published in: Proceedings of the 11th IEEE International Conference on Data Engineering, pp. 70–79, Taipei, Taiwan, March 6–10, 1995

Semantic Query Optimization for Methods in Object-Oriented Database Systems

Karl Aberer, Gisela Fischer

GMD–IPSI, Dolivostr. 15, 64293 Darmstadt, Germany
e–mail: {aberer, fischerg}@darmstadt.gmd.de

Abstract

Although the main difference between the relational and the object-oriented data model is the possibility to define object behavior, query optimization techniques in object-oriented database systems are mainly based on the structural part of objects. We claim that the optimization potential emerging from methods has been strongly underestimated so far. In this paper we concentrate on the question of how semantic knowledge about methods can be considered in query optimization. We rely on the algebraic and rule-based approach for query optimization and present a framework that allows to integrate schema-specific knowledge by tailoring the query optimizer according to the particular application's needs. We sketch an implementation of our concepts within the OODBMS VODAK using the Volcano optimizer generator.

1 Introduction

Progress in the optimization of declarative object-oriented queries has been concentrated on structural aspects, like the manipulation of complex data values and the usage of path indices [5][6][24]. In algebraic query optimization the optimization process is based on equivalences for the built-in query algebra operators. In semantic query optimization it is recognized that also application-specific knowledge in form of constraints on attributes can be used. But in most approaches the behavioral part of object-oriented data models is neglected: methods are regarded just as a slight generalization of attributes. We claim that methods lead to a new quality for semantic query optimization in OODBMSs, due to the much richer semantics they provide as compared to attributes. For instance, regardless of their actual procedural implementation the semantics of methods may be equivalent to queries (in some object-oriented database systems the operators of the query language are even provided as methods, e.g. GemStone [19]). Furthermore, in object-oriented database systems with extensible data models, the operations of new data model primitives may become visible through methods. Complex and expensive external operations stemming from complex application domains, like document processing, scientific or multimedia applications [1] are also introduced through methods. These operations may even be more dominant in query evaluation than the database operations themselves. The query optimizer should know the semantics of these methods. Using this approach in query optimization is the natural complement to exploiting method semantics in Semantic Concurrency Control (SCC), a concept already applied successfully in object-oriented database systems [22].

In this paper we concentrate on the question of how knowledge of method semantics can lead to algebraic equivalences which can be considered in query optimization. We will not consider queries containing methods with side effects, since we regard the problem of declarative updates to be orthogonal to the problem of using semantic knowledge in query optimization. We assume that the queries we consider are safe with regard to updates (which not necessarily implies that updates are not allowed!). For a discussion of problems related to updates see [10][18].

More precisely, the contribution of this paper is threefold. First, we identify the opportunities that are given for using semantic knowledge about methods through equivalences in an algebraic framework. This leads to results of theoretical interest, namely a classification of the way methods appear in algebraic representations of queries, and how equivalences can be derived from specifications of the semantics of methods. Second, we propose a design that allows to realize the possibilities we have identified, using existing database systems, query languages and query optimization technology as far as possible. Third, we introduce a prototype that allows to test the applicability and benefit of the concepts introduced in the paper in real world applications. The prototype is built for the object-oriented database system VODAK with the query language VQL, and uses the Volcano optimizer generator. To our knowledge

this is the first realization of semantic query optimization based on schema-specific equivalences in an OODBMS.

The remainder of the paper is organized as follows: in Section 2 we give a general classification scheme for methods and describe the example that we will use throughout the paper. We introduce the query language VQL, discuss how methods can be used in queries, and show how queries containing methods can be transformed using semantic knowledge about the methods. The representation of methods in a query algebra is examined in Section 3. In Section 4 we introduce the query algebra which is the basis for the optimization, and illustrate the derivation of query algebra rules from schema-specific knowledge on methods. Section 5 discusses the perspectives for applying these techniques. In Section 6 and 7 we describe the prototypical implementation of our concepts within the OODBMS VODAK using the Volcano optimizer generator. We discuss related work in Section 8 and conclude the paper in Section 9.

2 The unexplored side of objects: methods

In the following we will take a closer look at methods as they appear in an object-oriented data model. We refer to the data model VML [16] and the query language VQL [2] of the object-oriented database system VODAK, which is developed and implemented as a prototype at GMD-IPSI. However, the observations reported are valid for many other models and query languages.

2.1 Methods in object-oriented data models

We give an example of a database schema dealing with documents to illustrate the different ways how methods can be used.

The schema contains the classes Document, Section and Paragraph. In VML classes are not only containers for their instances, but also first class objects themselves. For the class objects Document and Paragraph the methods select_by_index and retrieve_by_string resp. are defined. select_by_index returns all documents with title t using a user-defined index for the selection, while retrieve_by_string retrieves all paragraphs which contain the string s using an external IR function. The instances of Document possess the attributes (in VML called *properties*) author, title and sections, and the method paragraphs which returns the identifiers of all paragraphs that belong to the same document. For the instances of Section the properties number, title, document and paragraphs are defined. An instance of the class Paragraph possesses the properties number, section and content. Furthermore the methods document which computes the document the paragraph belongs to, contains_string which searches for the existence of the string s in the content of the paragraph, and sameDocument which checks if two paragraphs belong to the same document, are provided. Within the implementation of sameDocument SELF denotes the receiver object of the method. The signatures of the properties and methods are given using the built-in complex data types of VML (the primitive built-in data types are STRING, INT, REAL, ... and typed object identifiers; the type constructors are TUPLE, SET, ARRAY and DICTIONARY.). We can access the properties using default methods provided automatically by the system for reading and writing public properties.

```
CLASS Document
  OWNTYPE OBJECTTYPE
  METHODS:
    select_by_index(t: STRING): {Document}
      { /* external implementation */}
END;
  INSTTYPE OBJECTTYPE
  PROPERTIES:
    title: STRING;
  METHODS:
    paragraphs(): {Paragraph};
      { /* compute all paragraphs of a document */}
  END;
END;

CLASS Section
  INSTTYPE OBJECTTYPE
  PROPERTIES
    document: Document;
    paragraphs: {Paragraph};
  END;
END;

CLASS Paragraph
  OWNTYPE OBJECTTYPE
  METHODS
    retrieve_by_string(s: STRING): {Paragraph}
      { /* external implementation */}
  END;
  INSTTYPE OBJECTTYPE
  PROPERTIES
    number: INT;
    section: Section;
  METHODS
    document(): Document;{ RETURN section.document;}
    contains_string(s: STRING): BOOL;
      { /* external implementation */}
    sameDocument(p: Paragraph): BOOL;
      { RETURN (SELF→document() == p→document());}
  END;
END;
```

From this example we can identify two orthogonal dimensions in which methods can be categorized. According to their *generality* we can distinguish *application-specific methods*, which are defined in application schemas, from *system-defined methods*, like default methods for property access or object creation and deletion. According to their *implementation* we distinguish methods with *internal encoding*, e.g. document, and methods with an *external im-*

plementation, e.g. contains_string and select_by_index. Additionally one has to consider that methods can be parametrized, e.g. sameDocument.

2.2 Methods in a query language: the VODAK query language VQL

VQL supports declarative access to object-oriented databases in the OODBMS VODAK. For pragmatic reasons VQL is based on a SQL-like approach. Compared to standard SQL, VQL provides features which are either convenient or necessary in order to efficiently access VODAK databases, in particular the use of method calls and path expressions.

Since we allow arbitrary method invocation in queries, we cannot determine in advance whether a query is a pure retrieval query or whether updates are performed due to the execution of the methods contained in the query [10]. In order to reflect this semantics in the syntax we replaced the SQL-keywords SELECT/UPDATE by the more general keyword ACCESS. A VQL query has the form

ACCESS expr(x_1,...,x_n)
FROM x_1 **IN** S_1, ..., x_n **IN** S_n
WHERE cond(x_1,...,x_n)

where x_i are query or range variables, and S_i are sets of object identifiers (usually the extensions of classes), expr(x_1,...,x_n) is an expression built up from query variables, constants, path expressions, method calls and operations on the primitive and complex data types (here and later on, we omit an exhaustive list due to space limitations). cond(x_1,...,x_n) consists of expressions, primitive predicates on built-in datatypes and boolean operators.

Methods can be used in any part of the query as shown by the following examples:

Example 1: Methods in the **WHERE** clause appear in atomic predicates. Note that, due to parametrization, method calls may serve as join predicates, e.g. f→sameDocument(q) is the join predicate in the following query which returns a set of tuples as result

ACCESS [p: p. number, q: q.number]
FROM p **IN** Paragraph, q **IN** Paragraph
WHERE p→sameDocument(q)

Example 2: Methods may appear in the **FROM** clause if they return sets of object identifiers. Using methods can lead to dependencies between the query variables. E.g. in the following query the query variable p is dependent on the variable d.

ACCESS d.title
FROM d **IN** Document, p **IN** d→paragraphs()
WHERE p→contains_string('Implementation')

Example 3: Methods in the **ACCESS** clause are used to bring the selected elements into the desired form for output or further processing, or to perform updates.

ACCESS [doc: d.title, paras: d→paragraphs()]
FROM d **IN** Document

2.3 Transformation of queries by using knowledge about methods

Algebraic query optimization basically relies on two types of knowledge, namely *equivalences* and *cost functions* for the algebra operators. In this paper we will focus on the examination of equivalences. However, we note that attributes are assumed to be obtained at uniform access cost. This is not true for methods, especially with respect to method parameter, see e.g. [14].

In the following we give an example of how equivalences derived from methods semantics can be used for query transformation, and hence optimization. For illustration purposes, in this section we do this on the query language level instead of the query algebra level relying on the reader's intuitive understanding of how query language expressions are evaluated. This gives a first flavor of how *schema-specific query algebra equivalences* can be used for algebraic query transformation.

Example 4: By inspecting the schema introduced in section 2 we can identify the following equivalences between expressions. For boolean expressions we use ⇔ to denote the equivalence, for other expressions we use ≡.

E_1: p→document() ≡ p.section.document
E_2: d.title == s
⇔ d **IS-IN** Document→select_by_index(s)
E_3: p.section.document **IS-IN** D
⇔ p.section **IS-IN** D.sections
E_4: p.section **IS-IN** S ⇔ p **IS-IN** S.paragraphs
E_5: **ACCESS** p **FROM** p **IN** Paragraph
WHERE p→contains_string(s)
≡Paragraph→retrieve_by_string(s)

p and q are variables for instances of the class Paragraph, d represents an instance of Document, D and S are sets of documents and sections resp., and s is a string. By D.sections and S.paragraphs we denote the union of all sections of the documents in D and the union of all paragraphs of the sections in S resp. (i.e. the system-defined methods which perform the access to the property are invoked for all objects in the set). Note that the equivalences have no distinguished direction, thus in principle they can be applied in both directions. The direction in which an equivalence is actually applied for a certain query will be chosen by the optimizer.

We now illustrate some query transformations using the above equivalences. Assume the following query is posed by a user. It retrieves all paragraphs which belong to a docu-

ment titled 'Query Optimization' and contain the string 'Implementation':

```
Q:  ACCESS p FROM p IN Paragraph
    WHERE p→contains_string('Implementation')
      AND (p→document()).title == 'Query Optimization'
```

This query can be transformed using equivalence E_2 to

```
Q': ACCESS p FROM p IN Paragraph
    WHERE p→contains_string('Implementation')
      AND p→document() IS-IN
        Document→select_by_index('Query Optimization')
```

Now we can apply equivalence E_1, E_3 and E_4 consecutively:

```
Q'': ACCESS p FROM p IN Paragraph
     WHERE p→contains_string('Implementation')
       AND p.section.document IS-IN
       Document→select_by_index( 'Query Optimization')
```

```
Q''': ACCESS p FROM p IN Paragraph
      WHERE p→contains_string('Implementation')
        AND p.section IS-IN (Document→select_by_index(
          'Query Optimization')).sections
```

```
Q'''': ACCESS p FROM p IN Paragraph
       WHERE p→contains_string('Implementation')
         AND p IS-IN (Document→select_by_index(
           'Query Optimization')).sections.paragraphs
```

Applying equivalence E_5 and some standard query transformations, e.g.

```
ACCESS p FROM p IN Paragraph
  WHERE p IS-IN (Document→select_by_index(
    'Query Optimization')).sections.paragraphs
 ≡ (Document→select_by_index(
     'Query Optimization')).sections.paragraphs)
```

yields the query plan P_Q:

```
PQ: Paragraph→retrieve_by_string('Implementation')
      INTERSECTION
    (Document→select_by_index
      ( 'Query Optimization')).sections.paragraphs
```

Several observations have to be reported for this example:

- The example schema is reasonably designed. The query was posed in a completely natural way from the user's perspective and there are also no natural alternatives for the query formulation which come closer to the final query plan (e.g. starting the query from the class Document always requires a join operation).
- The final query plan can, for a given typical database, be evaluated much more efficient than a straightforward evaluation of the query without transformation.
- There is no way for the optimizer to derive the final query plan from the user's query without having schema-specific information on the semantics of the methods.
- The schema-specific knowledge that has to be provided by the schema designer can be easily specified and is unambiguous: no deep analysis of the code is necessary, just a descriptive way to reflect the intended semantics of the methods in the schema is needed.

In this example different kinds of knowledge about methods semantics were used in order to optimize the query. Among these are knowledge about the implementation of path methods, properties of inverse links and method implementations that are semantically equivalent to query expressions. In the following we will systematically discuss the different types of knowledge and how they come into optimization.

3 Methods in a query algebra

Query optimization in VODAK is based on an algebraic representation of VQL queries. If we want to study the optimization of queries containing method calls, we first have to investigate how methods are represented in the query algebra. In principle, methods can occur there in two different ways: either as *algebraic operators* or as *parameters of algebraic operators*. As we will see, both possibilities are relevant, depending on the semantics of the methods.

3.1 Methods as parameters of query algebra operators

The most original extension of relational, nested relational or complex object algebras to object-oriented query algebras is the introduction of operators that allow iterative application of functions to the elements in a value of a *bulk type* (e.g. set, list, etc.; in this paper we will restrict ourselves to sets). The standard example of such an operator is

$$\mu<\lambda x.\ f(x)>(S) ::= \{f(s) \mid s \in S\}$$

where S is a set of type $\{T_1\}$, x is a variable of type T_1, and $\lambda x.\ f$ is a function of type $T_1 \rightarrow T_2$. The result of applying this operator is a set of type $\{T_2\}$. A more general form of this operator is

$$\mu<\lambda x_1...x_n.\ f(x_1,...,x_n)>(S_1,...,S_n) := \{f(s_1,...,s_n) \mid s_i \in S_i\}$$

In this context lambda abstraction is used for convenience. It is restricted to first-order abstractions, i.e. the type of the bound variable x is a primitive domain. The function f is specified as a composition of operations on built-in data types and method calls. Thus methods come into the query algebra as operator parameters.

Example 5: The method paragraphs() is applied to each document in the class Document as follows

$$\mu< \lambda x.\ x \rightarrow paragraphs()> (Document).$$

μ can also be used to express many other bulk operators. For instance, by lifting tuple constructors and selectors in this way, the standard relational operators of Cartesian product and projection can be expressed. In fact it was shown that together with *set collapse* which is defined as $\gamma(S) := \cup s \in S$, where S is of type $\{\{T\}\}$, this operator can serve as the basis for a complete complex object algebra [6].

Example 6: The usual projection operator π_a on a relation R can be expressed as follows

$\pi_a(R) := \mu< \lambda x.\ x.a>(R)$

3.2 Methods as algebraic operators

Usually operators of object-oriented query algebras manipulate bulk types whose components may have arbitrary domains. Such operators might also be realized through methods, since the parameter types of methods can be bulk types. We give (a very simple) example of how such operators might be introduced in VODAK.

Example 7:

```
CLASS Set_object
INSTTYPE OBJECTTYPE
  PROPERTIES
    elements: { OID }
  METHODS
    select( m1: VML_Method ) : Set_object;
      /* m1: OID→Boolean */
    map( m2: VML_Method ) : Set_object;
      /* m2: OID→OID */
  END;
END;
```

In this example we represent only two algebra operators on bulk types, namely **select** and **map**, as methods defined for the instances of a system-defined class Set_object. The instances of this class store values of a bulk type, namely sets of object identifiers of instances of arbitrary classes. When the method select is invoked for an instance i of Set_object, it first requests Set_object to create a new instance i'. Then select applies the method m_1 to all objects whose identifiers are stored in the property elements of i, stores the identifiers of those objects in the property elements of i' for which m_1 evaluates to TRUE, and returns the identifier of i'. Accordingly the method map applies m_2 to all objects in elements and returns the results in a new instance of Set_object.

If a unary algebra operator is represented as a method its argument corresponds to the receiver object of the method. For operators with more than one argument, the additional arguments appear in the parameter list of the method.

Methods like select and map may be used as physical implementations of query algebra expressions. Thus they can appear at the operator level in the query algebra.

4 Using semantic equivalences in rule-based query optimization

In this section we will show how to break down the general ideas presented so far to an approach that allows an implementation based on rule-based query optimization techniques.

4.1 Definition of the query algebra

For simplicity we restrict ourselves to the manipulation of bulk values of relation type, namely *set[tuple[domains]]*, where *domains* are arbitrary complex data types. This allows us to use mostly techniques well-known from relational query optimization for the manipulation of bulk values. Problems related to the manipulation of arbitrary complex values have already been extensively studied [23] and are orthogonal to the problem of exploiting the semantics of object definitions we examine here. We consider the extensions of our techniques to the general case as straightforward.

The query algebra operators are applied to complex values of type $\{[a_1: D_1, ..., a_n: D_n]\}$ where $D_1,...,D_n$ are complex data types. We assume that the tuple components are unordered. Operator arguments of this type are denoted by S, S_1 and S_2. The operator parameters are enclosed in <>. We define

$Ref(S) := \{ a_1,...,a_n \}$ for $Type(S)=\{[a_1: D_1, ..., a_n: D_n]\}$.

and refer to $a_1, ..., a_n$ as the *references* of S. In the following v_i denotes a value of type D_i. The algebra consists of the following operators:

get<a, class> := { [a: o] | o ∈ extension(class) }

natural_join(S_1, S_2) :=
{[a_1: v_1, ... ,a_i: v_i, ..., a_k: v_k, ..., a_n: v_n] |
∃ [a_1: v_1, ... ,a_i: v_i, ..., a_k: v_k] ∈ S_1
∧∃ [a_i: v_i, ..., a_k: v_k, ..., a_n: v_n]∈ S_2}
where $Ref(S_1) = \{a_1, ... , a_k\}$, $Ref(S_2) = \{a_i, .., a_n\}$ and $i \leq k$

union(S_1, S_2) := {[a_1: v_1, ... , a_n: v_n] |
∃ [a_1: v_1, ... , a_n: v_n] ∈ S_1 ∨ ∃ [a_1: v_1, ... , a_n: v_n]∈ S_2}
where $Ref(S_1) = Ref(S_2) = \{a_1, ... , a_n\}$

diff(S_1, S_2) := {[a_1: v_1, ..., a_n: v_n] |
[a_1: v_1, ..., a_n: v_n] ∈S_1 ∧[a_1: v_1, ..., a_n: v_n] ∉S_2}
where $Ref(S_1) = Ref(S_2) = \{a_1, ... , a_n\}$

select<condition(a_1,...,a_n)>(S) := {[a_1: v_1, ..., a_n: v_n] |
[a_1: v_1, ..., a_n: v_n] ∈S ∧ condition(v_1,...,v_n)==TRUE }
where $Ref(S) = \{ a_1,...,a_n \}$

join<condition(a_1,...,a_i, a_{i+1} , ...,a_n)>(S_1,S_2) :=
{[a_1: v_1, ... ,a_i: v_i, a_{i+1}: v_{i+1}.. a_n: v_n] |
[a_1: v_1, ... ,a_i: v_i]∈S_1 ∧[a_{i+1}: v_{i+1}.. a_n: v_n]∈ S_2
∧ condition(v_1,...,v_n)==TRUE }
where $Ref(S_1) = \{a_1, ... , a_i\}$ and $Ref(S_2) = \{a_{i+1}, .., a_n\}$

map<a, expression(a_1,...,a_n)>(S) :=
{ [a: v,a_1: v_1, ..., a_n: v_n] |
[a_1: v_1, ... ,a_n: v_n]∈S ∧ v=expression(v_1,...,v_n) }
where $Ref(S) = \{ a_1,...,a_n \}$ and a ∉ Ref(S)

flat<a, expression(a_1,...,a_n)>(S) := { [a: v,a_1: v_1, ..., a_n: v_n] |
[a_1: v_1, ... ,a_n: v_n]∈S ∧ v∈expression(v_1,...,v_n) }
where $Ref(S) = \{ a_1,...,a_n \}$, a ∉ Ref(S)
and expression is set-valued.

project<a_1,...,a_i>(S) :=
{ [a_1: v_1, ..., a_i: v_i] | ∃ v_{i+1}, ... , v_n [a_1: v_1, ..., a_n: v_n] ∈S }
where $Ref(S) = \{ a_1,...,a_n \}$

In this algebra arbitrarily complex expressions may appear in the operator parameters. **get, natural_join**, **union** and **diff** provide access to classes and standard relational join

operations (subsuming the set-theoretic operations). In analogy with relational algebra, **select** and (Θ-)**join** are defined. Thus methods can appear in selection and join conditions. **map**, **flat** and **project** allow to iteratively apply complex expressions with arbitrary return values to their arguments. It is important to notice that we derive these operators from the unary operator μ introduced in section 3.1, and thus do not produce an (implicit) Cartesian product (in other object-oriented algebras this is not always the case, see e.g. [25]; on the other hand, our **map** operator is comparable to the operator denoted by $X_{a:e}$ in [9]). The operators **flat** and **map** are to be considered as duals of each other with regard to set nesting, and **project** and **map** are to be considered as duals of each other with regard to tupling.
A VQL query of the form

ACCESS expression (x_1,...,x_n)
FROM x_1 **IN** C_1, ..., x_n **IN** C_n **WHERE** condition(x_1,...,x_n)

where C_1, ..., C_n are class names, is then mapped to the following algebra expression:

project<a>(
  map<a, expression(a_1,...,a_n)>(
    select<condition(a_1,...,a_n)>(
      join<true>(**get**<a_n,C_n>, (**join**<true>
        (**get**<a_{n-1},C_{n-1}> ...
          join<true>(**get**<a_1,C_1>,**get**<a_2,C_2>)...)))))))

4.2 Derivation of query algebra rules

If we want to utilize schema-specific knowledge in the optimization, we have to map this knowledge about methods to rules which can be applied to algebraic query expressions. According to [13] we distinguish two types of rules, namely *transformation rules* which transform a query algebra expression into another one, and *implementation rules* which map a query algebra expression to an executable query evaluation plan. We can identify different types of rules that can be derived from different specifications of method semantics.

Equivalent expressions. An expression equivalence is given by

x IN C: $expr_1$(x) == $expr_2$(x);

In principle, we can translate this kind of equivalence into a transformation rule of the query algebra by *lifting* the equivalence to bulk type operators as follows

map<?a_1, $expr_1$(?a_2)>(?A<?a_2, C>)
  ↔ **map**<?a_1,$expr_2$(?a_2)>(?A<?a_2, C>)

In this transformation rule ?a_1 and ?a_2 represent patterns for references in the operator arguments and ?A<?a_2, C> represents a pattern for an algebraic expression that returns object identifiers of instances of class C with reference ?a_2. Examples of such expressions are **get**<?a_2, C>, but also **map**<?a_2, $expr_1$(?a_3)>(**get**<?a_3, C'>) where $expr_1$ evaluates to object identifiers of instances of class C.

A typical example of obtaining this type of knowledge is through path methods:

E_1: p→document() ≡ p.section.document

is mapped to

map<?a_2,?a_1→document()> (?A<?a_1, Paragraph>))
  ↔ **map**<?a_2, ?a_1.section.document>(
      ?A<?a_1, Paragraph>)))

Equivalent conditions. This type is very similar to the previous one, except that here we consider only the equivalence of boolean expressions:

x IN C: $cond_1$(x) ⇔ $cond_2$(x);

We can translate this kind of equivalence also into a transformation rule of the query algebra by lifting it as follows

select<$cond_1$(?a_1)>(?A<?a_1, C>)
  ↔ **select**<$cond_2$(?a_1)>(?A<?a_1, C>)

A typical source for this type of equivalences are inverse links:

E_3: p.section.document **IS-IN**
    ⇔ p.section **IS-IN** D.sections

is mapped to

select<?a_1.section.document **IS-IN**
  ?A>(?A_1<?a_1, Paragraph>)
  ↔ **select**<?a_1.section **IS-IN**
  ?A.sections>(?A_1<?a_1, Paragraph>)

?A represents an expression returning a set of document object identifiers.

Implication of conditions. In this case we use a logical implication instead of an equivalence:

x IN C: $cond_1$(x) ⇒ $cond_2$(x);

The implication yields the following algebraic equivalence

select<$cond_1$(?a_1)>(?A<?a_1, C>)
  ↔! **natural_join**(**select**<$cond_1$(?a_1)>(
  ?A<?a_1, C>),**select**<$cond_2$(?a_1)>(?A<?a_1, C>))

In this case the **natural_join** operator behaves like an intersection as the set of references are the same for both operator arguments. The symbol↔! means that the rule may only be applied once, in order to avoid an infinite recursive application of the rule.

Although this type of equivalence appears to be of little use at the first glance, it can be very interesting for finding efficient execution plans in the presence of *precomputed information* as the following example illustrates. Assume the class Document provides an instance property largeParagraphs: {Paragraph}, and the class Paragraph provides an instance method wordCount(): INT, such that the access methods of Document and Paragraph guarantee the following property:

p IN Paragraph: p→wordCount() > 500
  ⇒ p IS-IN p→document().largeParagraphs

This implication can be transformed into an equivalence as indicated above, similarly as in case of equivalent conditions.

Equivalences Between Queries and Method Calls. This case is somewhat different from the previous ones as two language levels are combined:

methcall == query,

where query = **ACCESS** ... **FROM** ... **WHERE** ... . The method call methcall corresponds to an implementation of a query algebra expression which can be directly executed by the database system. Such an equivalence can be translated to an implementation rule. The query query in the equivalence is first translated into its algebraic representation A_{query}, and thus the implementation rule which is applicable only in one direction is:

A_{query} → methcall

As a concrete example of how this rule is derived we show how equivalence E_5 is mapped to an implementation rule:

E_5: **ACCESS** p **FROM** p **IN** Paragraph
WHERE p→contains_string(s)
≡ Paragraph→retrieve_by_string(s)

is mapped to

select<*?a*$_1$→contains_string(s)>(*?A*<*?a1*, Paragraph>)
→ Paragraph→retrieve_by_string(s)

Some natural extensions can be made for all of the four types of knowledge considered. For example, one can bind several variables in the equivalent expressions and conditions, or one can impose additional conditions on parameters appearing in these expressions.

5 Perspectives for applicability

So far we have shown how knowledge about methods can be used to derive rules for query optimization purposes. Before we sketch an implementation of our approach in the next section, we discuss its applicability, i.e. in which cases equivalences for methods can be defined and beneficially used in the optimization.

5.1 The origin of semantic knowledge

It is worthwhile to consider why a lot of semantic knowledge is available for optimization purposes. This gives an immediate insight in which situations our approach can pay off.

- In object-oriented schemas redundant structures (e.g. inverse links) can be found more often than in relational schemas, since the whole theory of normalization of relational database schemas is actually devoted to the issue of avoiding redundancy. In most cases redundant structures are provided in order to gain simple and efficient access to related data. Note that in contrast to relational systems redundant data in object-oriented databases can be easily kept consistent by encapsulating the consistency check into corresponding methods.
- The return values of methods constitute derived data, i.e. data which has been computed using data stored in the database, and therefore relationships between these return values and the database state exist (e.g. path methods or a method which computes the actual age of a person using the person's date of birth stored in the database).
- Methods may incorporate queries, i.e. provide the same semantics as a particular (possibly parametrized) query. Providing a method instead of a query can have two significant advantages: first, the user is relieved of formulating potentially difficult queries. Second, a method may provide a more efficient implementation of a particular query than the DBMS can.
- Object-oriented database systems are especially designed to support complex application domains, like document processing, scientific applications or multimedia applications. In such application domains complex and expensive external operations, which may even be more dominant in the query evaluation as the database operations themselves, can often occur.

5.2 Specification of the semantic knowledge

We have shown that different kinds of specifications of method semantics can be given for query optimization purposes. In this paper we do not address the design of an appropriate user interface for acquiring this knowledge. However, it is clear that such a feature can be provided at least for the knowledgeable user of the system, who designs complex and time-critical applications, but has no access to database internals. It is not necessary that this knowledge has to be provided in all cases in the explicit form as discussed in Section 4.2, but it may be derived from other information, like such about inverse links. Furthermore in some special cases it may be possible to derive this knowledge automatically, e.g. for methods generated by a path method generator [21]. Another scenario we envisage is the introduction of new data modeling primitives employing the extensibility of object-oriented data models. This approach is taken in VML where for example semantic relationships can be introduced through metaclass schemas [15][16][17]. These schemas are developed on the same core model as application schemas use, but can be hidden from the casual user of the new data modeling primitives who is not aware of the the metalevel of the schema. A detailed discussion of this approach will be the issue of a forthcoming paper.

6 Implementation

We can distinguish two techniques used for rule-based query optimization: one is based on general purpose tools for rule application, e.g. logical programming languages

[8], the other uses tools particularly suited for the needs of algebraic query optimization, e.g. EXODUS [11] and Volcano [13]. We follow the second approach and utilize the *Volcano optimizer generator* since it has been shown to be very efficient. This leads to the following questions when using extensible query optimizer generators:

- What are the requirements the query algebra should meet for rule-based optimization?
- How can we exploit schema-specific equivalences involving methods for rule-based optimization?

After introducing the Volcano optimizer generator we discuss these questions in detail.

6.1 The Volcano optimizer generator

The Volcano optimizer generator was developed as part of the Volcano project [13]. It generates efficiently working rule- and cost-based algebraic query optimizer modules using a *logical* and a *physical query algebra* and corresponding *transformation* and *implementation rules*. The logical algebra is the basis for the internal representation of the query, while the physical algebra is used to build up query evaluation plans. Its operators are concrete algorithms associated with cost functions. Transformation rules may be applied in both directions; they reorder operators in a logical algebra expression. Implementation rules map logical algebra expressions to algorithms and thus are only applicable in one direction. Each rule may be associated with a condition; then it is only applied if the condition holds. The optimization itself now consists of mapping a logical algebra expression step by step to the optimal (with respect to the execution costs) equivalent physical expression according to the given rules. In order to find the optimal physical expression each generated optimizer contains a fixed search algorithm based on exhaustive search for all logical transformations and branch-and-bound pruning when applying implementation rules.

The rule matching algorithm incorporated in Volcano can utilize *operator patterns* consisting of operator, operator argument and input variables. The content of operator arguments can only be checked in the condition code, thus no pattern matching on the arguments is supported. This restriction is a consequence of enforcing a strong typing on the rules themselves which, together with rule compilation, is necessary in order to achieve efficient rule matching. As we will see in the following, this has an important influence on the structure of the logical algebra which we use as the basis of the optimization.

The Volcano optimizer generator provides complete data model independence: the logical and physical algebra, the corresponding transformation and implementation rules as well as the (arbitrary complex) cost functions are freely definable by the optimizer implementor.

In the form given in Section 4.2 the rules are not well-suited for usage with the Volcano optimizer generator. Take for the example the rule

map<*?a*$_2$,*?a*$_1$→document()> (*?A*$_1$<*?a*$_1$, Paragraph>)) ↔
map<*?a*$_2$, *?a*$_1$.section.document>(
?A$_1$<*?a*$_1$, Paragraph>)))

The pattern matching algorithm of Volcano can detect the operator pattern, namely **map**<...>(*?A*$_1$<...>)), but matching the expression pattern of the parameter on the right side, namely *?a*$_1$.section.document, is left to the condition code, and hence to its implementor. This would basically require to implement another ”pattern matching algorithm” in the condition code of the rules which is not the way Volcano is intended to be used efficiently. As a consequence we have to simplify the operator arguments of the algebraic operators (a similar observation was reported in [4] for a pure structurally object-oriented algebra). More precisely, the specialized operators have parameters restricted to atomic expressions and conditions.

In the restricted algebra we substitute some of the operators of the general algebra. The substitution is given by the following table (the operators not mentioned in the table remain unchanged). Θ is one of the boolean binary operations on built-in data types (e.g. ==, !=, <, >, IS-IN, IS-SUBSET). Φ is an operation on the built-in data types (e.g. +, –, *, /, tuple access and construction, string operations). p and m denote property and method identifiers resp.

Restricted Algebra ($a_{new} \notin Ref(S_j)$, $a_i \in Ref(S_j)$, i=1, 2, 3,..., j=1,2)	General Algebra ($a_{new} \notin Ref(S_j)$, $a_i \in Ref(S_j)$, i=1, 2, 3,..., j=1,2)
select<a_1,Θ,a_2>(S)	**select**<$a_1 \Theta a_2$>(S)
join<a_1,Θ,a_2>(S_1,S_2)	**join**<$a_1 \Theta a_2$>(S_1,S_2)
map_property<a_{new}, p, a_1>(S)	**map**<a_{new}, a_1.p>(S)
map_method<a_{new}, m, a_1, <a_2, a_3,...>>(S)	**map**<a_{new}, $a_1 \rightarrow m(a_2, a_3, ...)$>(S)
flat_property<a_{new}, p, a_1>(S)	**flat**<a_{new}, a_1.p>(S)
flat_method<a_{new}, m, a_1, <a_2, a_3,...>>(S)	**flat**<a_{new}, $a_1 \rightarrow m(a_2, a_3, ...)$>(S)
map_operator <a_{new}, Φ, a1,...,an>(S)	**map**<a_{new}, $\Phi(a_1,...,a_n)$>(S)

Both algebras have the same expressive power. One can show this by translating expression composition which can take place on the parameter level in the general algebra to operator composition in the restricted algebra.

For query transformation based on the restricted algebra, a predefined set of transformation rules is provided. These are on the one hand many well-known rules from relational query optimization, e.g. associativity and commutativity of join or interchangeability of selection and join. On the other hand, there are rules that involve the new

operators, in particular **map_property, map_method, flat_property** and **flat_method**. Investigating these rules for optimization purposes is an interesting problem of its own right that we can not discuss in full detail in this paper. We give one example of the rules that were used in our implementation.

Example 8: Transformation of path expressions (which are implicit joins) to explicit joins. C is the name of the class to which the objects referred by the reference matched to $?a_2$ belong.

project<$?a_1$,$?a_2$,Ref($?A$)>(
  map_property <$?a_3$, $?p2$, $?a_2$>(
    map_property <$?a_2$, $?p_1$, $?a_1$>($?A$))) ↔
project< $?a_1$,$?a_2$,,Ref($?A$)>(
  join<$?a_4$, "==", $?a_2$> (
    map_property <$?a_3$, $?p_2$, $?a_4$> (**get** <$?a_4$, C>),
    map <$?a_2$, $?p1$, $?a_1$> ($?A$)))

6.2 Representation of rules in the restricted algebra

We show the representation of the rules in the restricted algebra for the rule derived from equivalence

E_3: p.section.document **IS-IN** D ⇔
p.section **IS-IN** D**.**sections

In the general algebra the corresponding rule was

select<$?a_1$.section.document **IS-IN** $?A$>(
  $?A_1$<$?a_1$, Paragraph>) ↔
select<$?a_1$.section **IS-IN** $?A$.sections>(
  $?A_1$<$?a_1$, Paragraph>)

This rule can be translated to the restricted algebra as follows:

natural_join (
  map<$?a_3$, document, $?a_2$>(
    map<$?a_2$, section, $?a_1$>($?A_1$<$?a_1$, Paragraph>))),
  $?A_1$<$?a_3$, Document>)) ↔
natural_join(
  map<$?a_2$, section, $?a_1$>($?A_1$<$?a_1$, Paragraph>),
  flat<$?a_2$, sections, $?a_3$>($?A_1$<$?a_3$, Document>))

A detailed description of transformation and implementation rules in Volcano and the translation of the rule shown above to a transformation rule is given in [3].

7 Prototypical implementation in VODAK

In the current VODAK version VQL queries can be posed interactively, within applications or within method implementations. Currently we have integrated query optimization facilities in the interactive mode. We have implemented the algebra as described in this paper, the set of fixed transformation and implementation rules in order to apply common algebraic transformations and enable the mapping from the logical to the physical algebra resp., and a simple cost model.

Since we want to take knowledge on schema-specific semantics into account for the optimization, we have to find a way to dynamically extend the predefined optimizer components, i.e. define new operators and rules. ”Dynamically” means here that the extension is automatically performed by the system according to the knowledge given by the schema designer, and not by the database system implementor. We integrate schema-specific semantics in the optimization process by mapping them to transformation and implementation rules, adding these rules and the methods which are defined as physical operators to the predefined rules and operators, and generating an individual optimizer module for each schema.

We have implemented a demonstrator that graphically illustrates how the VQL query optimizer works. This is achieved by tracing the single steps of the optimization process, i.e. by visualizing a query expression throughout the optimization process. This visualization of database internals can be used as a *debugging tool* for examining the impact of schema-specific equivalences on the optimization process.

8 Related work

The basis for object-oriented query algebras and optimization by exploiting structural properties of complex datatypes is by now well-known in the literature [4][5][6] [24]. Also the use of the Volcano optimizer generator for query optimization in a structural oriented query algebra was investigated. In [7] an object-oriented query algebra including a logical *materialization* operator for property paths was used. This operator can be considered as a predecessor of the operator **map**. The usage of Volcano for incorporating new data types and external methods in the context of scientific applications was proposed in [26]. However, there the authors leave the task of rule specification to the database implementor. It is curious to note, that most work known to us where account is taken of user-defined functions in query optimization, can be found in the context of relational and extensible relational database systems. In [23] the optimizer can use knowledge about user-defined abstract data types. In [8] knowledge about external functions is used in query optimization. There the authors use a logical programming approach to rule matching.

In this paper we have excluded the issue of nested queries, which is of considerable interest for semantic query optimization based on equivalences, as many equivalences give rise to rules involving nested queries. Some interesting transformation rules for nested queries can be found in [9]. A completely different approach to query optimization in the presence of methods can be found in the REVELATION project [12]: objects are asked to *reveal* the implementation of methods during query optimization. This differs from our approach as we are not interested in the imple-

mentation itself (except indirectly through cost models) but on the semantics that is realized through the implementation.

9 Conclusion

In this paper we investigated the role of methods in query optimization. We showed how query optimization can benefit from semantic knowledge about methods without revealing the *real* method implementation and thus violating the encapsulation. This knowledge can be expressed through schema-specific equivalences. We demonstrated how such equivalences can be translated into transformation and implementation rules which can be used by the query optimizer. By making some of the usually hardwired query optimizer components extensible schema-specific equivalences can be easily integrated in the optimization process. We sketched an implementation of our concepts within the object-oriented database management systems VODAK using the Volcano optimizer generator.

As already pointed out we consider the support of the extensibility of the VODAK database system as the major application of the techniques introduced. VODAK is now extensible not only at the data model level, as already realized through a metaclass concept, and on the level of transaction processing [18], but also on the query processing level. Future work will include testing these concepts in complex application domains involving new datatypes, like document storage or scientific applications. Furthermore the approach presented still leaves many research issues open. Prominent among these is to develop a methodology for obtaining and maintaining rules such that they are readily applied in the optimization process. First considerations in this direction were presented in the paper. Another issue of interest is the extension of the approach to rules that involve conditions.

References

[1] K. Aberer, K. Böhm, C. Hüser: "The prospects of publishing using advanced database concepts", *Proc. of the Intl. Conf. on Electronic Publishing*, 1994.

[2] K. Aberer, G. Fischer: "Object–Oriented Query Processing: The Impact of Methods on Language, Architecture and Optimization", *Technical Report GMD No. 763*, GMD-IPSI, July 1993.

[3] K. Aberer, G. Fischer: "Semantic Query Optimization for Methods in Object-Oriented Database Systems", *GMD Technical Report No. 849*, June 1994**.**

[4] C. Beeri: "Formalization of query languages for models with object-oriented features", *Object-Oriented Database Management Systems*, NATO ASI Series, Springer Verlag Berlin Heidelberg, August 1993 .

[5] E. Bertino, W. Kim: "Indexing Techniques for Queries on Nested Objects", *IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 2*, 1989.

[6] E. Bertino, M. Negri, G. Pelagatti, L. Sbattella: "Object-Oriented Query Languages: The Notion and the Issues", *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 3, pp. 223–237, June 1992.

[7] J.A. Blakeley, W.J. McKenna, G. Gräfe: " Experiences Building the Open OODB Query Optimizer", *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 287–296, Washington, DC, May 26–28, 1993.

[8] S. Chaudhuri, K. Shim: "Query Optimization in the Presence of Foreign Functions", *Proc. 19th VLDB*, pp. 529–542, Dublin, Ireland, August 24–27, 1993.

[9] S. Cluet, G. Moerkotte: "Nested Queries in Object Bases", *Proc. DBPL4*, Manhattan, New York City, August 30th – September 1st, 1993.

[10] G. Fischer: "Updates in Object-Oriented Database Systems Caused by Method Calls in Queries", *Proc. 3rd ERCIM Database Research Group Workshop on Updates and Constraints Handling in Advanced Database Systems*, Pisa, Italy, September 28–30, 1992.

[11] G. Gräfe, D. DeWitt: "The EXODUS Query Optimizer", *Proc. ACM SIGMOD Conference*, pp. 160–172, San Francisco, USA, May 27–29, 1987.

[12] G. Gräfe, D. Maier: "Query Optimization in Object-Oriented Database Systems: A Prospectus", *Advances in Object-Oriented Database Systems / Proc. 2nd International Workshop on Object-Oriented Database Systems*, pp. 358–363, Bad Münster am Stein-Ebernburg, Germany, September 27–30, 1988 (LNCS 334).

[13] G. Gräfe, W. J. McKenna: "The Volcano Optimizer Generator: Extensibility and Efficient Search", *Proc. 9th ICDE*, pp. 209–218, Vienna, Austria, April 19–23, 1993.

[14] J. Hellerstein, M. Stonebraker: "Predicate Migration: Optimizing Queries with Expensive Predicates", *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 267–276, Washington, DC, May 26–28, 1993.

[15] W. Klas: "A Metaclass System for Open Object-Oriented Data Models", *Dissertation*, Technical University of Vienna, January 1990.

[16] W. Klas, K. Aberer, E.J. Neuhold: "Object-Oriented Modeling for Hypermedia Systems using the VODAK Modelling Language (VML)", in: *Object-Oriented Database Management Systems*, pp. 389–434, NATO ASI Series, Springer Verlag Berlin Heidelberg, August 1993 .

[17] W. Klas, E. J. Neuhold, M . Schrefl: "Metaclasses in VODAK and their Application in Database Integration", *GMD Technical Report No. 462*, July 1990.

[18] C. Laasch, M. Scholl: "Deterministic Semantics of Set-Oriented Update Sequences", *Proc. 9th ICDE*, pp. 4–13, Vienna, Austria, April 19–23, 1993.

[19] D. Maier, J. Stein: "Development and implementation of an object-oriented dmbs", *Readings in Object-Oriented Database Systems*, pp. 167–185, Morgan Kaufman, 1990.

[20] B. McKenna, *Volcano Query Optimizer Generator Manual*, University of Colorado, Boulder, November 1992.

[21] A. Mehta, J. Geller, Y. Perl, E.J. Neuhold: "The OODB Path-Method Generator (PMG) Using Precomputed Access Relevance", *Proc. of the 2nd Int. Conference ob Information and Knowledge Management (CIKM–93)*, Arlington, USA, November 1–5 , 1993.

[22] P. Muth, T. C. Rakow, G. Weikum, P. Brössler, C. Hasse: "Semantic Concurrency Control in Object-Oriented Database Systems", *Proc. 9th ICDE*, pp. 233–242, Vienna, Austria, April 19–23, 1993

[23] L. Rowe, M. Stonebraker: "The POSTGRES Data Model", *Proc. 13th VLDB*, pp. 83–96, Brighton, England, September 1–4, 1987.

[24] G. Shaw, S. Zdonik: "Object-Oriented Queries: Equivalence and Optimization", *Proceedings of the 1st International Conference on Deductive and Object-Oriented Database Systems (DOOD '89)*, pp. 264–278, 1989.

[25] D.D. Straube, M.T. Özsu: "Queries and Query Processing in Object-Oriented Database Systems", *ACM Transactions on Information Systems, vol. 8, no. 4*, pp. 387–430, October 1990.

[26] R. Wolniewicz, G. Gräfe: "Algebraic Optimization of Computations over Scientific Databases", *Proc. 19th VLDB*, pp. 13–24, Dublin, Ireland, 1993.