published in: Eder J., Kalinichenko L. (eds.): Proceedings of the 2nd Int. East/West Database Workshop, 1994, Springer Verlag, pp. 108-120

Optimization of Object-Oriented Queries by Inverse Methods

Johann Eder, Heinz Frank, Walter Liebhart
Institut für Informatik, Universität Klagenfurt
Klagenfurt, Austria
e-mail: {eder, heinz, walter}@ifi.uni-klu.ac.at

Abstract

For object-oriented databases we propose a new technique for optimizing queries containing method invocations. This technique is based on the definition of inverse methods and query rewriting. It can be viewed as providing computed inverted access structures like (secondary) indexes provide stored inverted access structures. This technique can be applied to methods which can be fully specified as functions and to the usual comparison operations in queries. We introduce an extension to ODMG-93 [1] to define inverse methods and present the optimization algorithm for homogeneous as well as for heterogeneous collections. The application of this technique can reduce the cost of query-evaluation by orders of magnitude.

1 Introduction

Automatic optimization of queries is crucial for the applicability of declarative database query languages. The development of powerful query optimizers using efficient physical access structures was essential for the success of the relational data model. Therefore, also the optimization of queries for object-oriented databases is intensively researched [2,3,4]. However, most of the approaches proposed so far concentrate on the structural dimension of object-oriented databases while optimization of queries involving method calls has attracted comparatively little attention. Our work on schema integration [5] and view definition for object-oriented databases [6,7] made us aware of the great demand for effective optimization of such queries.

Optimization techniques reported recently include to estimate the cost of method invocation for cost-based query optimizers[2,8] and the precomputation of method calls [9,10,11] following the lines of view-materialization. The essence of the latter approach is to convert computed data (i.e. the result of method calls) to stored data so that all access structures and optimization techniques for stored data can be employed. Obviously, this technique requires considerable overhead for update operations and can only be applied to a very restricted class of methods. In particular, methods with parameters can hardly be materialized.

The approach presented here introduces computed inverted access to computed data through inverse methods. Let us introduce this technique with a small example taken from an usual schema integration problem where the scaling conflict between temperature values in degree Fahrenheit and Celsius are solved by conversion functions.

Suppose an object-oriented database to handle materials with their melting point for European and American users. During the schema integration process you decided to store the melting point of a material as degree Celsius and provide a conversion function, temp_fahrenheit, to get the temperature in Fahrenheit. If an user would like to know the names of all materials with a melting point less than 100 degree Fahrenheit, he would write the following statement (according to ODMG-93):

```
SELECT   m.name
FROM     m in Materials
WHERE    m→temp_fahrenheit () < 100
```

Traditionally, this query requires a full scan of the Materials class and an invocation of the method temp_fahrenheit for each object. With the introduction of the inverse method[1] temp_celsius, which transfers Fahrenheit to Celsius, we are able to rewrite the above query as:

```
SELECT   m.name
FROM     m in Materials
WHERE    m.melting_point < Material→temp_celsius (100)
```

This has the following advantages which can reduce the cost of query evaluation by orders of magnitude:

- The inverse method has to be computed only once, while the original method would have to be evaluated for each object.
- Access structures (indexes) can be used to avoid loading all objects from the disc into the main memory.

Obviously it is not possible to define inverse methods for all methods in the database - it is up to the database designer or tuner to decide whether it is possible and worthwhile to introduce an inverse method, like he decides for indexes. Until now the user or application programmer would have to rewrite the query instead of a query optimizer. For this purpose he needs detailed information about the internals of the objects. However, this would jeopardize view concepts and sacrifice encapsulation for performance. Using our optimization strategy protects encapsulation and view concepts and relieves the users of knowledge about internal data representation. In particular, in the case of multi-database systems, the user

[1] In section 2 we will argue why inverse methods have to be type methods.

even might not have access to the necessary information. In the example the users has to know whether the temperature is stored as degree Celsius or as degree Fahrenheit.

2 Definition of Inverse Methods

2.1 Optimizable Methods

Not every kind of method is appropriate for our optimization process. Most important, optimizable methods have to be side-effect free functions. Such methods can formally be defined as a family of functions:

For the following discussion let m be a method of the object-type C with parameter(s) P and return type Z. The method m is optimizable with an inverse method, if it can be fully specified as the following family of functions:

$$\forall P: m_P: C \rightarrow Z$$

A special subclass of these optimizable methods are those which use exactly one (distinguished) attribute for the computation. This class of methods are further characterized by the following condition:

Let a be an attribute of type A, o and o' be objects of type C, then:

$$\forall P \; \forall o, o': o.a = o'.a \Rightarrow o \rightarrow m_P = o' \rightarrow m_P$$

Such methods can be described as the following family of functions:

$$\forall P: m_P: A \rightarrow Z$$

2.2 Inverse Methods

For deciding which queries containing invocations of such methods can be optimized and for choosing the appropriate rewriting rules we need further information about properties of the methods, in particular about injectiveness and monotonicity.

2.2.1 Injective Functions

Injective functions have an inverse function. According to the above classification the inverse functions m^{-1} can be specified more precisely as:

- with distinguished attribute:
  $$m^{-1}_P: Z \rightarrow A, \text{ with } \forall a, P: m^{-1}_P(m_P(a)) = a$$
- without distinguished attribute:
  $$m^{-1}_P: Z \rightarrow C, \text{ with } \forall o, P: m^{-1}_P(m_P(o)) = o$$

To optimize inequality comparisons, we are interested in strict monotonous functions, a subclass of injective functions. The definition of the inverse of strict monotonous functions is the same as for injective functions.

2.2.2 Non Injective Functions

Non injective functions have no one-to-one inverse functions. Nevertheless, we are able to optimize non injective functions by the definition of an inverse method which maps its results into power set $\wp$. According to the above classification we again specify such methods more formally as:

- with distinguished attribute:
  m^{-1}_P: $Z \rightarrow \wp(A)$, with $\forall$ a, P: $m^{-1}_P(m_P(a)) = \{ a' \mid m_P(a) = m_P(a')\}$
- without distinguished attribute:
  m^{-1}_P: $Z \rightarrow \wp(C)$, with $\forall$ o, P: $m^{-1}_P(m_P(o)) = \{ o' \mid m_P(o) = m_P(o')\}$

2.3 Integration in ODMG-93

According to the above considerations we extend ODMG-93 with a special language construct to create an inverse method (similar to an index definition):

inverse operation <return_type> <operation_name> **(**<argument_list>**)**
on <type_name>
for <operation_name> [**injective** | **increasing** | **decreasing**]
[**based on** <attribute_name>] [<raises_expr>] <context_expr>

The definition of an inverse method starts with the keyword *inverse operation* followed by the return type, the name of the inverse method and its argument list, similar to the construct for defining a method in ODMG-93. The type, where the inverse method is attached is specified with the poperty *on*. The name of the corresponding original method is stated after the keyword *for*. To characterize the kind of the method (injective, strict monotonous increasing or strict monotonous decrasing) the optional keywords *injective*, *increasing* and *decreasing* are used. If the inverse method refers to a non injective method none of these parameters are allowed, but the return type of the inverse must be a set of the return type of the original method. The existence of a distinguished attribute is specified with the keyword *based on* followed by the name of the attribute. The parameter *raises_expr* is necessary for the treatment of exceptions raised by methods. According to ODMG-93 the definition of an inverse method ends with the method body, indicated with *context_expr* [1] in our language construct.

The definition of an inverse method underlies several constraints which can be checked automatically:

- The return type of the inverse method must be of the same type as the type of the corresponding attribute defined with the keyword *based on* or a set of this type. Without an distinguished attribute the return type of the inverse must be equal with the type specified with the *on* property (or a set of this type).

- The arguments of the inverse method must be equivalent to the argument list of the original method. Additionally, a further parameter is needed for the comparison operand. This parameter has to be the first in the argument list and its type must be the same as the result type of the original method.
- If the result type of the inverse method is a set of the type , or a set of the type of the distinguished attribute, respectively, then none of the optional parameters *injective*, *increasing* or *decreasing* is allowed.

Inverse methods are realized as type methods rather than as object methods.What are the reasons for this decision? Inverse methods could be invoked on elements of the codomain of the original method. We do not place it as (normal) method in the type of this codomain, because this type is frequently a value type (e.g. numbers) such that we cannot define methods there. Moreover, the semantic context of the inverse method is the original type. If the type of the codomain already has a suitable method, then the body of the inverse method can simply consist of a call of that method. Furthermore, the inverse method is not invoked on on an object of the type but rather results in an object (or a distuinguished attribute) such that we cannot define it as object method of the type.

Consider the small example from the introduction. The extended ODMG-93 definition of the inverse method temp_fahrenheit would be:

```
inverse operation real temp_fahrenheit (in real)
on Material
for temp_celsius increasing
based on melting_point  <context_expr>
```

We consider inverse methods as access structures belonging to the internal level of a database as they are used for optimization purpose only. Like indexes they can be added and dropped at runtime affecting the performance of the system only. Queries do not have to be reformulated when inverse methods are defined or deleted (physical data independence).

3 The Optimization Process

3.1 Rewriting Rules

We first present the rewriting rules for the optimizing queries for homogeneous collections. In the following sections we will extend this algorithm to heterogeneous collections and discuss all aspects of inheritance.

We optimize the following generic kind of queries:

```
SELECT      ....
FROM        objectVar in Type_Extent
WHERE       objectVar→Method(Parameter) RelOp ComparisonOperand
```

RelOp are the usual comparison operators, such as *equal*, *greater*, *less* and *in*. *ComparisonOperand* can be, for instance, a constant, a set or a subquery.

The condition part of the query is rewritten into:

WHERE FirstPart RelOp' SecondPart

The *FirstPart* of the rewritten query depends on the existence of a distinguished attribute, specified with the *based on* property. If there is one, then the FirstPart is rewritten to *objectVar.Attributename* otherwise only to *objectVar*.

The rewriting rules for *RelOp'* and *SecondPart* depend on the original relational operator *RelOp* and the kind of the method, as specified in table 1.

Method Kind	RelOp	RelOp'	SecondPart
injective	=	=	$T \rightarrow m^{-1}$ (CO, P)
	IN	IN	FOR x IN (CO) $\{T \rightarrow m^{-1}$ (x,P)$\}$[2]
strict monotonous increasing	$\prec, \preceq$ $\succ, \succeq$	$\prec, \preceq,$ $\succ, \succeq$	$T \rightarrow m^{-1}$ (CO, P)
strict monotonous decreasing[3]	$\prec, \preceq$ $\succ, \succeq$	$\succ, \succeq$ $\prec, \preceq$	$T \rightarrow m^{-1}$ (CO, P)
non injective	=	IN	FOR x IN (CO) $\{T \rightarrow m^{-1}$ (x,P)$\}$
	IN	IN	FLATTEN[4] (FOR x IN (CO) $\{T \rightarrow m^{-1}$ (x,P)$\}$)

T type where the inverse method m^{-1} is defined, stated by the *on* property
CO ComparisonOperand, e.g. a constant, a subquery, an object
P parameters of the original method

Table 1: Rewriting Rules

---

[2] As the inverse method must be performed on all members of the set it is necessary to use an iteration operator, which until now is not defined in ODMG-93 but for instance in O_2 [12].

[3] Strict monotonous decreasing methods require to switch the relational operators.

[4] As the result of the iterator operation is a set of sets, it is necessary to flatten the result. For this purpose we use the flatten operator defined in ODMG-93.

There may exist comparison operands, which are not valid for the inverse method, although they can be used in the original query (e.g. non surjective functions). For such operands the inverse method has to raise the special exception *inverse_exception*, which must be known by the query optimizer. If the optimizer recognizes such an exception the original statement has to be executed. The efficient treatment of such queries is subject of ongoing research.

3.3 Examples

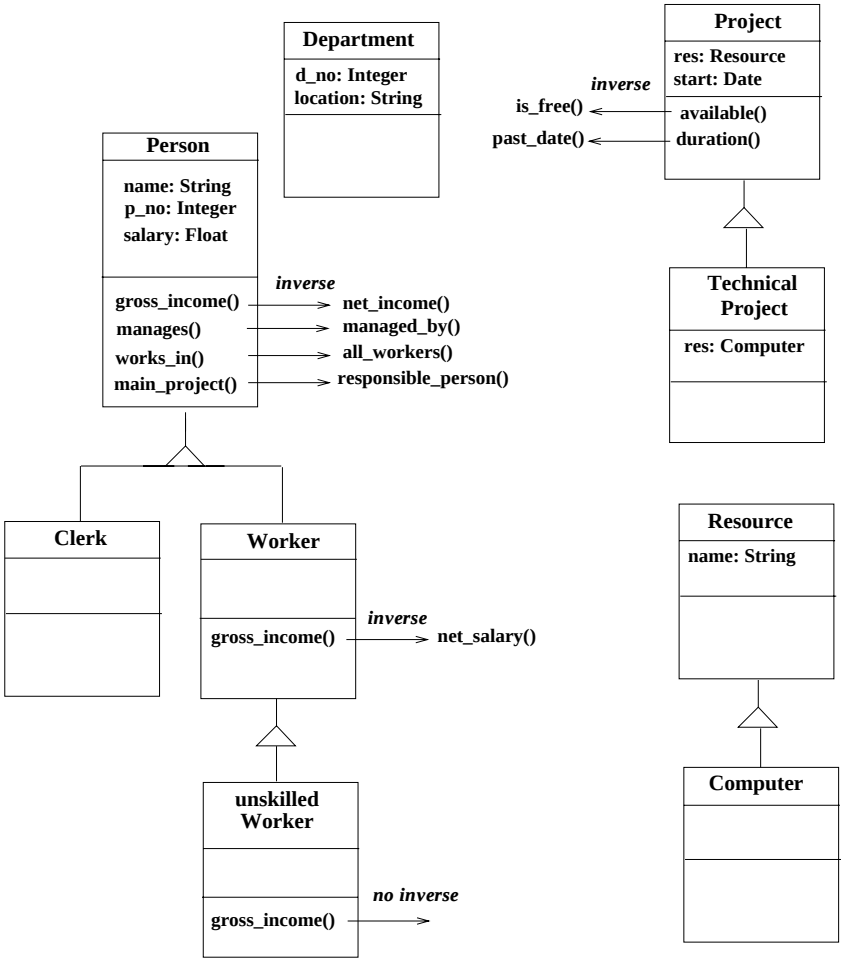

Figure 1: A Small Database Schema

Figure 1 shows an example of a small database schema in OMT-like syntax [13]. You can see a type Person with its subtypes Clerk, Worker and unskilled Worker. Each type has some attributes and methods. The inverse methods are shown with an arrow. There are also some other types, Project with the subtype Technical Project, Department and Resource with the subtype Computer.

All further query examples are based on this small schema. An informal description of each method is given in the query examples. We are aware that essential parts (such as relationships) are missing in our example. Moreover we do not consider implementation aspects of methods but concentrate only on the necessary parts to present our optimization strategy.

In the following examples we concentrate only on the condition part of the queries because of space limits.

Injective methods: We want to know the manager of the department 27 using the method manages. The inverse method is called managed_by and is based on the attribute p_no:

p→manages () = 27	p.p_no = Person→managed_by (27)

Non injective methods: We want a list of all persons working in the department 27. The method works_in in the example of figure 1 is based on the attribute p_no and returns for each person the corresponding department. However, the inverse method all_workers produces a set of values, exactly all persons being engaged in that department.

p→works_in () = 27	p.p_no IN Person→all_workers (27)

Strict monotonous decreasing methods: Compute a list of all projects running longer than 30 days. The method duration computes the duration time of a project, which is based on the attribute start. The inverse method past_date computes the date minus n days.

p→duration () > 30	p.start < Project→past_date (30)

Usage of the IN operator in a non injective method: We want to know all persons working in a department which is located in Austria.

p→works_in () IN (SELECT d.d_no FROM d in Departments WHERE d.location = "Austria")	p.p_no IN FLATTEN (FOR x IN (SELECT d.d_no FROM d in Departments WHERE d.location ="Austria") {Person→all_workers (x)})

Method without distinguished attribute: Compute a list of all persons, who have their main project located in Austria. The method main_project returns the main project of a person. The inverse method responsible_person computes the responsible person of a project.

p→main_project () IN (SELECT pro FROM pro in Projects WHERE pro.location = "Austria")	p IN (FOR x IN (SELECT pro FROM pro in Projects WHERE pro.location = "Austria") {Person→responsible_person(x)})

3.3 Advantages

The difference in the evaluation costs between the original and the rewritten query depends on the following:

- evaluation cost of the original method and its inverse
- the number of objects in the collection
- the existence of indexes
- the selectivity of the distinguished attribute
- whether the method is injective
- the cardinality of the result of the inverse method for non injective methods
- the cardinality of the comparison operand for IN comparisons
- whether the parameter of the method is independent of the object variable
- whether the comparison object depends on the object variable (correlated subquery)

A quantitative model is beyond the scope of this paper. For applying this technique the rewriting rules of a cost-based query optimizer should be extended. The advantage of this optimization technique, however, can be several orders of

magnitude. For an example take an injective method (without distinguished attribute), equality comparison, and a non-correlated comparison operand. For a collection of cardinality N it would be necessary to retrieve N objects and invoke the method N times, while the inverse method is evaluated only once and a single object is fetched from secondary memory. If the execution costs of the methods are about the same this results in a reduction of the evaluation costs of the queries by the factor N. For such methods and queries our optimization technique can even be integrated in rule based optimizers. On the other hand, if the comparison object is a correlated subquery the evaluation costs of the rewritten query might be higher, since the inverse method then has to be executed for each object of the collection. Such queries should only be optimized with a cost-based optimizer.

4 Aspects of Inheritance

4.1 Inheritance of Inverse Methods

Inverse methods are inherited, too. However, the scope of an inverse is tied to the scope of its method. If a method is overridden, neither the overridden method nor its inverse method are inherited, irrespective whether an inverse of the overriding method has been defined or not. However, it is possible to override inverse methods without overriding the original method, although this will rarely be necessary.

This general concept of inheritance is very useful for our optimization process. It may be necessary to optimize queries, whose target type is a subtype of the type where the inverse method has been defined, for instance consider a query which computes a list of clerks with a gross income greater than 30000. We can optimize this query by using the inverse method net_income defined within the type Person. To handle such queries, the optimizer needs to collect the necessary information about the inverse method, e.g. by searching the type hierarchy backwards.

4.2 Method Overriding

Our optimization technique can also be applied to heterogeneous collections, i.e. collections of objects of different types. If neither the method to be optimized nor the inverse method are overridden, then the optimization process presented in section 3 can be applied without change. Overriding of methods and inverses leads to a more complex optimization process dealing with overridden methods and therefore, with the existence of different inverse methods. Additionally, an overridden method may have no corresponding inverse method.

In our example we have three subtypes of Person, the types Clerk, Worker, and unskilled Worker (with the corresponding extents Clerks, Workers and unskilled Workers). Each of these types inherits an attribute salary, where the net income of a person is stored and a method gross_income, which computes the gross income

based on the salary. As the computation of the gross salary differs between clerks and workers the method gross_income is overridden in Workers. Obviously there exist different inverse methods called net_income and net_salary. Additionally, the gross income is overridden in unskilled Worker without defining an inverse method.

To rewrite queries over heterogeneous collections we have to distinguish the objects according to their types. Depending on the existence and the kind of inverse methods it is not always possible to fully optimize the whole query. To produce an optimized query, the condition is rewritten with a disjunction of clauses - one clause for each pivot type, i.e. the type T of the extent defined in the query and all (direct and indirect) subtypes of T, where either the method or its inverse is overridden. Let ST be the set consisting of the name of a pivot type T and all its (direct and indirect) subtypes which neither override the method m nor its inverse. For each pivot type we create the following clause:

ObjectVar.Type.Name IN ST AND
<optimized statement>

The <*optimized statement*> is the rewritten query based on the kind of the inverse method as explained in section 3. If an optimization is not possible, because of an inadequate relational operator or the overridden method has no inverse method, the original condition is used.

Consider the following condition part of a query based on our small example:

p→gross_income () > 10000	(p.Type.Name IN {Person, Clerk} AND p.salary > Person→net_income (10000)) OR (p.Type.Name IN {Worker} AND p.salary > Worker→net_salary (10000)) OR (p.Type.Name IN {unskilled_Worker} AND p→gross_income () > 10000)

As you can see method overriding decreases the advantages of our optimization strategy. Now it is necessary again to scan the whole collection, as the type of each object is necessary within the optimized query. But still the number of method calls is significant smaller than with the origin query. Obviously the advantage of our approach decreases the more methods are redefined within the type hierarchy.

4.3 Attribute Overriding

In the object-oriented paradigm the overriding of attributes is possible as the redefinition of methods within the type hierarchy. However, the overridding of attributes is restricted by the concept of covariance which is supported by ODMG-

93. The type of an attribute's redefinition must be a subtype of its original inherited definition. In the case of overridding the distinguished attribute we have to analyze possible constraints to our approach.

Suppose the following query, which computes all projects, whose resource is available on the 1st of October. The method available returns the date when the resource of the corresponding project is not any more used.

p→available () = "1.Oct. 1994"	p.res IN Project→is_free ("1.Oct. 1994")

The inverse method is_free returns a set of resources. As technical projects also belongs to the extent projects (by instance inheritance) and the attribute resource is overridden in type technical project we have to deal with the comparison of different types (in our example we have to compare objects of type Computer with objects of type Resource).

Although ODMG-93 does not consider that problem explicitly, such a query can be written in O_2. Regarding to our approach, we are able to optimize queries with an overridden distinguished attribute without any restrictions.

5 Conclusions

We presented a novel technique for the optimization of queries against object-oriented databases with method calls in the condition part of the query. Our approach introduces inverse methods as computed inverted access structures. We consider these inverse methods to somehow belong to the physical level of database systems. They can be added and dropped like stored access structures (indexes).

These inverse methods can then be used to rewrite the condition part of queries. For a significantly large class of queries our approach reduces the cost of query evaluation by orders of magnitude and our approach can be directly integrated into rule based optimizers. For other classes the cost of the original and the rewritten query have to be estimated, such that we recommend the integration only into cost-based optimizers.

Best results are obtained, when our technique is applied to queries against homogeneous collections or to heterogeneous collections, where the method used in the condition part is not overridden. Our approach has been extended to all heterogeneous collections with overridden methods too. However, the necessary type case distinction reduces some of the advantages.

References

1. Cattell R. The Object Database Standard: ODMG-93. Morgan Kaufmann Publishers, San Mateo, 1993
2. Mitchell G., Zdonik S., Dayal U. Optimization of Object-Oriented Queries: Problems and Approaches In: Dogac A., Özsu T., Biliris A., Sellis T. (ed) Proceedings of the NATO ASI Object-Oriented Database Systems, Kusadasi, Turkey, 1993, pp 30-66
3. Graefe G. Query Evaluation Techniques. ACM Computing Surveys, Vol. 25, No. 2, June 1993, pp 73-170
4. Freytag J., Maier D., Vossen G. Query Processing for Advanced Database Systems. Morgan Kaufmann Publishers, San Mateo, 1994
5. Eder J., Frank H. Schema Integration For Object Oriented Database Systems. In: Tanik M., Rossak W., Cooke D. (ed) Proceedings of Software Systems in Engineering, New Orleans, USA, 1994, pp 275-284
6. Dobrovnik M., Eder J. View Concepts for Object-Oriented Databases In: Proceedings of the 4th International Symposium on System Research, Informatics and Cybernetics, Baden, 1993
7. Dobrovnik M., Eder J. A Concept of Type Derivation for Object-Oriented Database Systems, In: Gün, Onvural R. Gelenbe E. (ed) Proceedings of the Eight International Symposium on Computer and Information Sciences (ISCIS VIII), Istanbul, 1993
8. Graefe G., Ward, K. Dynamic Query Evaluation Plans. SIGMOD Proceedings, ACM, 1989, pp 358-366
9. Bertino E. Method Precomputation in Object-Oriented Databases. SIGOIS Bulletin, 12(2,3), 1991, pp 199-212
10. Kemper A., Kilger C., Moerkotte G. Function Materialization in Object Bases. SIGMOD Proceedings, ACM, 1991, pp 258-267
11 Kemper A., Kliger C., Moerkotte G. Function Materialization in Object Bases: Design, Realization and Evaluation. IEEE Transaction on Knowledge and Data Engineering, Vol. 6, No. 4, August 1994, pp 587-608
12. The O_2 User Manual, Version 4.2, January 1993
13. Rumbaugh J. et al. Object-Oriented Modeling And Design. Prentice Hall, Englewood Cliffs, New Jersey, 1991