INFORMATION
SYSTEMS IN
MANAGEMENT

Information Systems in Management (2015) Vol. 4 (1) 3−13

ADAPTING SBA OPTIMIZATION METHODS DEVOTED TO QUERIES HAVING SUBQUERIES TYPED BY ENUMERATIONS FOR XQUERY EXPRESSIONS

MICHAŁ BLEJA

Faculty of Mathematics and Computer Science, University of Lodz

The paper presents the concepts related to the design of query optimization methods for nested queries. The proposed methods are dedicated for queries having so called weakly dependent subqueries. A subquery is weakly dependent from its direct operator if it depends only on expressions typed by enumerations. We have successfully developed the weakly dependent subqueries method in the context of object-oriented database management systems based on Stack-Based Approach. Systems storing XML data which are queried using XQuery can be considered simplified object-oriented databases. For this reason we try to adopt SBQL query optimization methods to XQuery expressions.

Keywords: query optimization, XQuery, enumerations, weakly dependent subquery, Stack-Based Approach, SBQL

1. Introduction

A query optimization [1, 2] is aimed at radical reducing of the query processing time. It must take into consideration both the time needed for the optimization and the processing time after optimizing. There are many various optimization methods in contemporary database management systems. Some of them assume using redundant data structures called indices [3]. Other techniques cope with caching query results [4] and different strategies of physical data organization. The major group of methods concern query rewriting [5, 6, 7, 8].

Rewriting consists in translating an original query into a semantically equivalent form ensuring better performance.

The ODRA (Object Database for Rapid Application Development) [9, 10, 11] prototype is the main component of the European project EGov Bus [12]. It is equipped with a very powerful query optimizer. In particular, the optimizer deals with optimization methods based on query rewriting. Among them we can mention the method of weakly dependent subqueries [5, 6], pushing selection before structure operators [7], the method of independent subqueries [8]. These techniques have been developed in the context of the Stack-Based Approach [13, 14] and its query/programming language SBQL. SBA constitutes a uniform conceptual platform for object-oriented databases which uniformly covers two aspects: querying and programming. It allows to construct optimization methods in their full generality. The rewriting rules presented in [5, 6, 7, 8, 14] deal with any data model (assuming that its semantics would expressed in terms of SBA). In particular, they hold for the relational model, the XML model [15] and any version of object-oriented model. The rules cope also with any operators and make no assumptions concerning the complexity of subqueries of a given query.

The XQuery [16, 17] semantics can be defined using the SBA concepts. It requires introducing three data structures which are essential for the precise semantic description: an object store, an environment stack, and a query result stack. Besides a special phase called static analysis [19, 20] is required to equip XQuery expressions with the information essential for detecting subqueries typed by enumerations. At this moment we translate basic XQuery expressions (so called FLOWR expressions) into their SBQL equivalents. SBQL queries are rewritten by the ODRA optimizer and then converted into XQuery ones. We plan to define the XQuery semantics using SBA concepts in the future.

The research presented in this paper concerns concepts related to optimization of nested queries. We deal with a special class of subqueries of a given query referred to as weakly dependent subqueries [5, 6]. The dependency is considered in the context of query operators such as selection, quantifiers, joins, etc. A subquery is weakly dependent from its direct operator if the dependency concerns only an expression which is typed by the enumeration. The number of evaluations of such the subquery can be limited to the number of enumerators occurring in the enumeration on which it depends. For instance consider the query which returns each employee earning above the average salary calculated for all employees having his/her gender. Without optimizing the subquery calculating the average salary for genders would be evaluated hundreds or thousands of times, while it could be evaluated only 2 times (once assuming *gender* = "male" and next one assuming *gender* = "female"). This subquery is a classical example of a weakly dependent subquery.

2. The Stack-Based-Approach

The Stack-Based Approach [13, 14] presents the right theory for object-oriented databases and their query/programming languages. We present these concepts of the Stack-Based Approach which are essential to the development of optimization methods for queries having weakly dependent subqueries. SBA involves the following concepts [13, 14]:

- naming-scoping-binding - each name in a query/program is bound to a suitable run-time entity depending on the scope for it.
- environment stack - it is responsible for binding names, procedure/method calls, scope control.
- total internal identification - each entity must have a unique internal identifier.
- object relativity - objects are treated uniformly and have the same formal properties regardless of the hierarchy level at which they occur.

SBA introduces a family of object store models M0, M1, M2, and M3 [14]. The simplest is M0 which deals with relational and XML-oriented data structures. In the M0 model each object is a triple consisting of an internal identifier, an external name and a value. The M1 model extends M0 with classes and static inheritance. Classes are understood as objects which store invariants (e.g. methods) of their instances. M2 extends M1 by the concept of dynamic object role. M3 augments M2 with the encapsulation mechanism.

SBQL is described in detail in [13, 14]. The syntax of SBQL is as follows:

- A single name or a single literal is an atomic query (e.g. *emp*, *dept*, *salary*, "Smith", 3000).
- If q is a query and θ (e.g. *sum*, *avg*, -) is a unary operator then $\theta(q)$ is a query.
- If q_1 and q_2 are queries and θ is a binary operator (e.g. *where*, =, +, *quantifier*) then $q_1 \theta q_2$ is a query.

XML (Extensible Markup Language) [15] is a flexible text format applied to store and exchange data. There are several database projects dedicated to store XML data. Among them we can mention Oracle XML DB, eXist-db, Apache Xindice. XQuery [16, 17] is a query language for addressing XML data. It navigates through XML documents using XPath [18] expressions. Queries in XQuery are often formulated using so called FLOWR [16, 17] (for, let, order by, where, return) expressions. To present XQuery examples we assume XML documents which correspond to the schema presented in Figure 1. For instance, the following query returns employees earning more than 3000 (The *doc* function is applied to open XML documents):

```
for $emp in doc("company.xml")//emp
where $emp/salary>3000                                    (1)
return $emp
```

Our optimization methods are entirely performed before a query is executed. It requires a special phase called static analysis [14, 19, 20] which simulates run-time actions during compilation-time. It uses an abstract syntax tree (AST) of a given query to perform static type checking. The static analysis acts on three data structures: a metabase, a static environment stack SENVS, and a static query result stack SQRES.

```
<?xml version="1.0"?>
<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema">

     <xs:simpleType name="enum_gender">
      <xs:restriction base="xs:string">
        <xs:enumeration value="male"/>
        <xs:enumeration value="female"/>
      </xs:restriction>
     </xs:simpleType>

     <xs:simpleType name="enum_education">
      <xs:restriction base="xs:string">
        <xs:enumeration value="vocational"/>
        <xs:enumeration value="secondary"/>
        <xs:enumeration value="higher"/>
      </xs:restriction>
     </xs:simpleType>

     <xs:element name="emp">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="fname" type="xs:string"/>
          <xs:element name="lname" type="xs:string"/>
          <xs:element name="salary" type="xs:float"/>
          <xs:element name="gender" type="enum_gender"/>
          <xs:element name="education" type="enum_education"/>
          <xs:element name="overtime" type="xs:positiveInteger"/>
        </xs:sequence>
      </xs:complexType>
     </xs:element>

     <xs:element name="depts">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="dept" maxOccurs="50">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="employs">
                  <xs:complexType>
                    <xs:sequence>
                      <xs:element ref="emp" maxOccurs="unbounded"/>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
     </xs:element>

</xs:schema>
```

Figure 1. Sample XML schema

The metabase models statically the object store. It is generated from an XML schema. SENVS reflects binding operations performed on the run-time environment stack. SQRES stores signatures of the intermediate and final query results. The static analysis causes among others that [14]:

- Each name in a query is equipped with the order number of a stack section which is relevant for binding it.
- Each non-algebraic operator is assigned to the environment stack section(s) which it opens.

In SBA detecting weakly dependent subqueries is based on analyzing scoping and binding rules for names occurring in queries. A subquery is weakly dependent from its direct non-algebraic operator if it involves a name which can be statically bound to an enumerated type in the scope opened by this operator [5, 6]. Other names should be bound outside that scope. A subquery is called independent if none of its names is bound in the stack section opened by its direct operator [8, 14].

3. Optimization of queries involving weakly dependent subqueries

The approach involves the following steps:

- Translating an XQuery expression q_1 into its SBQL equivalent q_2.
- Performing static analysis of q_2 and detecting weakly dependent subqueries.
- Transforming the query q_2 into a semantically form q_3 according to the rewriting rule dedicated to weakly dependent subqueries [5, 6].
- Applying the independent subquery method [8, 14] to q_3. Let q_4 denote the result of applying this method to q_3.
- Converting q_4 into its XQuery counterpart q_5.
- Running q_5 against the underlying XML data store.

The following example presents the general idea of the above approach. The query gets each employee who has the overtime hours greater than the average overtime hours calculated for all employees having his/her education.

```
for $emp in doc("company.xml")//emp
where $emp/overtime>
avg(doc("company.xml")//emp[education=$emp/education]/overtime)    (2)
return $emp
```

Consider the following subquery (3) of (2):

```
avg(doc("company.xml")//emp[education=$emp/education]/overtime)    (3)
```

Without optimizing the subquery (3) will be evaluated once for each employee (it can be thousands of times). However it is clear that (3) can be processed only three times. The expression *$emp/education* in (3) can take only three values: occupational, secondary, and higher. We call the subquery (3) weakly

dependent because it depends from its parent query only on the expression *$emp/education* typed by the enumeration *enum_education* {"occupational", "secondary", "higher"}. How such a query can be detected and how a general rewriting rule should look like?

After translating the query (2) into its SBQL counterpart it takes the form (4). For the query below we determine binding levels for its names and the number of scopes opened by the non-algebraic operators.

```
emp as $emp where $emp . overtime>
 1           2     2  3     3
avg((emp where education=$emp . education).overtime)
      1    3      3        2  4     4     3    3
```
(4)

The operator *as* names each element in a bag or sequence returned by a query. The *group as* operator names the whole query result. For instance, if a query *q* returns a collection *bag*{*e1*, *e2*, *e3*, ... }, then a query *q as aux* returns the collection of binders *bag*{*aux*(*e1*), *aux*(*e2*), *aux*(*e3*), ... }. A query *q group as aux* returns a single binder *aux*(*bag*{*e1*, *e2*, *e3*, ... }). The following subquery of (4)

```
avg((emp where education=$emp.education).overtime)
```
(5)

is weakly dependent from the first *where* operator because it contains only the name *education*(in the expression *$emp.education*) which is statically bound to the enumeration *enum_education* in the scope opened by this operator. Other names in the subquery (5) are not bound in the second stack section. Denote (5) by *wds*(*$emp.education*). Then, the original query (4) *emp as $emp where $emp.overtime*>*wds*(*$emp.education*) can be transformed to (6):

```
emp as $emp where if $emp.education = "vocational" then
                     $emp.overtime>wds("vocational")
                  else if $emp.education = "secondary" then
                          $emp.overtime>wds("secondary")
                       else $emp.overtime>wds("higher")
```
(6)

After unfolding (6) we retrieve the query (7).

```
emp as $emp where
 1          2
if $emp.education = "vocational" then $emp.overtime>
    2  3    3                        2  3     3
   avg((emp where education="vocational").overtime)
         1    3      3                   3    3
else if $emp.education = "secondary" then $emp.overtime>
         2  3   3                          2 3   3
   avg((emp where education="secondary").overtime)
         1    3       3                 3    3
     else $emp.overtime>
            2 3    3
        avg((emp where education="higher").overtime)
              1     3       3             3    3
```
(7)

The general idea of the method is to construct a suitable conditional statement (*if-then-else*). The conditions of the statement are based on the enumerators occurring in a given enumerated type. Is the above transformation advantageous for performance? The previously considered weakly dependent subquery (5) has been replaced by the three independent subqueries: *wds*("vocational"), *wds*("secondary"), and *wds*("higher"). Indeed, all three subqueries of (7) are independent from the first *where* operator. None of their names is bound in the second stack section which is determined by that operator.
After applying the independent subquery method [8, 14] to (7) it takes the form (8).

```
(wds("higher") as $aux3,
wds("secondary") as $aux2,
wds("vocational") as $aux1).
(emp as $emp where                                                    (8)
if $emp.education = "vocational" then $emp.overtime>$aux1
else if $emp.education = "secondary" then $emp.overtime>$aux2
else $emp.overtime>$aux3)
```

In (8) the weakly dependent subquery is evaluated only three times. The query (8) terminates optimization actions - no further transformations are possible by using the above methods. In consequence the query (8) will be converted into its XQuery counterpart:

```
let aux3:=
  avg(doc("company.xml")//emp[education="higher"]/overtime)
let $aux2:=
  avg(doc("company.xml")//emp[education="secondary"]/overtime)
let $aux1:=
  avg(doc("company.xml")//emp[education="vocational"]/overtime)    (9)
for $emp in doc("company.xml")//emp
where if($emp/education="vocational") then $emp/overtime> $aux1
else if($emp/education="secondary") then $emp/overtime> $aux2
else $emp/overtime> $aux3
return $emp
```

4. More general case

The examples in Chapter 3 presented only specific case of our transformation. The subquery (5) was weakly dependent from the most external operator. In general, however, the dependency of a subquery can be considered towards internal operators. The query (10) presents such a case.

```
for $dept in doc("company.xml")//dept
where every $emp in $dept/employs/emp satisfies $emp/salary >
avg(doc("company.xml")//emp[gender=$emp/gender]/salary)          (10)
return $dept
```

The query (10) retrieves departments which each employee has the salary greater than the average salary calculated for all employees having his/her gender. The SBQL equivalent of the query (10) has the form (11):

```
dept as $dept where ($dept.employs.emp as $emp ∀ $emp.salary>
 1              2     2  3    3   3 3          3   3 4  4
avg((emp where gender=$emp.gender).salary))
      1    4     4     3  5  5   4   4
```
(11)

Consider the following subquery of (11):

```
avg((emp where gender=$emp.gender).salary)
```
(12)

The subquery (12) is weakly dependent only from the quantifier operator because it involves the name *$emp* (in the expression *$emp.gender*) that is bound in the scope opened by this operator and *$emp.gender* is typed by the enumeration. According to our rewriting rule [5,6] the proper conditional statement is put directly after the operator on which (12) depends:

```
(dept as $dept) where (($dept . employs . emp as $emp)
  1               2       2   3    3    3  3
∀ ( if ($emp . gender="male") then
3         3  4    4
 ($emp . salary>avg((emp where gender="male").salary))
    3  4    4         1    4     4           4  4
else ($emp.salary>avg((emp where gender="female").salary))))
        3 4  4          1    4      4            4  4
```
(13)

The query (13) illustrates the general rule of transforming a weakly dependent subquery. The rule can be applied both to external and internal operators. As a result of our rewriting the query (13) contains two subqueries

```
avg((emp where gender="male").salary)
```
(14)

```
avg((emp where gender="female").salary)
```
(15)

which are independent both from the quantifier and from the first *where* operator. Denote (14) by *wds*("male") and (15) by *wds*("female"). After applying the independent subquery method [8, 14], the query (13) takes the following form:

```
(wds("female") as $aux1, wds("male") as $aux2).
((dept as $dept) where (($dept.employs.emp as $emp) ∀
( if ($emp.gender="male") then ($emp.salary>$aux2)
else ($emp.salary>$aux1)))
```
(16)

After converting (16) to the proper XQuery expression it takes the form (17):

```
let $aux2:=avg(doc("company.xml")//emp[gender="male"]/salary)
let $aux1:=
  avg(doc("company.xml")//emp[gender="female"]/salary)
for $dept in doc("company.xml")//dept
where every $emp in $dept/employs/emp satisfies
if($emp/gender="male") then $emp/salary > $aux2
else $emp/salary > $aux1
return $dept
```
(17)

5. Rewriting rule

The rewriting rule for queries involving weakly dependent subqueries can be formulated as follows. Let *q* be an XQuery expression of the form (18):

```
for $i in doc(uri)//q1
where q2
return $i
```
(18)

where q_2 has the form $q_2 = \alpha \circ wds(\$i/q_3) \circ \beta$; α and β are some parts of q_2 (maybe empty), $\circ$ is a concatenation of strings, $wds(\$i/q_3)$ is a weakly dependent subquery whose part $\$i/q_3$ depends on the parent query only and is of the enumerated type $ET = \{e_1,e_2,...,e_n\}$. Then the expression (18) is transformed into the following SBQL query:

```
q1 as $i where q2'
```
(19)

where $q_2' = \alpha' \circ wds(\$i.q_3)' \circ \beta'$ is an SBQL equivalent of q_2. The query (19) is rewritten to the form (20) according to the rule presented in [5, 6].

```
q1 as $i where if($i.q3=e1) then α'°wds(e1)'°β'
               else if($i.q3=e2) then α'°wds(e2)'°β'
                    else if($i.q3=e3) then α'°wds(e3)'°β'
                    ......
                        else if($i.q3=en-1) then α'°wds(en-1)'°β'
                             else α'°wds(en)'°β'
```
(20)

After applying the independent subquery method [14] to (20) it takes the form (21):

```
(wds(e1)'group as $aux1, wds(e2)'group as $aux2,...,
wds(en)'group as $auxn).
(q1 where if($i.q3=e1) then α'°$aux1°β'
         else if($i.q3=e2) then α'°$aux2°β'
              else if($i.q3=e3) then α'°$aux3°β'
              ......
                   else if($i.q3=en-1) then α'°$auxn-1°β'
                        else α'°$auxn°β'
```
(21)

The query (21) is converted into the following XQuery expression:

```
let $aux1 := wds(e1)
let $aux2 := wds(e2)
...
let $auxn := wds(en)
for $i in doc(uri)//q1
where if($i/q3=e1) then α°$aux1°β
      else if($i/q3=e2) then α°$aux2°β
           else if($i/q3=e3) then α°$aux3°β
           ......
                else if($i/qn-1=en-1) then α°$auxn-1°β
                     else α°$auxn°β
return$i
```

(22)

6. Conclusions

We have adopted the optimization techniques used for SBQL to XQuery expressions. The approach consists in transforming an XQuery expression into its SBQL equivalent. The result of this operation is next rewritten according to the predefined rule. Finally, the optimized SBQL query is converted into an XQuery expression. The proposed optimization method was aimed at limiting the number of processing of a weakly dependent subquery to the number of enumerators of the enumerated type that the subquery depends on.

In the future we are going to express semantics of XQuery in terms of SBA. It allows to avoid converting XQuery expressions into SBQL ones. The SBQL semantics is based on three data structures: an object store, an environment stack, and a query result stack. It respects several principles such as total internal identification, orthogonal persistence, and compositionality. These features much simplify developing query optimization methods. For this reason it is worth to apply concepts related to the Stack-Based Approach for defining XQuery semantics.

REFERENCES

[1] IoannidisY. E. (1996) *Query Optimization*, Computing Surveys, 28(1), pp. 121-123

[2] Jarke M., Koch J. (1984) *Query Optimization in Database Systems*, ACM Computing Surveys 16(2), pp. 111-152

[3] Kowalski T. et al. (2009) *Optimization of Indices in ODRA*, Proc. 1st ICOODB Conf., pp. 97-118, Germany

[4] Cybula P., Subieta K. (2010) *Query Optimization by Result Caching in the Stack-Based Approach*, ICOODB pp. 40-54

[5] Bleja M., Kowalski T., Adamus R., Subieta K. (2009), *Optimization of Object-Oriented Queries Involving Weakly Dependent Subqueries*, Proc. 2nd ICOODB Conf., pp. 77-94, Switzerland

[6] Bleja M., Stencel K., Subieta K. (2009) *Optimization of object-oriented queries addressing large and small collections*, IMCSIT, pp. 643-650

[7] Drozd M., Bleja M., Stencel K., Subieta K. (2012) *Optimization of Object-Oriented Queries through Pushing Selections*, ADBIS (2) pp. 57-68

[8] Płodzień J., Kraken A. (2000) *Object Query Optimization through Detecting Independent Subqueries*, Information Systems 25(8), pp. 467-490

[9] AdamusR. et al. (2008) *Overview of the Project ODRA*, Proc. 1st ICOODB Conf., pp. 179-197, Germany

[10] Lentner M., Subieta K. (2007) *ODRA: A Next Generation Object-Oriented Environment for Rapid Database Application Development*, Proc. 11th ADBIS Conf., Springer LNCS 4690, pp. 130-140

[11] *ODRA (Object Database for Rapid Application Development) Description and Programmer Manual* (2008), http://www.sbql.Pl/various/ODRA/ODRA_manual.html

[12] *eGov Bus: Advanced e-Government Information Service Bus* (2009), European Commission 6th Framework Programme, IST- 26727, http://www.egov-bus.org/web/guest/home

[13] AdamusR. et al. (2008) *Stack-Based Architecture and Stack-Based Query Language*, Proc. 1st ICOODB Conf., pp.77-95, Berlin

[14] Subieta K. (2005) *Theory and construction of object query languages*, Editors of the Polish-Japanese Institute of Information Technology, 522 pages (in Polish)

[15] *Extensible Markup Language (XML) 1.0* (*Fifth Edition*)(2008) http://www.w3.org/TR/2008/REC-xml-20081126/

[16] BrundageM. *XQuery: The XML Query Language*(2004), Addison-Wesley, 505 pages

[17] *XQuery 1.0: An XML Query Language* (*Second Edition*) (2010) http://www.w3.org/TR/xquery/

[18] *XML Path Language* (*XPath*) *2.0* (*Second Edition*) (2010) http://www.w3.org/TR/xpath20/

[19] Płodzień J., Subieta K. (2001) *Static Analysis of Queries as a Tool for Static Optimization*. Proc. IDEAS Conf., IEEE Computer Society, pp. 117-122

[20] Stencel K. (2006) *Semi-strong Type Checking in Database Programming Languages*. Editors of the Polish-Japanese Institute of Information Technology, 207 pages (in Polish)