Optimization of Object-Oriented Queries through Rewriting Compound Weakly Dependent Subqueries

Michał Bleja[1], Tomasz Kowalski[1,2], and Kazimierz Subieta[3,4]

[1] Faculty of Mathematics and Computer Science, University of Łódź, Poland
blejam@math.uni.lodz.pl
[2] Computer Engineering Department, Technical University of Łódź, Poland
tkowals@kis.p.lodz.pl
[3] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[4] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
subieta@pjwstk.edu.pl

Abstract. A new static optimization method for object-oriented queries is presented. We deal with a special class of subqueries of a given query called "compound weakly dependent subqueries". The dependency is considered in the context of SBQL non-algebraic query operators like selection, projection, join, etc. A subquery is weakly dependent from its nearest non-algebraic operator if it depends only on expressions that can be entirely evaluated on small collections. The subquery is considered compound if the dependency concerns at least two such expressions. The research follows the stack-based approach (SBA) to query languages and its query language SBQL (Stack-Based Query Language). Our optimization method is based on analyzing scoping and binding rules for names occurring in queries.

Keywords: query optimization, weakly dependent subqueries, object-oriented database, stack-based approach, SBQL.

1 Introduction

Efficient query evaluation is a very desirable and frequently critical feature of database management systems. The performance can be supported by various methods (indices, parallel execution, etc.). In this paper we consider query optimization based on query rewriting. Rewriting means transforming a query q_1 into a semantically equivalent query q_2 ensuring much better performance. It consists in locating parts of a query matching some pattern. Such optimization is a compile time-action performed before a query is executed. It requires performing a special phase called *static analysis* [10].

Our optimization method is based on the Stack-Based Approach (SBA) ([11], [12], [13]). SBA and its query language SBQL are the result of investigation into a uniform

[1] The author is a scholarship holder of project entitled "Innovative education ..." supported by European Social Fund.

conceptual platform for an integrated query and programming language for object-oriented databases. Currently a similar approach is developed in the Microsoft LINQ project [8] that integrates a query language with .Net programming languages (and recently with Java). One of the most important concepts of SBA is an *environment stack* (ENVS), known also as *call stack*. The approach respects the *naming-scoping-binding* paradigm, what means that each name in a query or program code is bound to a suitable run-time entity (e.g. object, attribute, variable, procedure, view etc.) depending on the scope for the name.

Analyzing query processing in the Stack-Based Approach it can be noticed that some subqueries are evaluated many times in loops implied by non-algebraic operators despite their results are the same in each loop iteration. This observation is the basis for an important query rewriting technique called the method of independent subqueries ([9]). It is also known from relational systems ([5], [6]) in a less general variant. In SBA this method is generalized for any kind of non-algebraic query operators and for very general object-oriented database model.

In [3] we present the generalization of the independent subqueries method to cases in which the subquery is dependent from its nearest non-algebraic operator, but the dependency is specifically constrained. The dependency concerns only an expression that can be entirely evaluated on a small collection. The rewriting rule in [3] is relevant regardless of whether the values of a small collection are available or unavailable during the compilation time. The values can be also changed after the compilation. If the dependency concerns a name that is typed by enumeration then a simpler rewriting rule may be used (see [2]). This rule is based on a conditional statement using all enumerators that have to be known during the compilation time.

In general, the dependency of a subquery from its nearest non-algebraic operator can concern two or more expressions dependent on small collections. The number of evaluations of such a subquery called “compound weakly dependent subquery” can be limited to the product of the sizes of these collections. The sizes of these collections should be significantly smaller in comparison to the collection size returned by the left subquery of the non-algebraic operator on which the subquery depends on. Comparing the sizes of collections enables the optimizer to check whether rewriting the query would guarantee better evaluation time. It implies introducing an efficient query evaluations cost model.

The rest of the paper is organized as follows. Section 2 describes the general idea of the compound weakly dependent subqueries method. Section 3 presents the corresponding algorithm that we have developed for the system ODRA [1]. Section 4 concludes.

2 The Optimization Method

To present SBA examples, we assume the class diagram (schema) presented in Fig.1. The classes *Student*, *Project*, *Emp* and *Dept* model projects implemented by students and supervised by employees working in departments. Names of classes (attributes, links, etc.) are followed by cardinality numbers (cardinality [1..1] is dropped).

Attributes *sex* of *Person* and *job* of *Emp* are of enumerated types. The first one takes values ("male", "female"), the second one takes values ("analyst", "programmer", "tester").

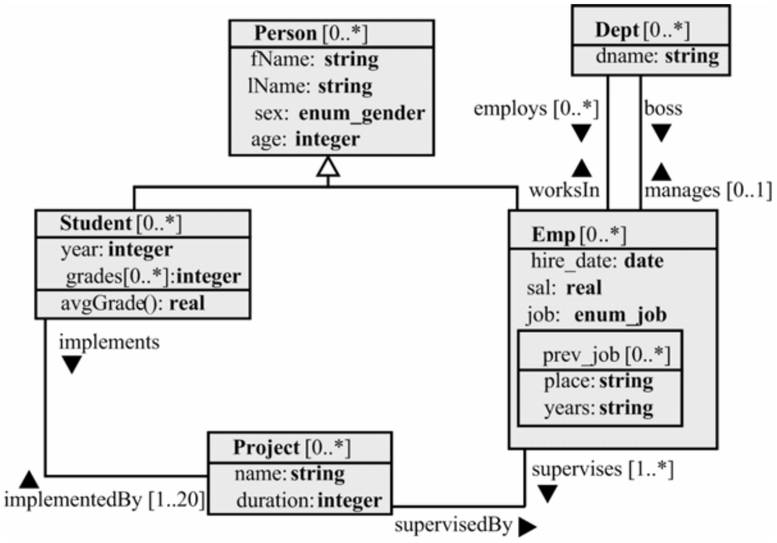

Fig. 1. A schema of an example database

2.1 Static Analysis of Queries

In this subsection we briefly present the mechanism of *static analysis* [10] used in our optimization method. It is a compile-time mechanism that performs static type checking ([4], [7]) on the basis of abstract syntax trees (ASTs) of queries. The static analysis uses special data structures: a metabase, a static environment stack S_ENVS and a static query result stack S_QRES. These structures are compile-time equivalents of run-time structures: an object store, an environment stack ENVS and a query result stack QRES, correspondingly. S_ENVS models bindings (in particular opening new scopes and binding names) that are performed on ENVS. The process of accumulating intermediate and final query result on QRES is modeled by S_QRES.

The main component of the metabase is a *schema graph* that is generated from a database schema. It contains nodes representing database entities (objects, attributes, classes, links, etc.) and interfaces of methods. The edges represent relationships between nodes. The graph nodes are identified by internal identifiers that are processed on static stacks. In our model the metabase stores also some statistical data. A node of the schema graph is associated with the estimated number of objects in the collection that is represented by the node. We designate it by *NE(Entity)*, where *Entity* is a unique node identifier. For clarity, *Entity* will be represented by an object name instead of a node identifier (in general, however, this assumption is not adequate, as node names need not be unique). For instance, *NE(Emp)* = 1000, *NE(Dept)* = 10, *NE*(*Student*)=2000, *NE*(*Project*)=100, *NE(Person)* = 3000.

2.2 General Idea of the Optimization Method Involving Compound Weakly Dependent Subqueries

The following example in SBQL illustrates the general idea of our method. The query gets employees having salary greater than the average salary calculated for employees working in their departments and having the same job. For the query below we determine binding levels for all the names and the numbers of stack sections opened by non-algebraic operators.

```
Emp as e where e . sal >
 1           2   2 3  3
             avg(e . worksIn . Dept . employs . (Emp where job=e . job) . sal)     (1)
                 2 3    3       3    3 3    3       3     4   4 2 5  5   3  3
```

Consider the following subquery of query (1)

avg(*e.worksIn.Dept.employs.*(*Emp* **where** *job=e.job*)*.sal*) (2)

The subquery (2) contains two names *e* that are bound in the 2nd stack section opened by the first *where* operator : in expression *e.worksIn.Dept* and in expression *e.job*. Hence the method of independent subqueries [9] cannot be applied. The method of queries involving large and small collections [3] cannot be applied too because it resolves dependency of subqueries concerning only one small collection. However the subquery (2) can be evaluated only 30 times instead of 1000 times. Therefore we have to develop a more general rewriting rule. After optimizing (1) it should take the following form:

(((*Dept* **as** *n1* **join bag**("analyst", "programmer", "tester") **as** *n2*)
join avg(*n1.employs.*(*Emp* **where** *job=n2*)*.sal*) **as** *n3*) **group as** *aux*). (3)
(*Emp* **as** *e* **where** *e.sal* > (*aux* **where** *e.worksIn.Dept=n1* **and** *e.job=n2*)*.n3*)

Names *n1*, *n2*, *n3* and *aux* are automatically chosen by the optimizer. In the two first lines of (3) before the last *dot* the query returns a bag named *aux* consisting of 30 structures. Each structure has three fields: an identifier (of a *Dept* object) named *n1*, a name of the job named *n2* and the average salary (named *n3*) calculated for employees having such job and working in this *Dept*. The last *dot* in the second line puts on top of ENVS a binder *aux* containing these structures. It is then used to calculate the query in the 3rd line. In this way average salaries are calculated three times for each department and they are used in the final query, as required.

Detecting subqueries such as (2) consists in verifying in which ENVS sections the names occurring in a subquery are to be bound. The binding levels for names are compared to the scope numbers of non-algebraic operators. We take into consideration only subqueries (referred to as "compound weakly dependent subqueries") of a given query that depend from their direct non-algebraic operator only on expressions returning small collections. If the number being the product of sizes of these collections is quite a lot smaller than the collection size returned by the left subquery of this non-algebraic operator, then we decide to rewrite such a query. Comparing the sizes of collections is necessary to check whether rewriting the query would ensure better performance. The query (1) involves two subqueries connected by the *where* operator. The left subquery *Emp as e* returns 1000 elements (according to the statistical data) and the right subquery (2) depends on the *where* operator on two expressions returning 10 and 3 elements, correspondingly. Hence it makes sense to rewrite (1) to the form (3). In this way the number of evaluations of the query (2) has been reduced to *NE(Dept)*sizeof(enum_job)*=30.

An essential difficulty of the algorithm consists in detecting specific parts of a compound weakly dependent subquery like *e.worksIn.Dept* and *e.job* for (2). We have chosen these parts because they contain the name *e* that is bound in the scope opened by the *where* operator. Besides these parts depend on small collections only. Other names in a subquery like (2) cannot be bound in the scope of its left-side direct non-algebraic operator. When the expression is typed by an enumeration (as in case of *e.job*) we use an explicit bag containing values of the enumerated type.

To limit the number of evaluations of a subquery like (2) to the number of collection elements returned by the subquery (4)

Dept **as** *n1* **join** bag(“analyst”, “programmer”, “tester”) **as** *n2* (4)

we factor a subquery like (5)

(*Dept* **as** *n1* **join** bag(”analyst”,”programmer”, ”tester”) **as** *n2*) (5)
join avg(*n1.employs.*(*Emp* **where** *job=n2*)*.sal*) **as** *n3*

out of the first *where* operator in (1). Query (5) is named with the auxiliary name *aux*. Then this name, as well as previously introduced names, are used to rewrite the subquery (2) to the form:

(*aux* **where** *e.worksIn.Dept=n1* **and** *e.job=n2*)*.n3* (6)

The query (6) returns for each employee the average salary calculated for employees working in his/her department and having the same job.

The method is experimentally tested within the ODRA system [1]. In the case of the query (1), the gain for a collection of 1000 employee objects is 10 times faster execution and the gain for 10000 employee objects is 118 times faster execution.

2.3 More General Case

The presented optimization method makes it possible to rewrite arbitrary compound weakly dependent subqueries. After rewriting such a subquery can be factored out of any non-algebraic operator. Besides, in general a query can contain several weakly dependent subqueries. Recursive application of our method enables rewriting all weakly dependent subqueries. At first, subqueries which are under the scope of the most nested non-algebraic operators are rewritten. The following example illustrates the generality of our method.

```
∃ Student as s (count(s .implements.Project.supervisedBy.(Emp where age>
2   1             2 3      3      3    3      3         3  3      4  4
       avg(worksIn.Dept.employs.(Emp where sex=s .sex).age)))>0)          (7)
             4  5 5   5    5     5       6    6 2 7 7 5 5
```

The above query returns *true* if at least one student implemented a project that was supervised by an employee satisfying the following criterion: the age of this employee must be greater than the average age calculated for employees working in his/her department and having the same sex as the given student.

Note that entire right-hand subquery of the quantifier is weakly dependent from it. Both expressions (*s.implements.Project* and *s.sex*) contain the name *s* that is bound in the 2nd stack section opened by the quantifier and depend on small collections. However the right-hand subquery of the quantifier also contains the subquery:

avg(*worksIn.Dept.employs.*(*Emp* **where** *sex*=*s* .*sex*).*age*) (8)

that is weakly dependent from the first *where* operator. The name *worksIn* in (8) is bound in the scope opened by this operator and denotes a pointer link to an element of a small collection of *Dept* objects. After rewriting the nested weakly dependent query (8) the query (7) takes the following form:

∃ *Student* ***as*** *s* (**count**(*s.implements.Project.supervisedBy.*(((*Dept* **as** *n1* **join**
avg(*n1.employs.*(*Emp* **where** *sex*=*s.sex*).*age*) **as** *n2*) **group as** *aux1*). (9)
(*Emp* **where** *age*>(*aux1* **where** *worksIn.Dept*=*n1*).*n2*)))>0)

Now the query (9) contains only one weakly dependent subquery: it is the entire right subquery of the quantifier. Applying the transformation to (9) we obtain the following query:

(((bag("male","female") **as** *n3* **join** *Project* **as** *n4*) **join**
(**count**(*n4.supervisedBy.*(((*Dept* **as** *n1* **join** **avg**(*n1.employs.*(*Emp* **where**
sex=*n3*).*age*) **as** *n2*) **group as** *aux1*).(*Emp* **where** *age*> (10)
(*aux1* **where** *worksIn.Dept*=*n1*).*n2*)))>0) **as** *n5*) **group as** *aux2*).(**forsome**
(*Student* **as** *s*) ((*aux2* **where** *s.implements.Project*=*n4* **and** *s.sex*=*n3*).*n5*))

The form (10) terminates the optimization action – no further optimization by means of our method is possible.

2.4 General Rewriting Rule

The general rewriting rule for queries involving compound weakly dependent subqueries can be formulated as follows. Let $q_1\ \theta\ q_2$ be a query connecting two subqueries by a non-algebraic operator θ. Let q_2 has the form:
$q_2 = \alpha_1 \circ cwds(\beta_1(C_1), \beta_2(C_2),\ldots, \beta_k(C_k)) \circ \alpha_2$; $k\geq 1$, α_1 and α_2 are some parts of q_2 (maybe empty), $\circ$ is a concatenation of strings, $cwds(\beta_1(C_1), \beta_2(C_2),\ldots, \beta_k(C_k))$ is a compound weakly dependent subquery where each part β_i (i=1..k) depends on θ only and contains a name C_i that is bound to an element of a small collection. Each $\beta_i(C_i)$ must be of the same type as the type of the collection of C_i elements. Then the query:

$$q_1\ \theta\ \alpha_1 \circ cwds(\beta_1(C_1), \beta_2(C_2),\ldots, \beta_k(C_k)) \circ \alpha_2 \qquad (11)$$

can be rewritten to:

(((C_1 **as** sc_1 **join** C_2 **as** sc_2 **join**…**join** C_k **as** sc_k)
join $cwds(sc_1, sc_2,\ldots, sc_k)$ **as** *aux1*) **group as** *aux2*). (12)
($q_1\ \theta\ \alpha_1 \circ$ ((*aux2* **where** $sc_1 = \beta_1(C_1)$ **and** $sc_2 = \beta_2(C_2)$ **and**…
and $sc_k = \beta_k(C_k)$).*aux1*) $\circ\ \alpha_2$)

The general idea consists in limiting the number of processings of a compound weakly dependent subquery *cwds*($\beta_1(C_1)$, $\beta_2(C_2)$,…, $\beta_k(C_k)$) to the number that is the product of the sizes of collections C_i (i=1..k) occurring in β_i. It is aimed by introducing an additional query with the *join* operator that is independent of θ. The entire result of this query is named *aux2*. It is a bag of structures *struct*{($sc_1(c_1)$, $sc_2(c_2)$,..., $sc_k(c_k)$), *aux2*(*cw*)}, where c_i (i=1..k) is an element of a bag returned by the name C_i and *cw* is an element returned by *cwds*(sc_1, sc_2,…, sc_k). The bag is then used (after the *dot*) to construct the query (*aux2 where* $sc_1 = \beta_1(C_1)$ *and* $sc_2 = \beta_2(C_2)$ *and…and* $sc_k = \beta_k(C_k)$)*.aux1*. It replaces the compound weakly dependent *cwds*($\beta_1(C_1)$, $\beta_2(C_2)$,…, $\beta_k(C_k)$) of (11). If some expression $\beta_i(C_i)$ is typed by an enumeration then instead of the collection name C_i the bag that consists of all the values of an enumerated type is used in (12).

3 The Rewriting Algorithm

The rewriting based on the rules (11) and (12) is accomplished by five recursive procedures. For the paper space limit we present only their signatures:

- *optimizeQuery*(*q:ASTtype*) – it applies the *queriesInvolvingCWDSMethod* procedure to AST node *q* as long as *q* contains subqueries depending on their non-algebraic operators only on expressions returning small collections.
- *queriesInvolvingCWDSMethod*(*q:ASTtype*) – it recursively traverses AST starting from node *q* and applies the *applyQueriesInvolvingCWDSMethod* procedure. If the procedure meets a non-algebraic operator then its right and left queries are visited by the same procedure. At first compound weakly dependent subqueries which are under the scope of the most nested non-algebraic operators will be rewritten.
- *applyQueriesInvolvingCWDSMethod*(θ*:ASTtype*) – it transforms according to rewriting rule (12) all right-hand subqueries of non-algebraic operator θ that depend on it only on expressions returning collections for which the number being the product of their sizes is small than the collection size returned by the left-hand subquery of the θ operator.
- *findCWDS*(θ*:ASTtype,q:ASTtype*): (*ASTtype*, *ASTtype*) – it applies the *getCWDS* function as long as the function returns a subquery of *q* (maybe the whole *q*) that is compound weakly dependent from θ.
- *getCWDS*(θ*:ASTtype,q:ASTtype*): (*ASTtype, ASTtype*) – it detects parts of query *q* that are dependent from θ operator on single names. Other names in the query cannot be in the scope of θ. If the dependency concerns expressions returning small collections or typed by enumerations then the function returns the query *q* and its dependent parts.

4 Conclusions and Future Work

We have presented a query optimization method which was aimed at minimizing the number of evaluations of compound weakly dependent subqueries. Our rewriting rule is very general, it works for any non-algebraic operator and for any data model (assuming that its semantics would be expressed in terms of SBA). The rule makes also no assumptions concerning what the compound weakly dependent subquery returns: it may return a reference to an object, a single value, a structure, a collection of references, a collection of values, a collection of structures, etc. Finally the rule makes rewrites for arbitrarily complex nested subqueries regardless of their left and right contexts.

The algorithm applied repeatedly detects and resolves all the possible compound weakly dependent subqueries in a query. Besides some subquery can be dependent from several non-algebraic operators, hence in each iteration of the algorithm the subquery is transformed and factored out of a next one. The prototype rewriting algorithm has been implemented by us in the ODRA system. We have to perform many experimental tests to confirm the efficiency of this algorithm. We also plan to implement another optimization variant that does not depend on a cost model.

References

1. Adamus, R., et al.: Overview of the Project ODRA. In: Proc. 1st ICOODB Conf., pp. 179–197 (2008) ISBN 078-7399-412-9
2. Bleja, M., Kowalski, T., Adamus, R., Subieta, K.: Optimization of Object-Oriented Queries Involving Weakly Dependent Subqueries. In: Proc. 2nd ICOODB Conf., Zurich, Switzerland, pp. 77–94 ISBN 978-3-909386-95-6
3. Bleja, M., Stencel, K., Subieta, K.: Optimization of Object-Oriented Queries Addressing Large and Small Collections. In: Proc. of the International Multiconference on Computer Science and Information Technology, Mrągowo, Poland, pp. 643–650 ISBN 978-83-60810-22-4, ISSN 1896-7094
4. Hryniów, R., et al.: Types and Type Checking in Stack-Based Query Languages. Institute of Computer Science PAS Report 984, Warszawa (March 2005), `http://www.si.pjwstk.edu.pl/publications/en/publications-2005.html` ISSN 0138-0648
5. Ioannidis, Y.E.: Y.E., Query Optimization. Computing Surveys 28(1), 121–123 (1996)
6. Jarke, M., Koch, J.: Query Optimization in Database Systems. ACM Computing Surveys 16(2), 111–152 (1984)
7. Lentner, M., Stencel, K., Subieta, K.: Semi-strong Static Type Checking of Object-Oriented Query Languages. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Štuller, J. (eds.) SOFSEM 2006. LNCS, vol. 3831, pp. 399–408. Springer, Heidelberg (2006)
8. Official Microsoft LINQ Project, `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx`
9. Płodzień, J., Kraken, A.: Object Query Optimization through Detecting Independent Subqueries. Information Systems 25(8), 467–490 (2000)
10. Płodzień, J., Subieta, K.: Static Analysis of Queries as a Tool for Static Optimization. In: Proc. IDEAS Conf., pp. 117–122. IEEE Computer Society, Los Alamitos (2001)
11. Subieta, K., Beeri, C., Matthes, F., Schmidt, J.W.: A Stack Based Approach to Query Languages. In: Proc. of 2nd Intl. East-West Database Workshop, Springer Workshop in Computing, Klagenfurt, Austria, September 1994, pp. 159–180 (1994)
12. Subieta, K.: Stack-Based Approach (SBA) and Stack-Based Query Language, SBQL (2008), `http://www.sbql.pl`
13. Subieta, K.: Stack-based Query Language. In: Encyclopedia of Database Systems 2009, pp. 2771–2772. Springer, US (2009) ISBN 978-0-387-35544-3,978-0-387-39940-9