Advanced Query Processing in Object Bases Using Access Support Relations

Alfons Kemper *Guido Moerkotte*

Universität Karlsruhe
Fakultät für Informatik
D-7500 Karlsruhe, F. R. G.
Netmail: *kemper/moer*@ira.uka.de

Abstract

Even though the large body of knowledge of relational query optimization techniques can be utilized as a starting point for object-oriented query optimization the full exploitation of the object-oriented paradigm requires new, customized optimization techniques—not merely the assimilation of relational methods. This paper describes such an optimization strategy used in the *GOM* (Generic Object Model) project which combines established relational methods with new techniques designed for object models. The optimization method unites two concepts: (1) ***access support relations*** and (2) ***rule-based query optimization***. Access support relations constitute an index structure that is tailored for accessing objects along reference chains leading from one object to another via single-valued or set-valued attributes. The idea is to redundantly maintain frequently traversed reference chains separate from the object representation. The rule-based query optimizer generates for a declaratively stated query an evaluation plan that utilizes as much as possible the existing access support relations. This makes the exploitation of access support relations entirely transparent to the database user. The rule-based query optimizer is particularly amenable to incorporating search heuristics in order to prune the search space for an optimal (or near-optimal) query evaluation plan.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1 Introduction

Object-oriented database systems are emerging as the next generation DBMSs for the non-standard application domains. However, these systems are still not adequately optimized: for applications which involve a lot of associative searching for objects on secondary memory they still have problems even to keep up with the performance achieved by, for example, relational DBMSs. Yet it is essential that the object-oriented systems will yield at least the same performance that relational systems achieve; otherwise their acceptance in the engineering field is jeopardized even though they provide higher functionality by type extensibility and type-associated operations that model the context-specific behavior. Engineers are generally not willing to trade performance for extra functionality and expressive power. Therefore, we conjecture that the next few years will show an increased interest in optimization issues in the context of object-oriented DBMSs. The contribution of this paper can be seen as one important piece in the mosaic of performance enhancement methods for object-oriented database applications.

Of course—as some authors point out, e.g., [9]—there are vast similarities between query processing in relational DBMSs and object bases. Therefore, the large body of knowledge of relational optimization techniques (e.g., [8, 16]) can serve as a basis. However, the full potential of the object-oriented paradigm can only be exploited for optimization if new access support techniques are tailored specifically for the object-oriented model(s)—and not merely assimilated from the relational model. The ***access support relations*** (ASRs)—first introduced in [10]—constitute one such approach. Access support relations form the basis of the query optimization strategy in the *GOM* (Generic Object Model) database system. They are a generalization of an indexing technique for path expressions first proposed for the GemStone data model [15] and, later, applied to ORION in [1]. Whereas the GemStone (and ORION) path expressions were limited to only single-valued attributes the access support relations allow also set-valued attributes along the path. Also, access support relations can be maintained in four different *extensions*, determining the amount of reference information that is kept in the index structure. Furthermore, an access support relation can be decomposed into arbitrary large *partitions*, which allows to adjust the indexing scheme to particular application profiles.

After briefly reviewing the access support relations the second part of this paper describes the essential parts of a rule-based query optimizer which—unlike the GemStone system—makes the exploitation of existing access support relations entirely transparent to the database user. Rule-based query optimization is not an entirely new idea: it is borrowed from relational query optimization, e.g., [5, 8, 13, 14]. [6] reports on a rule-based query optimizer generator, which was designed for their database generator EXODUS [2]. In the present work the idea of rule-based query optimization is utilized as a powerful tool to integrate the new index structure based on access support relations in object-oriented query evaluation. It is shown that the rule-based approach leads to a very modular design of such a complex transformation system. This enables the designer to experiment with different search heuristics to limit the number of transformations that have to be considered to derive a near-optimal evaluation plan.

Related work on object-oriented query processing is reported in [9, 12] where a graph-based approach was chosen for optimizing a limited class of queries, i.e., only queries that correspond to an acyclic graph are considered. Also, the cited work does not take general access support relations into account—it is based solely on (binary) indexes as known in relational DBMSs.

The remainder of this paper is organized as follows. In Section 2 we review the access support relations as a means for access support along reference chains. Then in Section 3 we introduce a QUEL-like query language, for which a term representation is developed. The transformation rules are discussed in Section 4. In order to reduce the search costs we develop heuristics for the sequence of applying the transformation rules in Section 5. Section 6 concludes the paper with a summary and a discussion of future developments.

2 Access Support Relations

In an earlier paper [10] we introduced *access support relations* as an index structure to support the evaluation of *path expressions*. They are briefly reviewed here.

A path expression has the form

$$o.A_1.\cdots.A_n$$

where o is a tuple structured object containing the attribute A_1 and $o.A_1.\cdots.A_i$ refers to an object or a set of objects, all of which have an attribute A_{i+1}. Thus, the result of the path expression is the set R_n, which is recursively defined as follows:

$$\begin{array}{rcl} R_0 & := & \{o\} \\ R_i & := & \bigcup\limits_{v \in R_{i-1}} v.A_i \quad \text{for } 1 \leq i \leq n \end{array}$$

Thus, R_n is a set of OIDs of objects of type t_n or a set of atomic values of type t_n if t_n is an atomic data type, such as *INT*.

It is also possible that the path expression originates in a collection C of tuple-structured objects, i.e., $C.A_1.\cdots.A_n$. Then the definition of the set R_0 has to be revised to: $R_0 := C$.

Formally, a path expression or attribute chain is defined as follows:

Definition 2.1 (Path Expression) *Let $t_0, \ldots, t_n$ be (not necessarily distinct) types. A path expression on t_0 is an expression $t_0.A_1.\cdots.A_n$ iff for each $1 \leq i \leq n$ one of the following conditions holds:*

- *Type t_{i-1} is defined as* **type** t_{i-1} **is** $[\ldots, A_i : t_i, \ldots]$, *i.e., t_{i-1} is a tuple with an attribute A_i of type t_i*[1].
- *Type t_{i-1} is defined as* **type** t_{i-1} **is** $[\ldots, A_i : t'_i, \ldots]$ *and the type t'_i is defined as* **type** t'_i **is** $\{t_i\}$, *i.e., t'_i is a set type whose elements are instances of t_i. In this case we speak of a set occurrence at A_i in the path $t_0.A_1.\cdots.A_n$.*

The second part of the definition is useful to support access paths through sets (note, however, that we do not permit powersets). If it does not apply to a given path the path is called *linear*. An access path that contains at least one set-valued attribute is called *set-valued*.

For simplicity we require each path expression to originate in some type t_0; alternatively we could have chosen a particular collection C of elements of type t_0 as the anchor of a path.

Since an access path can be seen as a relation we will use relation extensions to represent access paths. The next definition maps a given path expression to the underlying access support relation declaration.

Definition 2.2 (Access Support Relation) *Let $t_0, \ldots, t_n$ be types, $t_0.A_1.\cdots.A_n$ be a path expression.*

[1]meaning that the attribute A_i can be associated with objects of type t_i or any subtype thereof

Then the access support relation $[\![t_0.A_1.\cdots.A_n]\!]$ is of arity $n+1$ and has the following form:

$$[\![t_0.A_1.\cdots.A_n]\!] : [S_0, \ldots, S_n]$$

The domain of the attribute S_i is the set of identifiers (OIDs) of objects of type t_i for $(0 \leq i \leq n)$. If t_n is an atomic type then the domain of S_n is t_n, i.e., values are directly stored in the access support relation.

We distinguish several possibilities for the extension of such relations. To define them for a path expression $t_0.A_1.\cdots.A_n$ we need n temporary relations $E_1, \ldots, E_n$.

Definition 2.3 (Temporary Binary Relations) *For each A_j $(1 \leq j \leq n)$ we construct the temporary binary relation E_j. Relation E_j contains the tuples $(id(o_{j-1}), id(o_j))$ for every object o_{j-1} of type t_{j-1} and o_j of type t_j such that*

- *$o_{j-1}.A_j = o_j$ if A_j is a single-valued attribute*[2].
- *$o_j \in o_{j-1}.A_j$ if A_j is a set-valued attribute.*

Example 2.1 Consider the following database schema:

> **type** EMP **is** [Name: STRING, WorksIn: DEPT,
> Cars: CARSET, Salary: INT]
> **type** DEPT **is** [Name: STRING, Mgr: EMP,
> Profit: INT]
> **type** CARSET **is** { CAR }
> **type** CAR **is** [License: STRING, Make: STRING,
> HorsePower: INT]

Complex attributes in *GOM* are—like in almost all other object models—maintained uni-directionally. For example, in an extension of the above schema there exists a reference in the form of a stored OID from an *EMP*loyee to his *DEPT*, but not vice versa.

A path expression on this schema is:

$$EMP.WorksIn.Mgr.Cars.Make$$

The binary relations $E_1, \ldots, E_4$ may have the following extensions:

E_1

OID_{EMP}	OID_{DEPT}
...	...
id_2	id_5
id_1	id_4
...	...

E_2

OID_{DEPT}	OID_{EMP}
...	...
id_5	id_7
id_9	id_8
...	...

E_3

OID_{EMP}	OID_{CAR}
id_7	id_{11}
id_7	id_{12}
id_1	id_{13}
...	...

E_4

OID_{CAR}	$STRING$
id_{13}	"Benz"
id_{11}	"Jaguar"
id_{12}	"BMW"
...	...

[2]If t_n is an atomic type then $id(o_n)$ corresponds to the value $o_{n-1}.A_n$.

The id_j for $j = \{1, 2, 3, \ldots\}$ denote object identifiers which are system-wide unique. □

Let us now introduce different possible extensions of the access support relation $[\![t_0.A_1.\cdots.A_n]\!]$. We distinguish four extensions:

1. the *canonical* extension, denoted $[\![t_0.A_1.\cdots.A_n]\!]_{can}$ contains only information about complete paths, i.e., paths originating in t_0 and leading to t_n. Therefore, it can only be used to evaluate queries that originate in an object of type t_0 and "go all the way" to t_n.
2. the *left-complete* extension $[\![t_0.A_1.\cdots.A_n]\!]_{left}$ contains all paths originating in t_0 but not necessarily leading to t_n, but possibly ending in a *NULL*.
3. the *right-complete* extension $[\![t_0.A_1.\cdots.A_n]\!]_{right}$, analogously, contains paths leading to t_n, but possibly originating in some object o_j of type t_j which is not referenced by any object of type t_{j-1} via the A_j attribute.
4. finally, the full extension $[\![t_0.A_1.\cdots.A_n]\!]_{full}$ contains all partial paths, even if they do not originate in t_0 or do end in a *NULL*.

Definition 2.4 (Extensions) *Let $\bowtie$ (⟗, ⟕, ⟖) denote the natural (outer, left outer, right outer) join on the last column of the first relation and the first column of the second relation. Then the different extensions are obtained as follows:*

$$\begin{array}{lcl}
[\![t_0.A_1.\cdots.A_n]\!]_{can} & := & E_1 \bowtie \cdots \bowtie E_n \\
[\![t_0.A_1.\cdots.A_n]\!]_{full} & := & E_1 \text{⟗} \cdots \text{⟗} E_n \\
[\![t_0.A_1.\cdots.A_n]\!]_{left} & := & (\cdots(E_1 \text{⟕} E_2) \text{⟕} \cdots \text{⟕} E_n) \\
[\![t_0.A_1.\cdots.A_n]\!]_{right} & := & (E_1 \text{⟖} (\cdots \text{⟖} (E_{n-1} \text{⟖} E_n)\cdots)
\end{array}$$

Example 2.2 For our example path of Example 2.1 the canonical extension $[\![EMP.WorksIn.Mgr.Cars.Make]\!]_{can}$ looks as follows:

[\![EMP.WorksIn.Mgr.Cars.Make]\!]**can**

OID_{EMP}	OID_{DEPT}	OID_{EMP}	OID_{CAR}	$STRING$
...	...	...	...	...
id_2	id_5	id_7	id_{11}	"Jaguar"
id_2	id_5	id_7	id_{12}	"BMW"
...	...	...	...	...

Definition 2.5 (Decomposition) *Let $[\![t_0.A_1.\cdots.A_n]\!]_X$ be an $(n+1)$-ary access support relation with attributes $S_0, \ldots, S_n$ under extension X, for $X \in \{can, full, left, right\}$. Then the relations*

$$\begin{array}{lcll}
[\![t_0.A_1.\cdots.A_n]\!]_X^{0,i_1} & : & [S_0, \ldots, S_{i_1}] & \textit{for } 0 < i_1 \leq n \\
[\![t_0.A_1.\cdots.A_n]\!]_X^{i_1,i_2} & : & [S_{i_1}, \ldots, S_{i_2}] & \textit{for } i_1 < i_2 \leq n \\
\cdots & & & \\
[\![t_0.A_1.\cdots.A_n]\!]_X^{i_k,n} & : & [S_{i_k}, \ldots, S_n] & \textit{for } i_k < n
\end{array}$$

are called a decomposition of $[\![t_0.A_1.\cdots.A_n]\!]_X$. The relations $[\![t_0.A_1.\cdots.A_n]\!]_X^{i_j,i_{j+1}}$ for $(1 \le j < k)$, called partitions, are materialized by projecting the corresponding attributes of $[\![t_0.A_1.\cdots.A_n]\!]_X$. If every partition is a binary relation the decomposition is called binary. The above decomposition is denoted by $(0, i_1, i_2, \ldots, i_k, n)$.

The storage structure of access support relations is borrowed from the binary join index proposal by Valduriez [17]. Each partition is redundantly stored in two B$^+$-trees: the first being clustered (keyed) on the left-most attribute and the second being clustered on the right-most attribute. This storage scheme is well suited for traversing paths from left-to-right (forward) as well as from right-to-left (backward) within the access support relations even if they span over several partitions.

The different decompositions and extensions provide the database designer a large spectrum of design choices to tune the access support relations for particular application characteristics ([10] and, in more detail, [11] contain cost models that can be used to determine the best configuration for a given load profile).

The next definition states under what conditions an existing access support relation can be utilized to evaluate a path expression that originates in an object (or a set of objects) of type s. The predicate is essentially the formalization of the characteristics of the four extensions described on the previous page.

Definition 2.6 (Applicability) *An access support relation $[\![t_0.A_1.\cdots.A_n]\!]_X$ under extension X is applicable for a path $s.A_i.\cdots.A_j$ under the following condition—depending on the extension X:*

$$Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_i.\cdots.A_j) =$$

$$\begin{cases} X = full & \wedge \quad s \le t_{i-1} \quad \wedge \quad 1 \le i \le j \le n \\ X = left & \wedge \quad s \le t_{i-1} \quad \wedge \quad 1 = i \le j \le n \\ X = right & \wedge \quad s \le t_{i-1} \quad \wedge \quad 1 \le i \le j = n \\ X = can & \wedge \quad s \le t_{i-1} \quad \wedge \quad 1 = i \le j = n \end{cases}$$

Here $s \le t_{i-1}$ denotes that type s has to be identical to type t_{i-1} or a subtype thereof.

3 The Query Language and the Term Expressions

3.1 The Query Language

For our object model we developed a QUEL-like query language along the lines of the EXCESS object query language [3].

Let X_i be variables, T_i set typed expressions or type names (which represent the types' extension), and S a selection predicate. Then, a query has the following form:

range $X_1 : T_1, \ldots, X_n : T_n$
retrieve X_i
where S

Of course, the selection predicate S may itself contain a retrieve statement.

Example 3.1 The following example query will be used throughout the remainder of this paper to illustrate our optimization techniques. The query is based on the type definitions of Example 2.1 and has the following semantics: *"retrieve the managers of departments which generate losses and, at the same time, pay at least one of their employees an exorbitant salary exceeding 200K."*

range e : EMP, m : EMP
retrieve m
where m = e.WorksIn.Mgr **and**
e.Salary > 200000 **and**
e.WorksIn.Profit < 0

3.2 The Term Language

One of the main arguments for the term language used here is that every term corresponds to a query evaluation plan. The second argument is the simplicity of the first translation step of translating the user's query language into the term representation. Of course, this step may be more complicated for other query languages than the one used in *GOM*. But then at least the independence of the term language from the query language guarantees that only the preprocessing phase of the query optimizer has to be redesigned.

The first "high level" operator of the term language is the **retrieve** operator with the following parameters:

(**retrieve :B** BINDING **:S** SELPRED **:P** PROJ)

It represents a nested loop evaluation of the query specified by the parameters. The variables are bound from left to right to every possible value of the corresponding set in the **:B** clause which consists of pairs of range variable and type names or set valued expressions. On each binding the selection predicate following the label **:S** is evaluated, and in the case of success the binding of the variable corresponding to the one in the **:P** clause is gathered. Of course, different permutations of the pairs in the **:B** clause show different performance. But since this problem has already been excessively treated elsewhere we do not concern ourselves herewith.

The "low level" operators which are utilized in the optimization in order to increase performance by accessing the access support relations are:

1. (**getasr** ASR **:R** RESTR **:S** SELPRED **:P** PROJ) This operator retrieves tuples (projected onto the attributes in the *PROJ* list) from an access support

relation *ASR*, for which *RESTR* $\wedge$ *SELPRED* is satisfied. The **:R** clause is used to give explicit entries into the B$^+$ tree used to guarantee fast access to the tuples in the access support relations. Thus, the *RESTR* predicate can only refer to attributes at the left and/or right of an access support relation partition.

2. (**mkasr** TYPE PATH :S SELPRED :P PROJ)
This operator materializes a new temporary access support relation

3. (**appendasr** ASR PATH :S SELPRED :P PROJ)
The **appendasr** operator is utilized to extend an existing access support relation beyond the originally defined attribute chain

Each of the three latter operators returns an internal main memory representation of an access support relation, ASR. Besides those new operators dealing with access support relations there exist some useful operators from relational algebra, e.g., join, union, etc.

Terms as used for the selection predicate, *SELPRED* and *RESTR*, are of the form (**op** $t1$ $t2$) where $t1$ and $t2$ are constants, variables, or path expressions of the form (**path** v $A_1 \ldots A_n$) for a variable v and attributes A_i, and **op** is a comparator. A selection predicate can be built from terms using the usual boolean connectors. In a preprocessing step negations are eliminated in the usual way using de Morgan's law and reverting the comparators.

3.3 Translation of Queries into Term Representation

The initial translation of a user query into a term is straight forward. The **range** clause is translated into a binding list, marked **:B**, the **retrieve** clause into a projection list **:P**, and the **where** clause into a selection predicate prefixed with **:S**.

Example 3.2 To make things more concrete we give the translation of the example query into terms:

```
(retrieve :B ((e EMP) (m EMP))
          :S (and (= m (path e WorksIn Mgr))
                  (< (path e WorksIn Profit) 0)
                  (> (path e Salary) 200000))
          :P m)
```

This not yet optimized term expression yields a very simplistic evaluation: the nested loop evaluation. The strategy is to convert the terms of the binding list into nested loops and for each binding of the range variables separately evaluate the :S clause.

4 Transformation Rules to Optimize Term Representation

The query optimization steps are described as transformation rules or rewriting rules [7]. A rule is given in the form

$$l \longrightarrow r$$

which specifies that expression l is replaced by expression r. The expressions l and r themselves may contain meta-variables standing for a term or a list of terms which are denoted by a prefix "!" or "!!", respectively.

Example 4.1 The next transformation rule demonstrates the use of meta-variables.

(**and** (!!l0 *true* !!l1)) $\longrightarrow$ (**and** (!!l0 !!l1))

This transformation rule denotes that a constant *true* can be removed from a list of conjuncts. □

Further meta variables are as follows:

- e, f, g, v, e_0, e_1, etc. denote range variables
- A_1, A_2, ..., B_1, B_2, ..., D_1, ...denote attribute names
- Φ, Ψ denote comparison operators, e.g., $=$, **in**, $<$, etc.
- c denotes a constant, i.e., an atomic value or an object, and C denotes a constant set of objects or values.

In the remainder of this section we represent the main rule groups used in our implementation of the query optimizer. For each group we choose one representative member which, whenever possible, is illustrated by application to our running example. The optimization is separated into three main phases. The first is a preprocessing phase introduced in the next subsection. In the main optimization phase the different rules are applied (subsequent subsections) which is followed by the polishing phase as described in the last subsection of this section.

4.1 Preprocessing and Preliminaries

First the negations are eliminated. Further there exist two sets of rules which serve to simplify expressions. One is for the simplification of Boolean expressions, the other serves to simplify set expressions. These rule groups stand somewhat outside the regular rule system and are applied whenever necessary (cf. Section 5).

For commutative operators the possible transformation rules for rearranging predicates are built-into the rules whenever they are useful. This is needed to arrange terms in the order that is required to match the left-hand side of the transformation rules.

4.2 Prolonging Path Expressions

In order to utilize an existing access support relation $[\![t_0.A_1.\cdots.A_n]\!]_X$ to evaluate a query it may be necessary to first prolong the path expressions contained in the **:S** clause. This may be essential to make the access support relation *applicable* (cf. Definition 2.6)—depending on the extension X of the respective ASR.

4.2.1 Prolonging a Linear Path Expression

Let T be a **retrieve** term in which the **:S** clause contains a linear path expression of the following form:

T ≡ (**retrieve** **:B** ((e !b) !!bl)
  :S (**and** (= e (**path** v $A_i \ldots A_j$))
    !!sl)
  :P !p)

Then the following transformation can be applied throughout T, not affecting nested retrieves where e is not free:

(**path** e $A_{j+1} \ldots A_l$) $\longrightarrow$ (**path** v $A_i \ldots A_j\ A_{j+1} \ldots A_l$) [T1]

A further simplification is possible if—after the transformation—the range variable e is not further qualified in T. In this case (e !b) may be dropped from the binding clause and the term "(= e (**path** v $A_i \ldots A_j$))" can be dropped from the **:S** clause, which is formalized in the following rule:

(**retrieve** **:B** ((e !b) !!bl)
  :S (**and** (= e (**path** v $A_i \ldots A_j$))
    !!sl')
  :P !p)
$\longrightarrow$
(**retrieve** **:B** (!!bl)
  :S (**and** !!sl')
  :P !p) [T2]

The rule may only be applied if e does not occur free in !!sl' and !p.

4.2.2 Prolonging a Set-Valued Path Expression

The formulation of the rules for prolonging a set-valued require some care in order to guarantee that the transformation yields a semantically equivalent term. Let us illustrate the intrinsic problem on the following example:

(**retrieve** **:B** ((e EMP) (c CAR))
  :S (**and** (**in** c (**path** e Cars))
    (= 'Jaguar' (**path** c Make))
    (= 150 (**path** c HorsePower)))
  :P e)
$\not\equiv$
(**retrieve** **:B** ((e EMP))
  :S (**and** (**in** 'Jaguar' (**path** e Cars Make))
    (**in** 150 (**path** e Cars HorsePower)))
  :P e)

The left-hand **retrieve** term finds all *EMP*loyees who own a 'Jaguar' with 150 *HorsePower*. The right-hand term, however, retrieves the *EMP*loyees who own one *CAR* made by 'Jaguar' and one *CAR* (the same one or another one) that has 150 *HorsePower*.

Therefore, the rule T1 for prolonging has to be restricted for set-valued path expressions because only special cases guarantee semantic equivalence after prolonging a path expression involving a set-valued attribute. For example, a prolonging is—at least—possible if the intermediate range variable is qualified only once in the **:S** clause:

(**retrieve** **:B** ((e !b) !!bl)
  :S (**and** (**in** e (**path** v $A_i \ldots A_j$))
    (**in** !s (**path** e $A_{j+1} \ldots A_l$))
    !!sl)
  :P !p)
$\longrightarrow$ [T3]
(**retrieve** **:B** (!!bl)
  :S (**and** (**in** !s (**path** v $A_i \ldots A_l$))
    !!sl)
  :P !p)

The rule may be applied if e does not occur free in !!sl, !s, and !p.

There are other rules which allow prolonging under the following conditions:

- the intermediate (connecting) range variable e is further qualified only in a disjunction, i.e., in a term of the form (**or** !ol_1 !ol_2 ...). In this case e may be eliminated even if it occurs in more than one disjunct !ol_i.
- the term that qualifies the range variable, i.e., "(**path** v $A_i \ldots A_j$)" is linear. For this case an analogous rule to T1 can be formulated.

4.3 Splitting Path Expressions

Splitting of path expressions may be needed to utilize an existing access support relation $[\![t_0.A_1.\cdots.A_n]\!]$. We will provide the rule for linear paths only—an analogous rule exists for set-valued path expressions.

Let g be a new variable not occurring in the binding list of the enclosing retrieve expression.

(= !s (**path** v $A_i \ldots A_j\ B_l \ldots B_k$)) $\longrightarrow$ [T4]
(**and** (= g (**path** v $A_i \ldots A_j$))
  (= !s (**path** g $B_l \ldots B_k$)))

We have to add $(g\ t_j)$ to the binding list of the directly enclosing retrieve. Then g may be substituted for any other path prefix "$v\ A_i \ldots A_j$", yielding to the following transformation rule:

```
(and (= g (path v Ai...Aj))
     (Φ !s (path v Ai...Aj Dr...Dq)))   ⟶        [T5]

                  (and (= g (path v Ai...Aj))
                       (Φ !s (path g Dr...Dq)))
```

The combination of T4 (splitting) and T5 (substitution of path prefix) can be used to factor out common path prefixes in order to avoid multiple reference traversal along the same reference chain. This is built-into the search heuristics of the term rewriting system (cf. Section 5).

Example 4.2 Consider again our running example. In the remainder of this section we will transform this example step by step under the assumption that the following two access support relations exist: $[\![\text{EMP.Salary}]\!]_{can}$ and $[\![\text{EMP.WorksIn.Mgr}]\!]_{can}$.

```
(retrieve :B ((e EMP) (m EMP))
    :S (and (= m (path e WorksIn Mgr))       T4,T5
            (< (path e WorksIn Profit) 0)    ⟶
            (> (path e Salary) 200000))
    :P m)

          (retrieve :B ((e EMP) (m EMP) (d DEPT))
              :S (and (= d (path e WorksIn))
                      (= m (path d Mgr))
                      (< (path d Profit) 0)
                      (> (path e Salary) 200000))
              :P m)
```

Note, that this transformation actually results in a less efficient **retrieve** term. This "step backwards", however, is only performed by the optimizer if it leads to a subsequent transformation step that will utilize an access support relation which vastly optimizes the evaluation.

4.4 Utilization of ASRs for Single-Target Path Expressions

A selection predicate based on a path expression for which an applicable access support relation exists should be transformed into an equivalent operation on the access support relation.

Let c be a constant (object or value), then we can substitute

```
                          (in e (getasr [t0.A1.···.An]X
(= c (path e Ai...Aj))⟶            :R true                  [T6]
                                   :S (= c #j)
                                   :P #(i − 1)))
```

if $Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_i.\cdots.A_j)$ is satisfied for $s = type(e)$.

Attributes of the access support relations are referenced by their position, e.g., $\#j$ references the $j+1^{nth}$ attribute (the first attribute is denoted $\#0$).

There are similar rules for the **in** predicate, i.e., set-valued path expressions. Note, that we then have to distinguish the two cases

(**in** c (**path** e $A_i \ldots A_j$)) and (**in** (**path** e $A_i \ldots A_j$) c).

Example 4.3 Application of the above rule yields for our running example:

```
(retrieve :B ((e EMP) (m EMP) (d DEPT))
    :S (and (= d (path e WorksIn))
            (= m (path d Mgr))               T6
            (< (path d Profit) 0)            ⟶
            (> (path e Salary) 200000))
    :P m)

          (retrieve :B ((e EMP) (m EMP) (d DEPT))
              :S (and (= d (path e WorksIn))
                      (= m (path d Mgr))
                      (< (path d Profit) 0)
                      (in e (getasr [EMP.SALARY]can
                                 :R true :S (> #1 200000)
                                 :P #0)))
              :P m)
```

Note, that there is no ASR to evaluate the path expression (**path** *d Profit*).

4.5 Multi-Target Expressions

So far, we have utilized access support relations only for path expressions that are involved in a comparison predicate with a constant (c). Let us now consider comparisons with range variables (or even with other path expressions).

4.5.1 Bi-Connected Expressions

A two-target expression based on a path expression has the form (**in** e (**path** $v\ A_i \ldots A_j$)), where e and v are both range variables. The subsequent rule T7 should only be applied if the path cannot be prolonged to a predicate involving only one range variable and a constant (cf. Section 4.2).

```
                          (in (v e) (getasr [t0.A1.···.An]X
(in e (path v Ai...Aj))⟶             :R true                 [T7]
                                     :S true
                                     :P (#(i − 1) #j)))
```

Again, the application of this rule requires that $Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_i.\cdots.A_j)$ is satisfied for $s = type(v)$. The "e" could be generalized to a path expression originating in a range variable.

4.5.2 Multipily-Connected Paths

The rule T7 can be generalized to multipily-connected path expressions. Using the simplification rules cited in Section 4.1 the terms of the conjunction are first arranged in the desired order. Note that further references to the range variables e_i for $(1 \leq i \leq k)$ may obstacle the prolonging of the interconnected paths on the left-hand side of the rule T8.

```
(and (Φ₁ e₁ (path e₀ A_{i₀+1} ... A_{i₁}))
     (Φ₂ e₂ (path e₁ A_{i₁+1} ... A_{i₂}))
     ...                                      ⟶        [T8]
     (Φ_k e_k (path e_{k-1} A_{i_{k-1}+1} ... A_{i_k}))
     !!sl)

          (and (in (e₀ e₁ ... e_k)
                   (getasr ⟦t₀.A₁.···.A_n⟧_X
                           :R true :S true
                           :P (#i₀ #i₁ ... #i_k))))
               !!sl)
```

Application of T8 requires that the predicate $Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_{i_0+1}.\cdots.A_{i_k})$ is satisfied for $s = type(e_0)$. Here, the meta symbols $\Phi_1, \ldots, \Phi_k$ stand for = or **in**.

Example 4.4 The rule T8 can be applied to our running example:

```
(retrieve :B ((e EMP) (m EMP) (d DEPT))
   :S (and (= d (path e WorksIn))
           (= m (path d Mgr))
           (< (path d Profit) 0)               T8
           (in e (getasr ⟦EMP.SALARY⟧_can       ⟶
                  :R true :S (> #1 200000)
                  :P #0)))
   :P m)

     (retrieve :B ((e EMP) (m EMP) (d DEPT))
        :S (and (in (e d m)
                    (getasr ⟦EMP.WorksIn.Mgr⟧_can
                       :R true :S true
                       :P (#0 #1 #2)))
                (< (path d Profit) 0)
                (in e (getasr ⟦EMP.SALARY⟧_can
                       :R true :S (> #1 200000)
                       :P #0)))
        :P m)
```

4.6 Further Operators on Access Support Relations

4.6.1 Creating Temporary Access Support Relations

If there exists a path for which no access support relation is given one may introduce a temporary access support relation using the operator **mkasr**. The rules of the previous sections find their analogous counterpart.

4.6.2 Appending Access Support Relations

If there exists a path for which at least some part—in practice, the major or most selective part—is covered by an access support relation we may temporarily extend this access support relation using the **appendasr** operator.

```
(= c (path e A_i ... A_j D₁ ... D_r))  ⟶                [T9]

        (in e (appendasr (getasr⟦t₀.A₁.···.A_n⟧
                                 :R true :S true
                                 :P (#(i-1)...#j))
                   (path #j D₁ ... D_r)
                   :S (= c #(r+j-i))
                   :P #0))
```

Again, the transformation is only valid if $Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_i.\cdots.A_j)$ is satisfied for $s = type(e)$.

4.6.3 Joining Access Support Relations

A predicate based on the comparison of two path expressions which both have an applicable access support relations may be transformed into the join of the two access support relations:

```
(= (path e A_i ... A_j) (path f B_l ... B_k))  ⟶       [T10]

        (in (e f) (join (getasr ⟦t₀.A₁.···.A_n⟧
                                :R true :S true :P all)
                        (getasr ⟦s₀.B₁.···.B_m⟧
                                :R true :S true :P all)
                        :J (= #j #k)
                        :S (true)
                        :P (#(i-1) #(n+1))))
```

This transformation requires the satisfaction of $Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_i.\cdots.A_j)$ for $s = type(e)$ and $Applicable([\![s_0.B_1.\cdots.B_m]\!]_X, r.B_l.\cdots.B_k)$ for $r = type(f)$. The **:J** denotes the join predicate.

Since the join may be a very costly operation one should try every other possibility before committing this transformation T10. If the enclosing retrieve term contains a selective binding for e and f, e.g.,

```
:B ((e C₁) (f C₂) !!bl)
```

then these should be propagated into the :S clauses of the respective **getasr** term in order to minimize the number of joined tuples.

There is a similar rule for the comparison operator **in**, i.e., the first path being linear and the second path set-valued.

4.7 Introduction of Union

If nothing else works a disjunctive selection predicate may be evaluated separately, with the possibility of first transforming the predicate into disjunctive normal form.

```
(retrieve :B !!bl
          :S (or !s1 ... !sn)   ⟶                [T11]
          :P !p)

                    (union (retrieve :B !!bl :S !s1 :P!p)
                           ...
                           (retrieve :B !!bl :S !sn :P !p))
```

Of course, there exist more rules for disjunctions at a deeper level of nesting.

4.8 Moving Selection Predicates Inwards

For 'C' being any set valued term, e.g., a term with outer operator **getasr**, the following rule can be applied to move selection predicates inwards.

```
(and ((in ($e_0$ ... $e_l$ ... $e_k$)
          (getasr [[$t_0.A_1.\cdots.A_n$]]$_X$
              :R !r
              :S !s                      ⟶          [T12]
              :P (#$i_0$ ... #$i_l$ ... #$i_k$))))
     (in $e_l$ C)
     !!sl)

          (and ((in ($e_0$ ... $e_l$ ... $e_k$)
                    (getasr [[$t_0.A_1.\cdots.A_n$]]$_X$
                        :R !r
                        :S (and !s (in #$i_l$ C))
                        :P (#$i_0$ ... #$i_l$ ... #$i_k$))))
               !!sl)
```

If the predicate propagated into the **getasr** term constituted the last reference to e_l within the enclosing **retrieve** term we may also delete e_l from the **in** list and concurrently, the projection on column $\#i_l$ has to be removed from the **:P** clause.

```
(and ((in ($e_0$ ... $e_l$ ... $e_k$)
          (getasr [[$t_0.A_1.\cdots.A_n$]]$_X$
              :R !r
              :S !s                      ⟶          [T13]
              :P (#$i_0$ ... #$i_l$ ... #$i_k$))))
     !!sl)

   (and ((in ($e_0$ ... $e_{l-1}$ $e_{l+1}$ ... $e_k$)
             (getasr [[$t_0.A_1.\cdots.A_n$]]$_X$
                 :R !r
                 :S !s
                 :P (#$i_0$ ... #$i_{l-1}$ #$i_{l+1}$ ... #$i_k$))))
        !!sl)
```

This transformation is valid if the enclosing **retrieve** term (including the shown term list !!*sl*) does not contain a free reference to e_l. Furthermore, the e_l should be removed from the binding list of the enclosing **retrieve** term.

Analogous rules exist for the other operations like **mkasr**, **appendasr**, **join**, and for deeper levels of nesting. Further, there exist rules to move selections into such operator expressions which are already moved to the binding list (cf. Section 4.9).

Example 4.5 Consider the following transformation steps which illustrate the full use of T12 in combination with T13.

```
(retrieve :B ((e EMP) (m EMP) (d DEPT))
  :S (and (in (e d m) (getasr [[EMP.WorksIn.Mgr]]$_{can}$
                               :R $true$ :S $true$
                               :P (#0 #1 #2)))
          (< (path d Profit) 0)
          (in e (getasr [[EMP.SALARY]]$_{can}$
                        :R $true$ :S (> #1 200000)
                        :P #0)))
  :P m)

                          $\xrightarrow{\text{T12,T13}}$

(retrieve :B ((m EMP) (d DEPT))
  :S (and (in (d m)
              (getasr [[EMP.WorksIn.Mgr]]$_{can}$
                 :R $true$
                 :S (in #0 (getasr [[EMP.SALARY]]$_{can}$
                                :R $true$
                                :S (> #1 200000)
                                :P #0)))
                 :P (#1 #2))
          (< (path d Profit) 0))
  :P m)
```

Note that the optimization includes the removal of the range variable e (by application of T13 from the binding and from the projection list of the **getasr** term because e is no longer referenced.

4.9 Moving Predicates into the Binding List

A predicate that evaluates to a constant, e.g., a predicate that is based on the evaluation of a **getasr** expression should be moved into the binding list of the enclosing **retrieve** term. This avoids the nested loop evaluation by iterating exhaustively over all elements of the specified types. The following general transformation can be applied:

```
(retrieve :B ((e_0 S_0)···(e_k S_k) !!bl)
     :S (and (in (e_1 ... e_k)
                 (getasr [[t_0.A_1.···.A_n]]_X
                     :R !r
                     :S !s                      ⟶      [T14]
                     :P (#i_0 ... #i_k)))
             !sl)
     :P !p)

            (retrieve :B (((e_0 ... e_k)
                           (getasr [[t_0.A_1.···.A_n]]_X
                               :R !r
                               :S (and (in #i_0 S_0)
                                       ...
                                       (in #i_k S_k)
                                       !s)
                               :P (#i_0 ... #i_k)))
                          !!bl)
                 :S (and !sl)
                 :P !p
```

Example 4.6 The following transformation concludes the optimization of our example query:

```
(retrieve :B ((m EMP) (d DEPT))
    :S (and (in (d m)
                (getasr [EMP.WorksIn.Mgr]_can
                    :R true
                    :S (in #0 (getasr [[EMP.SALARY]]_can
                                  :R true
                                  :S (> #1 200000)
                                  :P #0)))
                    :P (#1 #2))
            (< (path d Profit) 0))
    :P m)                  ─T14→

  (retrieve
     :B (((d m)
          (getasr [EMP.WorksIn.Mgr]_can
              :R true
              :S (and (in #0 (getasr [[EMP.SALARY]]_can
                                 :R true
                                 :S (> #1 200000)
                                 :P #0))
                      (in #1 EMP)
                      (in #2 DEPT))
              :P (#1 #2))))
     :S (and (< (path d Profit) 0))
     :P m)
```

4.10 Introduction of Restriction Predicates

The following rule introduces restriction predicates. It can only be applied once since there is only one restriction term allowed. Thus this operation is left to the end of the term rewriting to choose the most selective term for restriction. This rule can be applied if the term !s1 concerns only the entry attribute of the **getasr** operation (cf. Section 3.2 for the semantics of the **:R** clause):

```
(getasr [[t_0.A_1.···.A_n]]      (getasr [[t_0.A_1.···.A_n]]
   :R true                          :R !s1
   :S (and !s1 !!sl1)    ⟶         :S (and !!sl1)        [T15]
   :P !p1)                          :P !p1)
```

4.11 Deletion of the Retrieve Operator

If the selection predicate is empty and only one variable is left in the binding list then we may remove the outer retrieve. More formally the following rule is valid:

```
(retrieve :B (e !t) :S true :P e)  ⟶  !t        [T16]
```

4.12 Polishing of Resulting Terms

4.12.1 Dealing with Access Support Relation Partitions

So far we have only considered access support relations under no decomposition, i.e., $[\![t_0.A_1.\cdots.A_n]\!]_X$. According to our convention this should have been denoted more precisely as $[\![t_0.A_1.\cdots.A_n]\!]_X^{0,n}$. Introduction of access support relation partitions is now straight forward. This is the first step of the polishing phase.

4.12.2 Isolating Common Subexpressions

The second step in the post-transformation phase consists of finding common subterms—analogously to [4]—to avoid evaluating them twice. This is especially important if some access support relation partitions are shared by several access support relations.

5 The Rule Interpreter

In this section we introduce the governing strategies and mechanisms utilized in our query optimizer. This is a very important issue since if the rules were applied in an unordered and exhaustive manner there would be the problem of exponential explosion of the search space. Thus guidance is needed to govern the deductive process of term rewriting. We have developed a number of technics to solve this problem.

The basis of optimizing the rewriting process consists of organizing the rules into groups of rules with similar intention.

For each group of rules a mode of application can be given. This mode is either *all* or *single*. If *all* is defined, all rules of this mode are applied until no further rule of this group is applicable. An example of a rule group where *all* is specified as the application mode is the moving selection predicates inwards group. If *single* is specified there will be at most one successful rule application of a member of the respective group every time the rule group is visited by a term. Since there are rules which may be better in some sense, the rules within a rule group may be ordered.

The *successor group* from which the next rules are to be applied are declared by defining the *rule group net*. This net of rule groups is described by giving a successor group for each rule group for the case that at least one rule applied successfully and—a different one—for the case that all rules failed to match. To give an example, if there is a successful application of a **getasr** introducing rule, the successor is the rule group of moving selection predicates inwards. If there is no further possibility to introduce a **getasr** operation the successor group is the one trying to move set valued terms into the binding list whenever possible.

Further there are two special rule groups for simplifying expressions one for simplifying Boolean expressions and one for simplifying set expressions. Since application of these rule groups may not interfere with the order in which the other rule groups are applied, they are invoked only if necessary and without change to the successor rule group. This is described by specifying *simbool* and/or *simset* in the rule group net for each rule group where the corresponding set of simplifications might be worthwhile to attempt.

Sometimes it does make sense not to obey the default successor rule group given by the net. Instead one might want to choose the application of a different rule group, or even the application of a certain rule. As an example consider the rule group *prolonging*. The standard successor group in the case of successful application of a rule is the introduction of a **getasr** operator. But if the prolonging has been beyond what is covered by an access support relation successive splitting is reasonable. Thus, with every rule there may be a successor rule group associated or a successor rule. This avoids many useless tests for possible rule applications. The applied successor rule may also depend on the history of the term considered.

The first choice for a strategy to process a query term is, of course, to first prolong and then split the path expressions in such a way that the existing access support relations become applicable. Then to introduce the **getasr** operations, move the selection predicates inwards, then move the **getasr** operations to the binding list and remove the **retrieve**. If this fails a strategy where new access support relations are temporarily created (**mkasr**) or appended (**appendasr**) is followed. The application of joins is delayed to a point where all other strategies failed. Since every strategy demands a different rule group net, there exists one corresponding net for each strategy. With each term the current strategy is associated. The strategy is changed if there is no more successful rule application within the considered rewriting mode. The successor strategy may depend on the structure of the term and on its history.

We now come to the management of terms which is highly interconnected with rule processing. At the beginning of the optimization process there is only one term. This term is put into the list of active terms. After a successful rule application the result replaces the only term in the active terms list. This is the default for most of the rules. If alternatives have to be considered—as in the case of the application of **appendasr** or **mkasr** rules—the result term of a rule application does not replace the original term but is (by default) added to the beginning of the list of active terms. This results in a depth first search. Other search strategies can be specified as well. This is necessary if the optimization is stopped by some criterion before all terms are optimized to the normal form where no further rule application is possible. If there is a change in the rule application strategy, the term is saved in the list of optimized terms before starting a new optimization phase. The last step of processing is done by polishing the resulting terms, e.g., taking care of access support relation partitions, eliminating common subexpressions within a query term, etc. If the resulting list of optimized terms contains more than one term the cost model whose basis was developed [11]) will be applied (not yet integrated) and the terms will be ordered accordingly. The cheapest term is then chosen according to the recorded database characteristics and translated into an executable query evaluation plan.

6 Conclusion

In this paper we have shown how access support relations can be utilized in query evaluation against object bases. The access support manager which controls and maintains the access support relations has been implemented in C and runs on a DEC station 3100 under Ultrix. We described the essential parts—consisting of 16 rules, each a representative of a larger rule group—of a rule-based query optimizer. The complete query optimizer was realized in Lisp and consists of a core of about 80 rules dealing with access support relations—aside from the trivial simplification rules.

Utilizing the rule-based approach we were able to realize the prototype with relatively modest effort. The rule-based design is particularly amenable to

- incorporating new rules due to revised evaluation strategies or new indexing structures
- researching different search heuristics to find a near-optimal evaluation plan without exhaustive search.

The performance of the query optimizer is—in the current prototype version—not really sufficient for a production quality system. It took, for example, about a second to transform the (simple) example query. However, for experimentation and evaluation purposes the performance is quite sufficient. In order to gain performance the term rewriting rules may be converted to C transformation routines.

Currently we are incorporating the cost model that was developed in [11] for evaluating the usefulness of access support relations into the query optimizer. This would enable the quantitative comparison of alternative transformations based on the current object base extension, i.e., number of objects, size of access support relations, selectivity of restriction predicates, etc.

In summary, we showed that access support relations as an indexing scheme in conjunction with rule-based query optimization provide a very promising road to performance enhancement of query processing in object bases.

Acknowledgements

We thank P. C. Lockemann for his support and our students U. Oetken, H. Ott, A. Papapostolou, K. Peithner, H. Spiess, and R. Waurig for their implementation efforts in realizing the access support relation manager and the query processor of *GOM*.

References

[1] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. Knowledge and Data Engineering*, 1(2):196–214, Jun 1989.

[2] M. Carey and D. J. DeWitt. An overview of the EXODUS project. *IEEE Database Engineering*, 10(2):47–53, Jun 1987.

[3] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 413–423, Chicago, Il., Jun 1988.

[4] U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 384-391, 1986.

[5] J. C. Freytag. A rule-based view of query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 173–180, San Francisco, 1987.

[6] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, 1987.

[7] G. Huet. Confluent reductions: Abstract properties and applications of term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[8] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, pages 111–152, Jun 1984.

[9] B. P. Jeng, D. Woelk, W. Kim, and W. L. Lee. Query processing in distributed ORION. In *Proc. of the EDBT (Extending Data Base Technology) Conf.*, Venice, Italy, Mar 1990.

[10] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Atlantic City, NJ, May 1990.

[11] Alfons Kemper and Guido Moerkotte. Access support in object bases—including an analytical cost model. Interner Bericht 17/89, Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Oct 1989.

[12] K. C. Kim, W. Kim, and D. Woelk. Acyclic query processing in object-oriented databases. In *Proc. of the Entity Relationship Conf.*, Italy, Nov 1988.

[13] K. Lehnert. *Regelbasierte Beschreibung von Optimierungsverfahren für relationale Datenbankanfragesprachen.* PhD thesis, Technische Universität München, 8000 München, West Germany, Dec 1988.

[14] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, 1988.

[15] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In K. R. Dittrich and U. Dayal, editors, *Proc. IEEE Intl. Workshop on Object-Oriented Database Systems, Asilomar, Pacific Grove, CA*, pages 171–182. IEEE Computer Society Press, Sep 1986.

[16] P. G. Selinger et al. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, Ma., Jun 1979.

[17] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, Jun 1987.