Processing Object-Oriented Queries with Invertible Late Bound Functions

Staffan Flodin **Tore Risch**
Dept. of Computer and Information Science
Linköping University, Linköping, Sweden
{stala, torri}@ida.liu.se

Abstract

New demands are put on query processing in Object-Oriented (OO) databases to provide efficient and relationally complete query languages. A flexible OO data model requires overloading and late binding of function names. Relational completeness requires capabilities to handle queries where functions are inverted, i.e. where it is possible to select those objects y that satisfies $fn(y)=x$ where x is known. A system that supports both late binding and inverted functions must be able to solve $fn(y)=x$ for a given x and unknown y when fn is late bound, i.e. the resolvent (implementation of a function name) to apply on y is selected based on the type of y. This combination of late binding and inverted function calls require novel query processing capabilities to fully utilize indexes referenced in late bound function calls. This paper presents an approach to the management of late binding in query processing. The main result is a query processing method where late bound function calls are efficiently executed and optimized for both inverted and regular execution. The proposed solution is based on substituting each late bound function call in the execution plan with a special function, DTR, which dynamically selects the actual resolvent to call. We define the inverse of DTR and its correctness. We show a dramatic execution time improvement by making DTR invertible and by defining its cost model for query optimization. The improvements are verified by performance measurements.

Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995

1 Introduction

The evolution of relational databases into *Object-Relational* databases has created the need for relationally complete and declarative Object-Oriented (*OO*) query languages. Development of such query languages has prompted research on new query optimization methods, e.g. [5] [21]. Good query optimization is as important for OO query languages as it is for relational query languages. A powerful OO data modelling language permits the construction of more complex schemas than for relational databases. Thus, query processing over these OO data models is at least as demanding as relational query processing.

Flexible OO data models require *overloading* and *late binding* of function[1] names [1]. Overloading of function names means having the same name to denote different implementations, where each implementation is called a *resolvent*. Late binding, as opposed to early binding, is choosing the resolvent at runtime instead of at compile time. Late binding is necessary in a flexible OO data model, e.g. when querying sets of objects of different types.

Another characteristic of flexible OO queries is the ability to invert functions, i.e. the problem of asking for which objects y a given function fn returns a specific value x, i.e. retrieving all y that satisfies $fn(y)=x$. For example, if we have a function `reports_to(employee)->manager` we not only want to know the manager of a given employee, but also all employees that report to a given manager. Of particular interest is the problem when the inverted function is overloaded and the type of its argument does not uniquely identify the implementation of the overloaded function, i.e. a late bound function call. For example, the function `reports_to` for managers may be different than for employees and we want to find all employees or managers that report to a given manager.

1. In this paper the term function is used in favour of the term method.

To give the optimizer more transformation choices, relational query optimization techniques first expand all views referenced in a query and then apply cost-based optimization strategies on the fully expanded query [16] [22]. This expansion allows the query optimizer to consider all indexes on relations referenced in a query. Relational optimizers thus do *global optimization* by looking inside all referenced views. The same principle can also be applied to OO queries. This requires that the optimizer can inspect the function definitions in the database schema [5] rather than treating them as black box routines as in C++ models [3]. However, the requirements on OO query languages to allow both late binding and powerful query optimization are in conflict, since late binding obstructs the full expansion of function definitions.

In this paper we focus on the management of such late bound function calls in an Object-Relational DBMS with a relationally complete query language. We propose a combination of solutions to the management of late bound function calls. We have also made performance measurements of our method in our research platform, AMOS, where it is shown that our method is superior to conventional late binding when indexes are available. Our approach address the following issues:

- To support global query optimization, the query compiler determines at compile time the cases when early binding cannot be used and only then it uses late binding.
- When late binding is required for some function call, the query compiler will insert a special function, DTR (*Dynamic Type Resolver*), in the execution plan. Based on the type of the object bound to the argument, DTR chooses the correct resolvent to apply from a minimal set of possible resolvents computed by the compiler.
- The DTR function and its inverse are defined in terms of the resolvents that are possible to apply.
- The cost based query optimizer needs to know the cost profile of each DTR call. DTR is regarded as an expensive predicate [10] where the cost model of DTR is defined in terms of the costs of the possible resolvents of the late bound function call. The cost model is defined for both DTR and its inverse.
- Since the binding policy is made transparent to the user, the system must incrementally recompile functions whose execution plans become invalidated due to the introduction or deletion of a resolvent.

This paper is organized as follows: In section 2 related work is discussed. In section 3 the terminology is introduced and the rules for late binding are defined. The problem of optimizing queries with inverted functions is presented in section 4. The definition, correctness and cost model for DTR and its inverse is defined in section 5. In section 6 the performance measurement is presented, and finally section 7 summarizes our experiences and outlines future work.

2 Related work

Object-Oriented query processing has gained much attention recently [5] [17] [20] [21] [23]. However, to the best of our knowledge, the problem of using inverted late bound functions has not been dealt with. The problem of having late bound function calls in execution plans was identified in the Revelation project [5] as a problem that "still presents a challenge".

A problem is that late bound function calls obstruct global optimization. Our approach is to do local optimization of the resolvents of late bound functions and then define DTR in terms of the locally optimized resolvents. Another approach to this problem is to use *dynamic query optimization* [4] where the original query plan is split into separately optimized chunks (e.g. one for each resolvent of a late bound function), and where the total query plan is generated at start-up time of the application program. We have chosen not do use dynamic optimization to avoid high overhead of optimization at runtime.

In the EXTRA/EXCESS project [25] the problem of optimizing *regular*[2] late bound function calls is discussed. They planned to use a combination of two approaches; a runtime dispatch on the dynamic type and a more complex method where the result was obtained by taking the multiset union of running several preoptimized functions. Their proposal did not address the problem of having inverted late bound function calls.

In C++ based systems [3] the management of late bound functions is controlled by the C++ compiler and indexes used inside the resolvents cannot be utilized by the compiler.

3 Type resolution in Object-Oriented queries

Our data model, AMOS [7], is an Object-Relational data model based on the functional paradigm of DAPLEX [18] and Iris [8]. Functions model object attributes and relationships between objects through three basic function types: *Stored functions* store properties of objects in the database. *Derived functions*, defined as queries, correspond to views in the relational model that are parameterized, and *Foreign functions* are defined using an external programming language [11]. Our model extends the Iris data model with rules [19] [15]. AMOS has a query language, AMOSQL, a deviate of OSQL [12].

In the AMOS data model the types are organized in a hierarchy where subtypes inherit all of their properties

2. By regular call we mean the not inverted call from their supertypes. In a subtype it is possible to redefine inherited properties and to add properties. We denote 't_i subtype of t_j' as $t_i < t_j$.

Consider the following example of a schema.

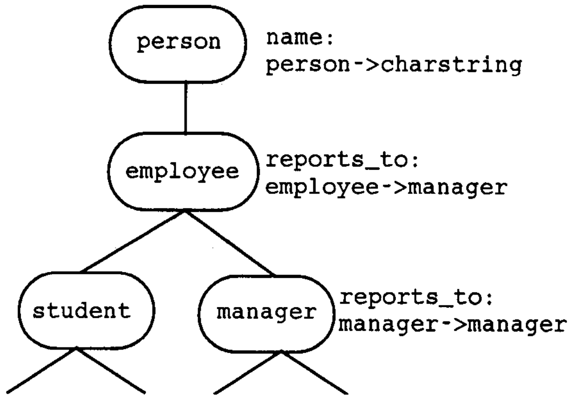


Figure 1 A type hierarchy

The types are represented as ellipses and properties are modelled as functions. The `name` property is inherited from type `person` to its subtypes, i.e. `employee`, `student` and `manager`. The property `reports_to` is defined in type `employee` and redefined in type `manager`.

Polymorphism in AMOS is controlled by the *type compatibility rule* [14] which says that a binding *ref*$\leftarrow$*o* where the reference *ref* is of type *A* and the object *o* is of type *B*, is legal only if *B* is a subtype of or equal to *A*. Therefore we have a *static type* and a *dynamic type*:

- The *static type* [14] of a reference *ref* is the declared type of the reference. We denote the static type of a reference *ref* as *S(ref)*.
  The static type of a reference to a function resolvent is the static type of its argument. For example the static type of the function resolvent `person.name`, *S*(`person.name`), is the type `person` since the function is declared to take objects of type `person` as argument.
- The *dynamic type* [14] of a reference is the type of the object referenced at runtime. We denote the dynamic type of a reference *ref* as *D(ref)*.
- We define the *dynamic type set* of a type *t* to be the set of all possible dynamic types of a reference with the static type *t*. Constrained by the type compatibility rule we have in the dynamic type set exactly *t* and all subtypes of *t*. The dynamic type set of a type *t* is denoted *T*(*t*). For any reference *ref* it holds that $D(ref) \in T(S(ref))$.

To exemplify these notions recall the schema of figure Figure.

The static type of a resolvent `person.Name` is the type `person`. The dynamic type set of type `person` is the set of types {`person, employee, student, manager`}.

Because of the type compatibility rule all instances of a type are also instances of all supertypes to that type, called *inclusion polymorphism* in [2]. This means that retrieving all the instances of a type implies retrieving all the instances of all subtypes of that type. This is a very important characteristic that is fundamental to the AMOS data model. To give a more precise description we need the notion of the *extent* of a type. The set of all instances of a type *t* that is not also a member of any subtype of type *t* is called the *extent of type t* denoted *ext(t)* [20] [6]. The *deep extent* [20] of a type *t*, denoted $ext^*(t)$, is the union of the extents of each type in the subtree of the type hierarchy rooted at type *t*. If a type t_i is a subtype of type t_j then the deep extent of type t_j will contain the deep extent of type t_i, i.e. $t_i < t_j \leftrightarrow ext^*(t_i) \subseteq ext^*(t_j)$. When properties of some type *t* are queried the entire deep extent, $ext^*(t)$, is considered.

Function name *overloading* is permitted in AMOS which is the possibility to give one function name several implementations where each implementation is called a *resolvent* of the name. Each resolvent is identified by its *signature*, i.e. an annotation of the name with the type of the argument[3]. For example the function named `reports_to` in figure Figure is overloaded with two resolvents defined on the types `employee` and `manager`. The resolvent names are `employee.reports_to` and `manager.reports_to`, respectively.

Let $resolvent(fn,t_i)$ be a function that returns the resolvent $t_j.fn$ of the function name *fn* to be applied on objects bound to a reference with static type t_i. For the returned resolvent, $t_j.fn$, there must not exist any other more specific resolvent, $t_k.fn$, that is applicable to objects of type t_i, i.e.

$$resolvent(fn,t_i) = t_j.fn \rightarrow \neg\exists t_k.fn[\quad t_i \leq S(t_k.fn) \wedge S(t_k.fn) < S(t_j.fn)]$$

Now consider the following AMOSQL query in the context of figure 1:

```
SELECT name(p) FOR EACH person p;           (1)
```

which selects the names of all persons. It returns a set where the resolvent `person.name` is applied to every element in $ext^*(person)$.

Then consider another query in the context of figure Figure

```
SELECT reports_to(e) FOR EACH employee e;(2)
```

3. Our implementation allows several arguments but here we assume one argument and one result for simplicity.

Since `reports_to` is defined for the type `employee` and redefined for the type `manager`, selecting the `reports_to` of all employees returns a set where `employee.reports_to` is applied to all elements in *ext*(*employee*) $\cup$ ext^*(*student*) and `manager.reports_to` is applied to the elements in ext^*(*manager*).

In the AMOSQL statement (2) above, the result of the query is the union of the results of applying two different resolvents to two disjoint subsets of ext^*(*employee*). By contrast, in AMOSQL statement (1) a single resolvent is applied to the entire ext^*(*employee*). Therefore, in AMOSQL statement (2) late binding must be used whereas in AMOSQL statement (1) early binding can be used.

Whenever possible, early binding should be used, but, as just shown, late binding is required sometimes. For this, the query compiler must decide at compile time, the correct binding of a function call and selects late binding if required.

4 Queries with inverted functions

A desirable feature of Object-Oriented query processing supported in the AMOS data model is the ability to invert functions. By inverting functions we refer to the problem of finding all arguments, *arg*, that satisfies *fn*(*arg*)=*value* where *value* is bound, i.e. those $arg \in fn^{-1}(value)$.

The inverses of the three function types in our data model are processed as:

•The inverse of a stored function is provided by the system; the function and its inverse can be declared indexed for fast access.

•The inverses of foreign functions are definable by the programmer [11].

•The inverses of derived functions are inferred by the system.

Derived functions are optimized using global optimization, which means that derived function calls are substituted by their bodies; the optimizer has full insight in the implementation of all derived functions. Global optimization corresponds to the *revealer* in the Revelation project [5] [13] or view expansion in relational optimization [22]. Thus, globally optimized queries contain only stored and foreign functions, and therefore only stored and foreign functions need to be invertible.

The following definition of the function `supervises` retrieves all managers `m1` that report to a certain manager `m`. It uses the inverse of the resolvent `manager.reports_to`.

```
CREATE FUNCTION
   supervises(manager m)->>manager
   AS
   SELECT m1 FOR EACH manager m1
   WHERE m=reports_to(m1);                    (3)
```

Since there exists no redefinition of the function `reports_to` in the dynamic type set of type manager it can be bound early to resolvent `manager.reports_to` and the function call can therefore be substituted by the resolvent body.

By contrast, if function `supervises` is defined to return employees instead of managers as

```
CREATE FUNCTION
   supervises(manager m)->>employee
   AS
   SELECT e FOR EACH employee e
   WHERE m=reports_to(e);                     (4)
```

then the inverse of the function `reports_to` must be bound late. Here there is an ambiguity what resolvent body to substitute for the function call to `reports_to`. To efficiently manage such late bound calls, our approach is to substitute the call by a DTR function that will select the resolvent at runtime. The problem is then how to define such a DTR function that is invertible and optimizable and utilizes indexes whenever possible.

5 An approach to management of late binding

One way of combining late and early binding is to have the programmer declare those function calls that are to be bound late, similar to *virtual functions* in C++ [24]. This is not a good solution for databases since it is very difficult to know what functions should be bound late as the database schema evolves.

Our approach is to make the binding policy transparent to the user by having the query compiler resolve when late binding must be used. When the query compiler infers that a function call in the execution plan must be bound late, it replaces the call by a function DTR. The DTR function is a general algebraic construct that can be used in any context in the execution plan. It is thus not dependent on any specific join method. The DTR function is given the set of all *possible resolvents* of the late bound function call. The set of possible resolvents of a call, *fn*(*arg*), is the set of resolvents of *fn* defined for types in the dynamic type set of the static type of the argument *arg*. Furthermore, if the resolvent applicable to the static type of the argument is inherited, it must also be added to the set. Let $res^*(fn, S(arg))$ denote the possible resolvents of a function call *fn*(*arg*). For $res^*(fn, S(arg))$ it must thus hold that:

$\forall r.fn[r.fn \in res^{*}(fn, S(arg)) \rightarrow$
$S(r.fn) \in T(S(arg)) \vee r.fn{=}resolvent(fn, S(arg))]$

The DTR call is constructed as DTR(*rs, arg*) where $rs = [t_i.fn \ldots t_j.fn]$ is the *resolvent sequence*. The resolvent sequence is $res^{*}(fn, S(arg))$ sorted into a partial order with more specific resolvents preceding less specific ones, i.e. $S(t_i.fn) < S(t_j.fn) \rightarrow i < j$, where *i, j* are the positions of the resolvents in *rs*.

For example the AMOSQL statement (4) where function `reports_to` is bound late will be transformed to:

```
CREATE FUNCTION
   supervises(employee m)->>employee
   AS
   SELECT e FOR EACH employee e
   WHERE m=DTR([manager.reports_to,
               employee.reports_to],e); (5)
```

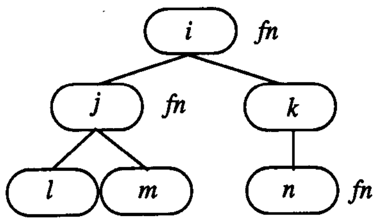
We have chosen not to globally optimize calls to DTR. Instead we separately optimize for each DTR call the possible resolvents and their inverses and then define the DTR call in terms of these execution plans. A globally optimized DTR would result in a large and complex execution plan where all possible resolvents are inlined with their type dispatch expressions.

5.1 Transparent binding policy

A function must be bound late when there exists more than one resolvent in the dynamic type set to the argument of the function, i.e.

$latebinding(fn(arg)) \leftrightarrow card(res^{*}(fn,S(arg)))>1$

where *card*(P) is the number of elements in the set P. Consider the following example



```
SELECT fn(arg) FOR EACH i arg; (i)
SELECT fn(arg) FOR EACH k arg; (ii)
SELECT fn(arg) FOR EACH j arg; (iii)
```

Figure 2 Type hierarchy

In (*i*) and in (*ii*) the function `fn` must be late bound since `fn` is redefined in *T*(*S*(`arg`)).

In (*iii*) no redefinition of `fn` exists in the dynamic type set of the type `j` and the function `fn` can be bound early to the resolvent `j.fn`. If later, a resolvent `m.fn` is created then (*iii*) becomes invalidated and must be recompiled to bind `fn` late. Thus, the query compiler must function incrementally to provide transparency of binding policy as the schema evolves. Our incremental query compiler [9] supports schema evolution by recompiling the affected functions when new resolvents are introduced or existing resolvents are deleted or modified.

5.2 Regular late bound calls

For a regular late bound function call, *fn(arg)*, where the argument, *arg*, is bound the resolvent is selected based on the dynamic type of the argument. Thus the resolvent to apply is obtained by computing *resolvent(fn,D(arg))*.

Recall that the first argument of DTR is a sorted sequence of resolvents with more specific resolvents early. The DTR function implements the computation of *resolvent(fn, D(arg))* by selecting the first resolvent, *t.fn*, in the sorted sequence that satisfies $S(t.fn) \geq D(arg)$. In this way the overhead of resolvent resolution is *O(n)*, where *n* is the cardinality of the DTR resolvent sequence. *n* is always less than or equal to the total number of resolvents of a given function name in the database schema. The algorithm for DTR is:

```
resolvents=DTR resolvent sequence
resolvent=first(resolvents)
while resolvent!=NULL
    if D(arg)≤S(resolvent)
        return(apply(resolvent,arg))
    end if
    resolvents=resolvents-resolvent
    resolvent=first(resolvents)
end while
```

Figure 3 The DTR algorithm

Compile time cost based optimization is used in AMOS, i.e. the costs and *fanout* of calls are estimated at compile time. The fanout of a function call is the estimated number of result tuples [11].

The execution cost of a late bound call must be estimated based on the possible resolvents that can be applied. For example, estimating the cost of (*ii*) in figure Figure is done based on the cost and fanout of the resolvents `i.fn` and `n.fn`.

Avoiding bad execution plans is more important than finding the optimal plan. Therefore, we adopted the conservative approach to use the maximum cost and the max-

$$DTR^{-1}\left([t_i fn^{-1},\dots,t_n fn^{-1}], val\right) = \bigcup_{i=1}^{n} E_{DTR}\left(t_i fn^{-1}(val), [t_i fn^{-1},\dots,t_n fn^{-1}]\right)$$

Figure 4 Result of DTR^{-1}

imum fanout of the possible resolvents as estimates of the cost and fanout, respectively, of a DTR call.

5.3 Inverted late bound calls

Consider again AMOSQL statement (4) containing a late bound call to `reports_to`.

If the function `reports_to` was uninvertible the only way of executing this query would be to apply the `reports_to` function on each object in *ext*(employee)* to check if the result is equal to `m`. The execution cost of the query would then be $O(card(ext^*(employee)))$.

Furthermore, consider the following query:

```
SELECT y FOR EACH number y
   WHERE add1(y)=9;                                   (6)
```

The derived function `add1` is overloaded on argument types *number* and *real* where *real* is a subtype of *number*. The call to `add1` in this query will be an inverted late bound call.

Executing this query without the possibility to invert `add1` would mean browsing through *ext*(number)* which is equal to $ext^*(integer) \cup ext^*(real)$ since *integer* and *real* are subtypes to *number*. Obviously such a query is not executable in finite time.

To handle (4) and (6) efficiently late bound calls must be replaced by an inverted call to DTR, DTR^{-1}.
If DTR($[t_1.fn,\dots, t_n.fn]$, *arg*)=*res* then
$arg \in$ DTR^{-1}($[t_1.fn^{-1},\dots, t_n.fn^{-1}]$, *res*) [4]. Thus, the execution strategy of DTR^{-1} is defined in terms of the inverses of the elements in its resolvent sequence. The following must be addressed to define DTR^{-1}:

•The correct result must be defined.

•An efficient execution strategy must be devised.

•A cost model must be defined.

Correctness

In AMOSQL statement (6) the function `add1` should produce all numbers `y` that satisfy `add1(y)=9`. That is, all objects in *ext*(number)* that satisfies the condition. Analogously in AMOSQL statement (4), the desired result is all objects e in *ext*(employee)* that satisfies `reports_to(e)=m`.

An inverted late bound function call, $R=fn^{-1}(val)$, is considered as correct if each object in the result set, R, when used as argument to the function *fn* produces the value *val*.

Definition of correctness

Let $R=\{o_1\, o_2 \dots o_n\}$, where $R=fn^{-1}(val)$ is the result of executing the inverse of $fn(arg)=val$.
An inverted late bound function call $R=fn^{-1}(val)$ is correct if and only if
$\forall o\, \exists t.fn[o \in R \wedge t.fn=resolvent(fn, D(o)) \rightarrow t.fn(o)=val]$

Definition of DTR^{-1}

To execute DTR^{-1} means to execute the inverses of all resolvents in its resolvent sequence. Therefore, to be able to optimize DTR^{-1} all resolvents in the resolvent sequence must be optimized for inverse execution. If any resolvent in its resolvent sequence lacks an inverse then the DTR^{-1} is also lacking an inverse and is considered uninvertible.

The result of DTR^{-1}($[t_1.fn^{-1},\dots, t_n.fn^{-1}]$, *val*) is the union result of a special execution strategy, E_{DTR}, applied to the result of each of the possible resolvents, figure 4.

We introduce E_{DTR} to remove the objects in the result of a resolvent that belongs to the deep extent of a more specific resolvent. Without E_{DTR} the previously defined correctness will be violated. Before E_{DTR} is defined the idea is illustrated by an example:

Consider the following two definitions of `reports_to`:[5]

```
CREATE FUNCTION
   manages(manager m)->department AS
   SELECT d FOR EACH department d
   WHERE mgr(d)=m;                                    (7)

CREATE FUNCTION
   reports_to(employee e) -> manager AS
   SELECT mgr(dept(e));                               (8)

CREATE FUNCTION
   reports_to(manager m) -> manager AS
   SELECT mgr(super(manages(m)));                     (9)
```

4. By $t_1.fn^{-1}$ we mean $(t_1.fn)^{-1}$

5. The function super returns the next higher department.

Type	Inst	Properties	Inst	Properties
employee	e1	dept(e1)=d2	e2	dept(e2)=d2
manager	m1	dept(m1)=d1	m2	dept(m2)=d2
department	d1	super(d1)=d2, mgr(d1)=m1	d2	mgr(d2)=m2

Figure 5 Example database population

Type `manager` is a `subtype` to type `employee`. In addition to these types there is a type `department`. The database is populated with two employees, two managers and two departments as show in figure 5.

The query:

```
SELECT e FOR EACH employee e
   WHERE reports_to(e)=m2;                    (10)
```

will be rewritten as

```
SELECT e FOR EACH employee e
   WHERE e IN DTR^-1([manager.reports_to^-1,
                  employee.reports^-1],
                  m2);                        (11)
```

According to the formula in figure 4 the result of DTR^{-1} will then be the combined result of the two resolvents where `manager.reports_to`$^{-1}$`(m2) = {m1}` and `employee.reports_to`$^{-1}$`(m2) = {m2, e1, e2}`. Notice that `m2` has to be removed from the result of `employee.reports_to`$^{-1}$`(m2)`. If not removed `m2` is violating the correctness definition since `reports_to(m2)`$\neq$`m2`. The result of the query will be the set `{m1, e1, e2}`.

Execution of resolvents in DTR^{-1}

To satisfy the formula in figure 4 the result of executing each resolvent in the resolvent sequence must not include any object o that is in the extent of the static type of a more specific resolvent in the resolvent sequence. Thus, for each $t_i.fn^{-1}$ in the resolvent sequence, rs, the result of $E_{DTR}(t_i.fn^{-1}(val), rs)$ the following must hold:

Let $R=\{o_1\ o_2 \ldots o_n\}$, where $R=E_{DTR}(t_i.fn^{-1}(val), rs)$.

$$\forall o\ \forall t_j.fn[o \in R \wedge t_j.fn \in rs \wedge S(t_j.fn) < S(t_i.fn) \rightarrow o \in ext^*(S(t_i.fn)) \wedge o \notin ext^*(S(t_j.fn))]$$

DTR^{-1} algorithm

From the above, an execution algorithm can be devised for DTR^{-1}. The algorithm uses a function *instance_of* that given an object returns the most specific type of the object. By having the sequence of resolvents in DTR sorted into a partial order with more specific resolvent early, the DTR^{-1} execution algorithm is:

```
types={}
resolvents=DTR^-1 resolvent sequence
result={}
For Each resolvent in resolvents
    tmpres=apply(resolvent res)
    For Each o in tmpres
        validresult=TRUE
        For Each t in types
            If $instance\_of(o) \le t$
                then validresult=FALSE
            end if
        end For Each
    If validresult then $result=result \cup o$
    end For Each
    $types=types \cup S(resolvent)$
end For Each
return(res)
```

Figure 6 DTR^{-1} algorithm

The above algorithm executes every resolvent in the DTR resolvent sequence and removes those objects that should be the result of a more specific resolvent. The removal is done with the **if** statements, where the set *result* is extended with the object *o* if that object is an instance of a type that is not a suptype to or equal to any type in the set *types*.

DTR^{-1} cost model

The cost of DTR^{-1}, *C*, is the sum of the costs of the possible resolvents plus the cost of executing DTR^{-1} itself. The fanout, *F*, is estimated as the sum of the fanouts of the possible inverse resolvents. Let $c_1 \ldots c_n$ be the execution costs

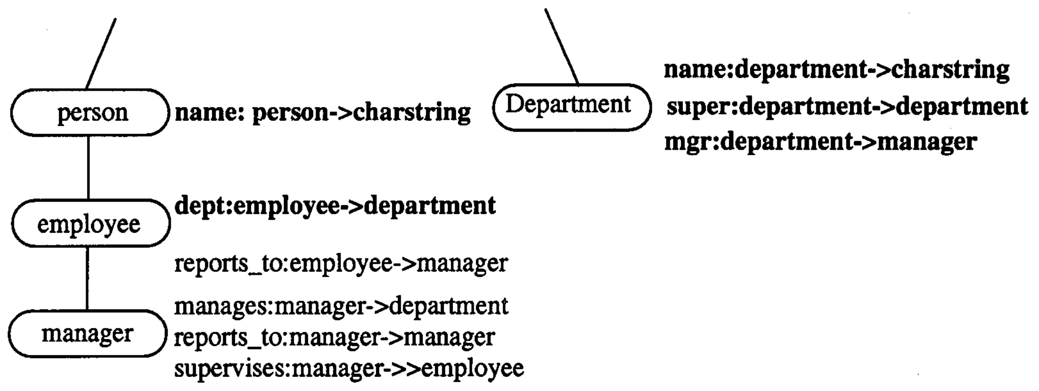

person
name: person->charstring
employee
dept:employee->department
reports_to:employee->manager
manager
manages:manager->department
reports_to:manager->manager
supervises:manager->>employee
Department
name:department->charstring
super:department->department
mgr:department->manager

Figure 7 Test database schema

and $f_1 \ldots f_n$ the fanouts of $t_1.fn^{-1}(res) \ldots t_n.fn^{-1}(res)$, respectively. The cost and fanout of DTR^{-1} is then estimated as shown in figure 8. The k in the cost formula is the overhead of checking the type of every object o emitted from each resolvent. The cost and fanout are used in the cost based optimizer to decide when DTR^{-1} favourable compared to DTR.

$$C = \sum_{i=1}^{n} k \times f_i + \sum_{i=1}^{n} c_i \qquad F = \sum_{i=1}^{n} f_i$$

Figure 8 DTR^{-1} execution cost and fanout

6 Performance measurement

The performance of having only DTR was compared experimentally to the performance of having both DTR and DTR^{-1} for the AMOSQL function (4). The test database schema is pictured in figure 7 where bold names denote stored functions and the other functions are derived functions. All stored functions have indexes on their argument and their result.

The derived functions in the schema are defined previously in the AMOSQL examples (4), (7), (8) and (9).

The function `supervises` binds late the inverted call to the function `reports_to`. What this test shows is that inverted late bound calls with index utilization can decrease the execution time considerably.

The database was populated automatically where all objects are given unique names. The stored functions `Super:department->department`, `Mgr:department->manager` and `Dept:employee->department` were populated randomly. The performance measure is the normalized execution time of function `supervises` with a randomly chosen manager as argument. The two strategies were tested on the same database with the same managers as argument to function `supervises`. The database was scaled up in each test for managers / employees as 1/10 2/40 5/100 25/500 50/1000 250/5000 500/10000. The result is pictured in figure 9.

Figure 9 shows that the cost of executing `supervises` using DTR^{-1} is constant when there are more than 40 employees as it should be, since there are constantly 20 employees per department and the cost of DTR^{-1} is proportional to the fanout of the resolvents plus the fixed execution cost. The cost of executing each resolvent is constant since hash indexes are used. By contrast, the execution of `supervises` using DTR in the forward direction is linear to $ext^*(employee)$ as expected. Note that it will be marginally cheaper to choose DTR in favour of DTR^{-1} when $card(ext^*(employee))<50$. With a proper k value in the cost model of DTR^{-1}, figure Figure, the optimizer will choose the correct strategy.

7 Summary and future work

Having late binding in the query language is necessary in the presence of inheritance and operator overloading. In database query languages late binding is somewhat problematic since good query optimization is very important to achieve good performance. Late bound function calls cannot be fully optimized at compile time; thus some work has to be done at run time. It is important to do as little as possible at runtime.

In this paper we have given a solution to the management of late binding. We have shown how to decide when late binding must be used, and that schema evolution in presence of a transparent binding policy requires an incremental query compiler. We introduced a DTR function to handle late bound function calls. We also defined its inverse, DTR^{-1}, and its correctness criteria were formalized. Cost models for DTR and DTR^{-1} were defined and used in a cost based query optimizer. We proved that DTR needs to be invertible and optimizable for efficient execu-

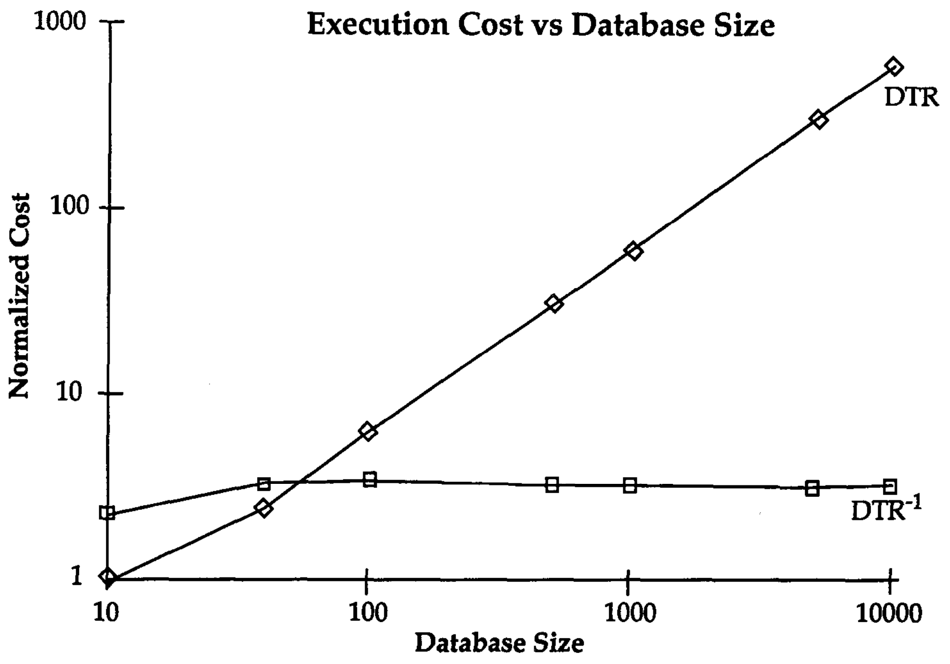


Figure 9 Test result graph

tion of queries with late bound function calls. As verified in the performance measurement example, the performance is drastically improved by using DTR^{-1} when appropriate.

We have chosen to do local optimization of the resolvents in the DTR resolvent list. This is due to the complicated nature of their execution and their type dispatch. The DTR is then viewed by the optimizer as an expensive predicate. Subject of future work is to investigate optimization methods that produce better plans than the present local optimization. It might for example be beneficial to find common subexpressions among the resolvents and move these subexpressions to the enclosing function body, thus going further towards global optimization. Along with improved optimization techniques the cost model might prove to be too conservative and a good heuristic estimate based on the cost and fanout of the possible resolvents might prove to be the best solution. The problem is, however, to find such a heuristic.

Acknowledgments

This project was supported by NUTEK(The Swedish National Board for Industrial and Technical Developement), TFR (Swedish Research Counsil for Engineering Science), and CENIIT(The Center for Industrial Information Technology).

References

1. M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik, "The Object-Oriented Database System Manifesto", *Proc. of the 1st Intl. Conf. on Deductive and Object-Oriented Databases,* Kyoto, Japan, Dec. 1989.

2. L. Cardelli, P. Wegner "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys,* Vol. 17, No. 4, 1985

3. R. G. G. Catell, T. Atwood, J. Duhl, G. Ferran, M. Loomis, D. Wade, "The Object Database Standard: ODMG - 93" *Morgan Kaufmann publishers*

4. R. L. Cole, G. Graefe, "Optimization of Dynamic Query Evaluation Plans", *Proc. of the ACM SIGMOD Conference,* May 1994, pp 150-160.

5. S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt, B. Vance, "Query Optimization in Revelation, an Overview*" *IEEE Data Engineering bulletin, Vol. 14, No. 2,* June 1992, pp. 58-62

6. U. Dayal, "Queries and Views in an Object-Oriented Data Model", *Proc. 2nd Intl. Workshop on Database Programming Languages,* 1989.

7. G. Fahl, T. Risch, M. Sköld, "AMOS - An Architecture for Active Mediators" *Proc. Intl. Workshop on Next Generation Information Technologies and Systems*, Haifa, Israel, June 1993, pp 47-53.

8. D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, W.K. Wilkinson, " Overview of the Iris DBMS", *Object-Oriented Concepts, Databases, and Applications,* ACM Press, Addison-Wesley Publ. Comp., 1989

9. S. Flodin "An Incremental Query Compiler with Resolution of Late Binding" *Research Report LITH-IDA-R-94-46,* Department of Computer and Information Science, Linköping University

10. J. M. Hellerstein, M. Stonebraker, "Predicate Migration Optimizing Queries with Expensive Predicates", *Proc. of the 1993 ACM SIGMOD Conference*, pp. 267-276.

11. W. Litwin, T. Risch, "Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates", *IEEE Transactions on Knowldge and Data Engineering, Vol 4, No. 6,* December 1992.

12. P. Lyngbaek, "OSQL: A Language for Object Databases", *Technical Report HPL-DTD-91-4,* Hewlett-Packard Company, 1991.

13. D. Mayer, S. Daniels, T. Keller, B. Vance, G. Graefe, W. McKenna, "Challenges for Query Processing in Object-Oriented Databases", *Query Processing for Advanced Database Systems,* Morgan Kaufmann Publishers, 1994.

14. B. Meyer, "Object-Oriented Software Construction", Prentice Hall, 1988.

15. T. Risch, M. Sköld, "Active Rules based on Object Oriented Queries", *IEEE Data Engineering bulletin, Vol. 15, No. 1-4,* Dec. 1992, pp. 27-30

16. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, "Access Path Selection in a Relational Database Management System", 1979 *ACM SIGMOD conf.,* pp. 23-34, 1979.

17. G. M. Shaw, S. B. Zdonik, "Object-oriented queries: Equivalence and Optimization", *Proc. 1st Conf. Deductive and OO Databases, 1989,* pp. 264-278.

18. D. W. Shipman, "The Functional Data Model and the Data Language DAPLEX", *ACM Transactions on Database Systems, Vol. 6, No. 1,* March 1981.

19. M. Sköld, "Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques", *Lic Thesis No 452,* Department of Computer and Information Science, Linköping University.

20. D. D. Straube, M. T. Özsu, "Queries and Query Processing in Object-Oriented Database Systems", *ACM Trans. on Information Systems, Vol. 8, No. 4,* October 1990

21. D.D Straube, M. T. Özsu. "Query optimization and Execution Plan Generation in Object-Oriented Data Management Systems", To be published in *IEEE Transactions on Knowledge and Data Engineering, April 1995.*

22. M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification", *Proc. of the 1975 ACM SIGMOD Conf.*

23. M. Stonebraker, L. Rowe, "The design of POSTGRES", *Proc. of the 1986 ACM SIGMOD Conf.,* pp. 340-355

24. B. Stroustrup, "The C++ Programming Language", pp. 189-191., Addison Wesley, 1991.

25. S. Vandenberg, D. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance"., *Proc. of the 1991 ACM SIGMOD conf.,* pp 158-167