more expensive than the cork. How much costs the bottle and how much the cork?

```
aus <<L(CORK):: 0 to 110>>
ext BOTTLE:=CORK+100
mit BOTTLE+CORK=110
   Result:
<<L(CORK, BOTTLE)::
    5     105>>
```

By the first line is assumed that the cork costs 0,1,2,3,.., or 110 pfennigs. The lesson, which we want to learn with this example and some of the following ones, is, because the hardware is fast, we can often use slow algorithms, which are easy to specify, instead of efficient ones, which are hard to write and understand. Now, we consider the problem as Diophantine equation directly.

Program 2b: bottle_and_cork_b.otto

```
aus <<L(CORK):: 0 to 10>>
ext <<L(BOTTLE):: 0 to 110>> at CORK
mit BOTTLE=CORK+100 and BOTTLE+CORK=110
```

Program 3a: interests.otto

How grows an amount of one mark with 2.5 percent interests since the birth of Otto the first?

```
aus <<L(YEAR):: 912 to 2011>>
recext AMNT:=[1.;pred(AMNT)*1.025] &&
       at YEAR
mit YEAR mod 100 = 0 &&
    or YEAR in M[912;922;2011]
round 2
```

Result:

```
<<L(YEAR, AMNT)::
     912            1.
     922            1.28
    1000            8.78
    1100          103.77
    1200         1225.95
    1300        14483.04
    1400       171098.54
    1500      2021309.65
    1600     23879178.92
    1700    282101846.42
    1800   3332671195.7
    1900  39371232197.3
    2000  465120569571.
    2011  610278493603.>>
```

In our opinion this is the simplest solution for this problem. In the recursive extension we have to specify at first the value of the initial element and then the value of all following ones, where it is allowed to use the value of the predecessor.

Program 3b: interests2.otto

Show the development of interests for 10 Euro after 10 years and 11 days with 8.5 percent.

```
ext INTS:=1.085
ext <<L(YEAR):: 0 to 10>>
recext AMNT:=[10.;pred(AMNT)*INTS] &&
       at YEAR
ext <<L(DAY):: 0 to 11>>
recext DAYAMNT:=[L-(AMNT)[1]; &&
    pred(DAYAMNT)*INTS**(1 div 365)] &&
    at DAY
```

It results a tabment of type

INTS, L(YEAR, AMNT), L(DAY, DAYAMNT), where the first list contains 11 and the second 12 elements. Corresponding selections can be added. It is obvious that this program can be taught, contrary to program 3a only in higher classes.

Program 4: one single data operation

```
3+4
```

or

```
3
+4
```

Program 5: plus.otto

Add the 4 to a tabment.

```
3,5,"XX",9
+4
```

Result:

```
<<ZAHL, ZAHL, TEXT, ZAHL::
  7     9     XX    13>>
```

"ZAHL" is German and means number.

Program 6: simple_average.otto

```
<<L(X)::
    1 3 4 3 4 2 2>>
avg
```

Result:

```
<<PZAHL::2.71428571429>>
```

"PZAHL" is a number with a ``Punkt`` (German) (float).

Program 7: average_sum.otto

Compute the average and sum of several numbers.

```
<<L(X)::
    1 3 4 3 4 2 2>>
ext Y:=avg(L(X))
ext Z:=sum(L(X))
forget X
```

Result:

```
<<Y,            Z::
  2.71428571429 19>>
```

Program 8: succ.otto

Find all pupils who have 3 equal successive marks in one subject.

```
aus doc("pupils.xml")
mit NAME:MARK=succ(MARK) &&
        and MARK=pred(MARK)
```

Here, "pupils.xml" is up to additional tags of type M(NAME, CLASS, M(SUBJECT, L(MARK))).

Program 9: last.otto

Find all pupils with all marks, whose last two math marks are 1.

```
aus doc("pupils.xml")
mit NAME:L-(MARK)[1]=1 and  &&
    L-(MARK)[2]=1 and SUBJECT="Maths"
```

Program 10: last.otto

Compute the median and average of all marks of each pupil.

```
aus doc("pupils.xml")
gib B(NAME, AVERAGE, B(MARK)) &&
    AVERAGE:=avg(MARK)
mit pos(MARK)>(count(L(MARK)) div &&
    2)-1 and &&
    pos(MARK)<(count(L(MARK))div 2)+1
ext MEDIAN:=avg(L(MARK))
round 2
```

3. OttoQL in mathematical education

Because *OttoQL* needs mathematical concepts, it should be taught not only in computer science lessons, but also in mathematics. Moreover because a lot of knowledge, which is relevant for school, can be represented by tables or documents, it can even be used in geography, physics, chemistry, or English,... . So pupils can query for capitals of countries, densities of metals, or dates of poets etc. .

In this section we only want to present some further simple examples, which can be easily expressed in *OttoQL*, but which replace in a certain sense relative complex theories from a point of view of application.

Program 11a: nulls.otto

Find the (integer) Nulls of the polynomial X2 – 15X+56.

```
<<L(X):: -100 to 100>>
mit X**2 - 15 * X + 56 =0
```

Program 11b: null_sine_a.otto

Find an approximation of the Null of the sine function in the interval [3, 4].

```
recext L(LE,RI)WHILE((RI &&
 - LE)>0.0001):= &&
 [3.,4.; &&
 CASE pred(LE)when &&
     (sin((pred(LE)+pred(RI))div 2)&&
 <0.0001)(pred(LE)+pred(RI))div 2 &&
 when(true)ENDCASE,&&
 CASE pred(RI)when &&
 (sin((pred(LE)+pred(RI))div 2)&&
 >0.0001)(pred(LE)+pred(RI))div 2 &&
 when(true)ENDCASE]
mit pos-(LE)=1
```

The selection can also be omitted, because the intermediate table is not too large. Here, a list of right and left pairs is constructed, where left and right are always taken from one of the values: pred(LEFT), pred(RIGHT) or (pred(LEFT) + pred(RIGHT)) div 2.

Program 11c: null_sine_b.otto

Find an approximation of the Null of the sine function in the interval [3, 4] in a yet simpler way.

```
<<L(X):: 3 to 4 step 0.00001>>
mit sin(X)>0
mit pos-(X)=1
```

Program 12: integral_sine.otto

Find the area under the sine function in the interval [0, 3.14159].

```
aus <<L(X):: 0 to 3.14159 step 0.0001>>
ext RECTANG:=sin(X+0.0001 div 2)*0.0001
forget X
sum
```

Program 13a: max_sine_a.otto

Find the maximum of the sine function in the interval [0, 3].

```
recext L(LE,RI) while &&
        (RI - LE > 0.0001) := &&
[0.         ,3.; &&
case pred(LE) when (sin((pred(RI) + &&
           pred(LE)) div 2 +0.001) &&
- sin((pred(RI)+pred(LE))div 2)<0.) &&
((pred(RI) + pred(LE)) div 2)  when &&
                   (true) endcase, &&
case pred(RI) when (sin((pred(RI) + &&
           pred(LE)) div 2 +0.001) &&
- sin((pred(RI)+pred(LE))div 2)>0.) &&
((pred(RI) + pred(LE)) div 2)  when &&
                    (true) endcase]
ext PI:=LE*2 at RI
```

Program 13b: max_sine_b.otto

Find the maximum of the sine function in the interval [0, 3] in a yet simpler way.

```
aus <<L(X):: 0. to 3. step 0.00001>>
mit sin(X+0.000001) - sin(X) >= 0.
mit pos-(X)=1
ext Y:=sin(X)
```

The above examples demonstrate that the applications of integral- and differential calculus can be easily handled with *OttoQL*. We shall simplify the syntax of *OttoQL* a little bit, such that the last but one query will look in future a little bit simpler.

Program 14: derivation_sine.otto

Draw the sine function and an approximation of the first derivation in the interval [0, 4].

```
aus <<L(X)::0 to 4 step 0.01>>
ext DEL:=0.0001
ext SINE:=sin(X)
ext DERIVATION:=(sin(X+DEL)- sin(X)) &&
                div DEL
forget DEL
tag0 XX
ext FORMAT:=&&
<<M(SELECT,     TYPE,  L(STYLE))::
    XX          svg        >>
```

Here, you have to click "graphic" [5]. This example is for higher classes only. The interesting fact is that the precision of floats for graphical representations is sufficient. The limit is not needed, to illustrate the differential quotient. The result of the above program is of type L(X, SINE, DERIVATION). All pairs (X, SINE) and (X, DERIVATION) are interpreted in the "graphic" mode as points, which are drawn.

With the following examples about the multiplication operation we want to show that *OttoQL* and corresponding algorithms can be used also in very early classes, also without computer.

Program 15a: three_times_four_a.otto

Anne, Paul, Ernst, and Susi each got 3 apples. How many apples are this altogether?

```
aus <<L(CHILD)::
```

```
        Anne
        Ernst
        Paul
        Susi>>
ext <<L(APPEL)::
        | | |>> at CHILD
gib L(APPEL)
count
```

BAR is a data type, which contains only one value, which is only useful for school applications. If we consider the result of the extension, it becomes visible that the multiplication computes the area of a rectangle.

Program 15b: three_times_four_b.otto

```
aus <<L(CHILD)::
        Anne
        Ernst
        Paul
        Susi>>
ext <<L(APPLE)::
        | | |>> at CHILD
gib COUNT_APPLES &&
    COUNT_APPLES:=count(APPLE)
```

Program 15c: three_times_four_c.otto

```
aus <<L(CHILD)::
        Anne
        Ernst
        Paul
        Susi>>
recext APPS:=[3;pred(APPS)+3] at CHILD
mit pos-(APPS)=1
gib APPS
```

In one of the class levels, where straight lines are treated the widow rent as a connection of several lines could be treated in mathematic lessons, too.

Program 16: widow_pension_east2.otto

```
ext NULL:=0
ext <<L(MY_PEN)::1 to 3000>> # net
ext PARTNER_PEN:=1200.80     # net
ext GREAT_WIDOW_PEN:=0.55 * PARTNER_PEN
ext WIDOW_PEN:=case GREAT_WIDOW_PEN &&
    when(MY_PEN < 637.03) &&
    GREAT_WIDOW_PEN - &&
    (MY_PEN - 637.03)*0.4 &&
 when (GREAT_WIDOW_PEN - &&
 (MY_PENSION - 637.03)*0.4 > 0.) &&
    0. when(true) endcase at MY_PEN
gib L(MY_PEN,NULL,MY_PEN,WIDOW_PEN)
```

The great widow pension is for people, who are older than 45 years.

Program 17: flag_german.otto

```
ext R1:=<<L(X):: 1.6 to 5 step 0.05>>
ext <<L(Y)::2 + 0.3*sin(X) to 3 + &&
            0.3*sin(X) step 0.05>> at X
ext R2:=R1 -tup (0,1)
ext R3:=R1 -tup (0,2)
ext RGB:=(0.,0.,0.) leftat R1 # black
ext RGB:=(1.,0.,0.) leftat R2 # red
ext RGB:=(1.,1.,0.) leftat R3 # yellow
```

By the first two lines a tabment R1 of type L(X, L(Y)) is generated. This tabment is extended by R2, which is also of type L(X, L(Y)). Finally a tabment of type (RGB, R1, RGB, R2, RGB, R3) results. Because of the sine function the flag is flying. The style data are omitted in this program. Each point gets the color of its last preceding RGB-value.

4. OttoQL queries on documents

Program 18: serial_letter.otto

Fit a letter to several persons

```
aus doc("persons3.xml")
ext doc("letter3.xml") at PERSON
ext DEAR := case "Lieber" when &&
    (SEX = "m") &&
    "Liebe"  when(true) &&
    endcase at ADDRESS
ext NAME2:=DEAR^" "^FIRSTNAME^","
ext NAME3:=FIRSTNAME^" "^NAME &&
           leftat ADDRESS
replace  ADDRESS by &&
         ADDRESS:=(STREET,ZIP_LOC)
forget PERSON,DEAR
untag0
tag0 LETTERS
ext FORMAT:= &&
<<M(SELECT, TYPE,L(STYLE))::
    LETTERS div
   LETTER div   page-break-after:always
    DATE    div   margin-left:80%
    SENDER  div   margin-bottom:2em
    ADDRESS div   margin-bottom:5em
    NAME    div
    FETT    span  bold
    P       span  width:20%
    KB      div>>
```

Here "person3.xml" is a list of PERSONs, on which additionally a selection could be applied, if desired. By the first extension the letter is duplicated for each person. This is the most interesting part of this program. This duplication is realized by a very general operation, which the user has to learn. But, he can use this operation in a lot of applications. Such a concept is easier to learn and remember than a sequence of clicks. Working with menus and submenus should not be subject of school lessons, if there are no concepts behind. The value of DEAR depends on the SEX-field, which is superordinated to each letter. "^" is the string concatenation. More details you find in [5]. It is evident that this program can be modified to person- and letter files with another structure and other content.

Often a user does not know the exact field (column) name of a table or a document. Therefore it should be possible to specify a query only by one or more words. This can be realized also in *OttoQL*. Additionally, it can be specified, elements of which level have to contain the corresponding word.

Program 19: saale.otto

Find all rivers with all data, which contain the word "Saale".

```
aus doc("fluesse.xml") # rivers
mit "Saale"
```

Here, the condition is an abbreviation of "Saale" in worte(tup(NAME)), where "NAME" is a toplevel field of "fluesse.xml" (“Worte”=”words” (German)).

5. Related work

XQuery [7], 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*. Let us consider our first example in *XQuery*:

BMI-example in *XQuery*:

```
let $persons:=(<PERSONS>
 <PERSON>
   <NAME>Klaus</NAME>
   <LENGTH>1.68</LENGTH>
   <SUBTUP>
     <AGE>18</AGE>
     <WEIGHT>61</WEIGHT>
   </SUBTUP>
   <SUBTUP>
     <AGE>30</AGE>
     <WEIGHT>65</WEIGHT>
   </SUBTUP>
   <SUBTUP>
     <AGE>56</AGE>
     <WEIGHT>80</WEIGHT>
   </SUBTUP>
 </PERSON>
 <PERSON>
   <NAME>Rolf</NAME>
   <LENGTH>1.78</LENGTH>
   <SUBTUP>
     <AGE>40</AGE>
     <WEIGHT>72</WEIGHT>
   </SUBTUP>
 </PERSON>
 <PERSON>
   <NAME>Kathi</NAME>
   <LENGTH>1.7</LENGTH>
   <SUBTUP>
     <AGE>18</AGE>
     <WEIGHT>55</WEIGHT>
   </SUBTUP>
   <SUBTUP>
     <AGE>40</AGE>
     <WEIGHT>70</WEIGHT>
   </SUBTUP>
 </PERSON>
 <PERSON>
   <NAME>Walleri</NAME>
   <LENGTH>1.</LENGTH>
   <SUBTUP>
     <AGE>3</AGE>
     <WEIGHT>16</WEIGHT>
   </SUBTUP>
 </PERSON>
 <PERSON>
   <NAME>Viktor</NAME>
   <LENGTH>1.61</LENGTH>
   <SUBTUP>
     <AGE>13</AGE>
     <WEIGHT>51</WEIGHT>
   </SUBTUP>
 </PERSON>
 <PERSON>
   <NAME>Bert</NAME>
   <LENGTH>1.72</LENGTH>
   <SUBTUP>
     <AGE>18</AGE>
     <WEIGHT>66</WEIGHT>
   </SUBTUP>
   <SUBTUP>
     <AGE>30</AGE>
     <WEIGHT>70</WEIGHT>
   </SUBTUP>
 </PERSON>
</PERSONS>//PERSON[.//AGE>=20])
let $bmis:=(
  for $p in $persons
  return <PER> {$p/NAME}
               { for $t in $p/SUBTUP
                 return <TU> {$t/AGE}
  <BMI>{$t/WEIGHT    div($p/LENGTH    *
$p/LENGTH)}</BMI></TU>
                       }
                  </PER>)
return  <results><BMIDUR>{round-half-to-
even(avg($bmis//BMI),2)}</BMIDUR>
         {   for   $a   in   distinct-
values($bmis//AGE)
           order by $a
           return <AG>
                    <AGE> {$a} </AGE>
                    <BMIDUR>{round-
half-to-even(avg($bmis//TU
[AGE = $a]/BMI),2)}</BMIDUR>
       { for $p2 in $bmis[.//AGE=$a]
         for $b in $p2//TU[AGE=$a]//BMI
         order by $b,$p2/NAME
return<TU2><BMI>{round-half-to-
even($b,2)}</BMI>{$p2/NAME}</TU2>
                    }
                 </AG>
}
</results>
```

In *XAL* [8] and most other languages and algebras the select operation is in general commutative, contrary to our approach. Nevertheless, we have optimization possibilities, too; see [4] and [10].

Li, Yu, and Jagadish [9], 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 in this paper and two *DTD*'s to illustrate their theory. Here, we will consider the second example of [9], where two similar *DTD*'s of "Bib.xml" are given:

```
A: BIBLIOGRAPHY = BIB*
   BIB = (YEAR, BOOK*, ARTICLE*)
   BOOK = TITLE, AUTHOR*
   ARTICLE = TITLE, AUTHOR*
B: BIBLIOGRAPHY = BIB*
   BIB = (BOOK* | ARTICLE*)
   BOOK = YEAR, TITLE, AUTHOR*
   ARTICLE = YEAR, TITLE, AUTHOR*
```

Query 2: Find additional authors of the publications, of which Mary is an author.

```
for $a in doc("Bib.xml")//AUTHOR,
    $b in doc("Bib.xml")//AUTHOR
where $a/text() ="Mary"
    and $a != $b
    and exists mclas($a, $b)
return $b
```

The same query in *OttoQL*:

```
aus  doc("Bib.xml")
mit  TITLE:AUTHOR="Mary"
ohne AUTHOR:AUTHOR="Mary"
gib  B(AUTHOR)
```

In [9] 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 [9], contrary to our approach, the system needs these tags to find corresponding ancestors.

According to Bast and Weber [1], IR goes one step into DB-integration. Here, Bast and Weber started with efficient algorithms and data structures and then they developed a user interface. In *OttoQL* we started to develop an end-user language, example by example, and now we try to develop a theory and an efficient implementation. The *Complete Search Engine* of [1] is a full text retrieval system, which uses tags and joins. The system does not support aggregations and does not allow restructuring. We present the main example of the paper: Which German chancellors had an audience with the pope?

```
german chancellor politician:
audience pope politician:
```

The program in *OttoQL*, if a search engine is implemented:

```
aus doc("xml_search_engine.h2o")
mit "german" and "chancellor"
gib M(politician)
intersect &&
 {aus doc("xml_search_engine.h2o")
  mit "audience" and "pope"
  gib M(politician)}
```

6. Conclusion

"Information is the reduction of uncertainty"; it "aids for complexity reduction are abstraction and modeling" [12]. In [11] the abstract algebraic background for our theory is presented and in [2] and [6] a light version of a part of *OttoQL* is specified.

The greatest restrictions of a computer language in the future are not the software and hardware restrictions, but the ability of people to learn and remember the concepts and syntax of the language. We should teach already now easy to specify and universal algorithms instead of efficient ones, because in future we will be able to implement them more efficiently (This does not mean that we are up to now unsuccessful on this area).

The main contribution of our data model with *OttoQL* to i-society is that we bring the old notion of "repeating group" in new clothes to the end-user. The future will show, whether the people will accept this kind of universal abstract model.

We are working now on a very universal file concept [3] and try to generalize the *join* operation . Further, we will start to implement optimization strategies [10].

Further, the next important task is to test our current implementation of *OttoQL* in German and other schools. We have to comprehend, which of the presented concepts are easy to learn and understand and where are the problems. For this a lot of methodical and didactical problems have to be solved.

7. Acknowledgements

The authors would like to thank D. Schamschurko and M. Schnabel for their valuable contributions to the implementation of *OttoQL*.

8. References

[1] H. Bast and I. Weber, “The Complete Search Engine: Interactive, Efficient, and Towards IR & DB Integration”, CIDR, 2007, pp. 88-95

[2] K. Benecke, “A powerful tool for object-oriented manipulation” in On Object Oriented Database: Analysis, Design & construction, IFIP TC2/WG 2.6 Working conference, July 1991, pp. 95-121

[3] K. Benecke, “A first View to the H2O Storage structure - The Marriage of the TID-Concept with the XML-File Structure”, Technical Report University Magdeburg Nr.:FIN-16-2008, http://www.cs.uni-magdeburg.de/fin_media/downloads/forschung/technical_reports_und_preprints/2008/TechReport16-p-1446.pdf

[4] K. Benecke, “Towards Unifying Selection Mechanisms for DB- and IR-Systems”, Workshop Information Retrieval 2009, Fachgruppe Information Retrieval LWA 2009

[5] K. Benecke and A. Hauptmann and D. Schamschurko and M. Schnabel, “Internet Server for OttoQL”, http://otto.cs.uni-magdeburg.de/otto/web/index.html, 2011

[6] K. Benecke and X. Li, “A Restructuring Operation for XML Documents”, Technical Report University Magdeburg Nr.:FIN-009-2009

[7] S. Boag, D. Chamberlain, M. F. Fernandez, D. Florescu, J. Robie, and J.Simeon, “XQuery 1.0: An XML Query Language”, http://www.w3.org/TR/xquery/, 2007

[8] F. Frasincar and G.-J. Houben and C. Pau “XAL : an Algebra for XML Query Optimization”, ADC 2002, Melbourne Australia, pp. 49-56

[9] Y. Li and C. Yu and H.V. Jagadish “Schema-Free XQuery”, VLDB Conference, 2004. pp. 72-83

[10] A. Hauptmann, “OttoQL: Problems of the Implementation of non-relational Database Languages (with particular considerations of locigal Optimization), (German), Studienarbeit, University Magdeburg, 2010

[11] H. Reichel, “Initial Computability, Algebraic Specifications, and Partial Algebras”, Oxford, UK: Claredon Press, 1987
[12] “Information Society”, Wikipedia, http://de.wikipedia.org/wiki/Informationsgesellschaft

