[22] S. T. Shenoy and Z. M. Ozsoyoglu. A system for semantic query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 181–195, San Francisco, May 87.

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

[24] S. Zdonik and D. Maier. Fundamentals of object-oriented databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, pages 1–32. Morgan-Kaufman Publ. Co., 89.

[9] 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.

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

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

[12] 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.

[13] A. Kemper and G. Moerkotte. Access Support Relations: an index structure for object bases. Technical Report 17/89, Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Oct 1989. Submitted for publication to: *Information Systems* (accepted subject to revision).

[14] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 364–374, Atlantic City, NJ, May 1990.

[15] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 290–301, Brisbane, Australia, Aug 1990.

[16] 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.

[17] M. K. Lee, J. C. Freytag, and G. M. Lohman. Implementing an optimizer for functional rules in a query optimizer. Technical Report RJ 6125, IBM Almaden Research Center, San Jose, CA, 1988.

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

[19] 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.

[20] 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.

[21] 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.

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.

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.

Acknowledgement

P. C. Lockemann's continuous support of our research is gratefully acknowledged. Further, we would like to express our thanks to our students W. Häfelinger, A. Horder, U. Oetken, H. Ott, A. Papapostolou, K. Peithner, A. Saad, H. Spies, M. Steinbrunn, R. Waurig, and A. Zachmann for their help in "getting the prototype running." Especially K. Peithner (query evaluator), H. Spies (pattern matcher), and R. Waurig (transformation rules and search heuristics) worked on the realization of the GOM modules that were described in this paper. Our colleagues—and most recent GOM project members—C. Kilger and H.-D. Walter carefully read a preliminary draft of this paper.

References

[1] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. of the DOOD Conference*, pages 40–57, Kyoto, Japan, Dec 1989.

[2] D. S. Batory, T. Y. Leung, and T. E. Wise. Implementation concepts for an extensible data model and data language. *ACM Trans. Database Syst.*, 13(3):231–262, Sep 1988.

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

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

[5] 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.

[6] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, Jun 1990.

[7] J. Duhl and C. Damon. A performance comparison of object and relational databases using the Sun benchmark. In *Proc. of the ACM Conf on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 153–163, San Diego, Ca., Sep. 1988.

[8] 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.

evaluation this is only true for the "inner" variables. In the above example a boolean variable for the *greater* node would be generated to hold the result of the evaluation of the corresponding subgraph. This value has to be recomputed only if the variable e changes its value.

Further we adhere to principles of *dependent* and *late binding*. The first concept—dependent binding—is realized whenever a restriction in the binding list of the term applies. This is the case if, e.g., an access support relation is applicable. Assume, for the above example, the existence of an access support relation $[\![\text{Emp.WorksIn.Mgr}]\!]_{can}$. Then, the optimizer could have generated the following term representation:

```
(retrieve  :B ((e EMP) (m (getasr [[Emp.WorksIn.Mgr]]_can
                                  :R true
                                  :S (=  #0 e)
                                  :P (#2))))
           :S (and  (>  (path e Salary) 200000)
                    (<  (path e WorksIn Profit) 0))
           :P m)
```

Then m would only be bound to the managers of the department the employee e works in. Note, however, that this term is not optimal. The binding of e could be performed by an additional projection on the last column of the access support relation. As we see most work to apply dependent binding is already done by the query optimizer when selection predicates are moved into the binding list.

The principle of *late binding* states that terms are only evaluated when there value is needed. This is especially useful if the selection predicate is an *and* term containing entries which "consult" different variables. In the above example we could evaluate both entries in the remaining selection predicate before binding m. This way m is not bound for employees not satisfying the selection predicate.

9 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 14 term rewriting 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 90 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

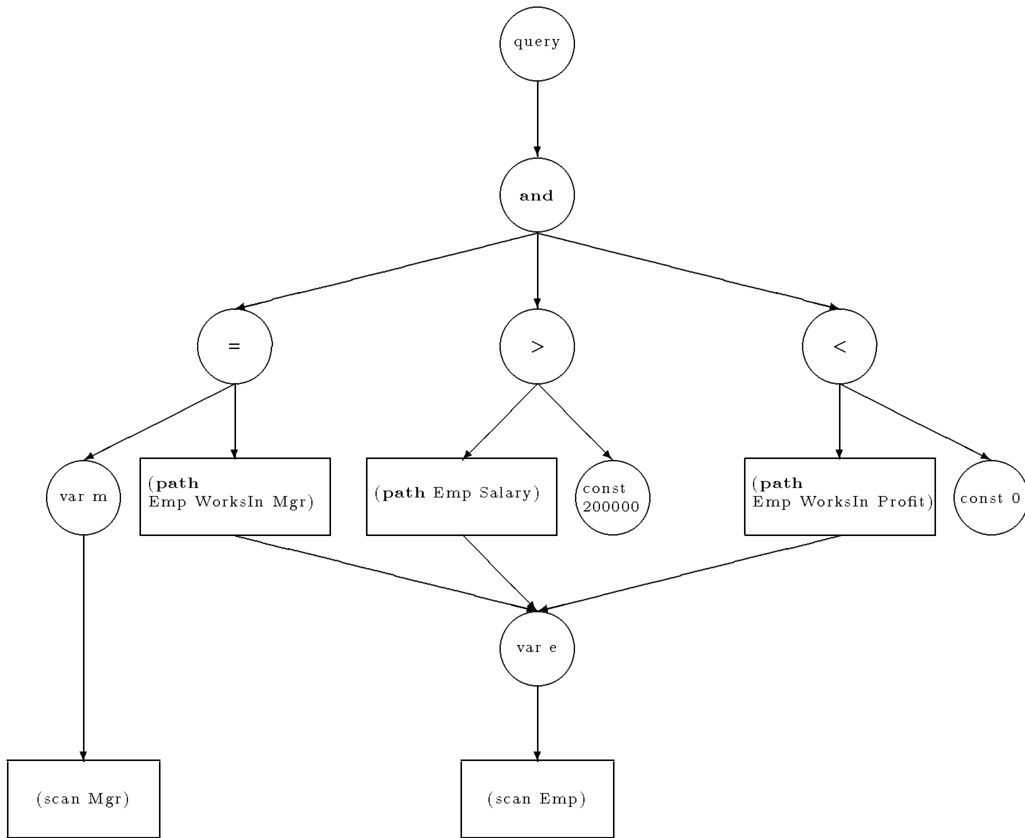

Figure 6: Query Evaluation Graph for the Running Example

join or semijoin depending on the projected columns and the selection predicate which (if possible) is distributed over the joins.

Since common subexpressions are mapped onto the same node within G this—in general—results in an acyclic graph and not in a tree. This is especially important if some access support relation partitions are shared by several access support relations.

We illustrate the translation process by the above example. The term[8]

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

is translated into the graph that is shown in Figure 6.

8.2 Translating the Graph Representation into Executable Code

In the last step this graph representation of the query is translated into executable code. For every node occurring in the graph a variable is declared to hold the result of the evaluation of the corresponding subgraph. A computed value then remains valid until at least one variable holding the value of a subnode changes. During the nested-loop

[8]For the purpose of this discussion we assume that optimizer was not able to transform the term.

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 which is controlled by the *environment manager*. 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 [14] will be applied 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.

8 Evaluating Optimized Terms

8.1 Translation of Terms into a Graph Representation

In order to perform a nested-loop evaluation the optimized terms are first translated into an acyclic directed graph $G = (\Gamma, <)$. This graph results from introducing one node for each subterm in the term to be translated. Thus there exist *query*- (including subqueries), *variable*-, *constant*-, *and*-, *asr*-, *join-nodes*, etc. The resulting nodes are then ordered by $<$ where $n_1 < n_2$ means that n_1 must be evaluated before n_2.

Special treatment is necessary for the *getasr* nodes. 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 straightforward. For each *getasr* applied to a decomposed access support relation the different parts are connected by a

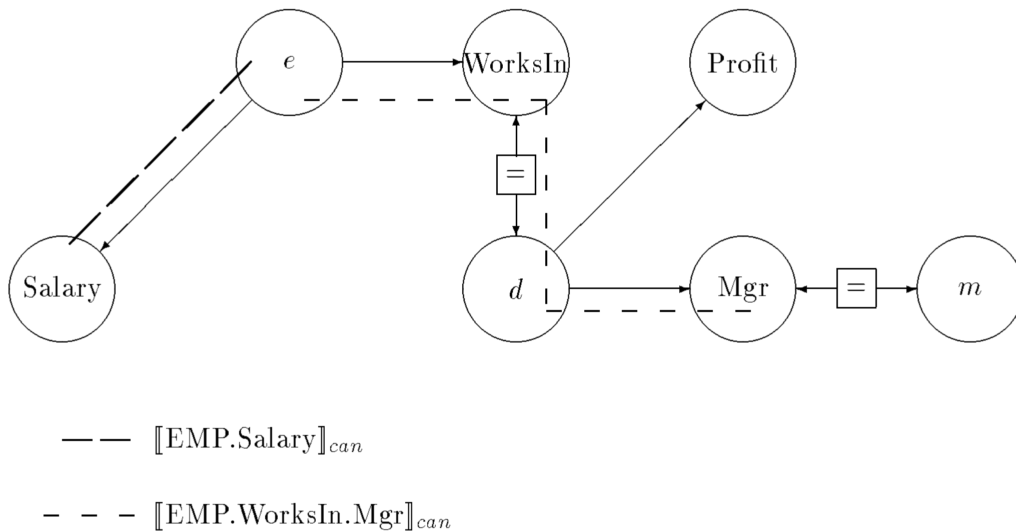

Figure 5: Access Support Relation Overlay

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 containing rules to move 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 simplifiying 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 subsequent splitting is reasonable. Thus, with every rule there may be

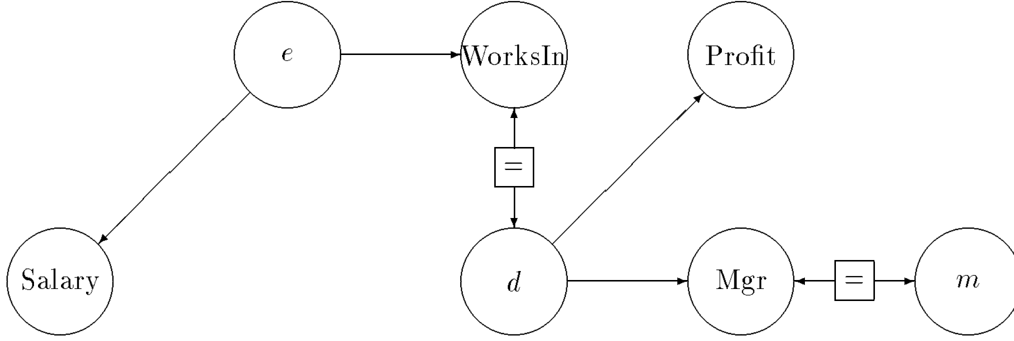

Figure 4: Graphical Representation of the Selection Predicate

Definition 7.1 (connection-path) *In this directed graph we call a sequence of nodes* $N_0, N_1, \ldots, N_{m-1}, N_m$ *a connection-path between nodes* N_0 *and* N_m *iff there exist directed edges between* N_i *and* N_{i+1} *for* $(0 \leq i < m)$. □

For each existing access support relation $[\![t_0.A_1.\cdots.A_n]\!]_x$ we then analyse whether it is possibly usable in the evaluation of the selection predicate. This depends on the extension X. We will outline the algorithm:

if $X = can$ **then**
find a range variable x of type $\leq t_0$—remember that this requires that x is bound to a set of elements of type t_0 or a subtype thereof—and connection-paths from x to A_1 and from A_i to A_{i-1} for $(1 \leq i < n)$
if $X = left$ **then**
find a range variable x of type $\leq t_0$ and find the greates $p \leq n$ such that connection-paths from x to A_1 and from A_i to A_{i-1} for $(1 \leq i < p)$ exist
if $X = right$ **then**
find the smallest q for $(0 \leq q < n)$ such that a range variable x of type $\leq t_q$ and connection-paths from x to A_{q+1} and from A_i to A_{i-1} for $(q < i < n)$ exist
if $X = full$ **then**
find the smallest q and the largest p for $(0 \leq q < p \leq n)$ such that a range variable x of type $\leq t_q$ and connection-paths from x to A_{q+1} and from A_i to A_{i-1} for $(q < i < p)$ exist

The above algorithm reflects the applicability definition of access support relations (cf. Definition 3.6).

Example 7.2 The analysis of possible access support relations is graphically shown in Figure 5.

In this case—since both ASRs are in *canonical* extension—we had to find a complete "overlay" of the access support relation path within the directed graph. This is indicated by the (differently) dashed lines. ◇

7.2 Rule Organization

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

7.1 Detection of "Usable" Access Support Relations

The foremost goal of the optimization heuristics is the effective utilization of existing access support relations. Thus, the selection predicate has to be analyzed with respect to the existing access support relations, the definition of which is obtained from the *schema manager*. For this analysis the selection predicate of a retrieve term is transformed into a directed graph structure within which the search for "introducible" ASRs can be performed efficiently.

In this analysis we will make two simplifying assumption:

- attributes defined in different types of the database schema are named differently. It is easy to relax this restriction—we would just have to maintain the types together with the attribute name in the subsequent graph representation.
- the selection predicate of the query term consists of a non-nested conjunction. Also this restriction has been relaxed in our actual implementation.

The directed graph representation of the selection predicate is defined as follows (note that this graph representation is only used to find "matching ASRs—it is not used for further optimization or evaluation steps):

1. for each range variable and for each attribute that occurs in some path expression a node is introduced which is labelled with the range variable or the attribute, respectively.
2. for each path expression (**path** x $A_i \ldots A_j$) introduce edges from (the node labelled) x to (the node labelled) A_i and from A_l to A_{l+1} for $i \leq l < j$.
3. for each term (Φ (**path** y $B_r \ldots B_q$) (**path** x $A_i \ldots A_j$))[7] introduce the following edges—depending on Φ
   - **if** $\Phi \in \{=, \textbf{seteq}\}$ **then**
     introduce an edge from B_q to A_j and from A_j to B_q
   - **if** $\Phi =$ **in then**
     introduce an edge from A_j to B_q

Example 7.1 Let us reconsider a term representation of our running example:

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

The graph representation of the selection clause is shown in Figure 4. For illustration purposes we labelled the edges that were introduced due to some operation Φ among two access support relations. ◇

[7] For simplicity, we assume that a term consisting of a single range variable x is represented as (**path** x).

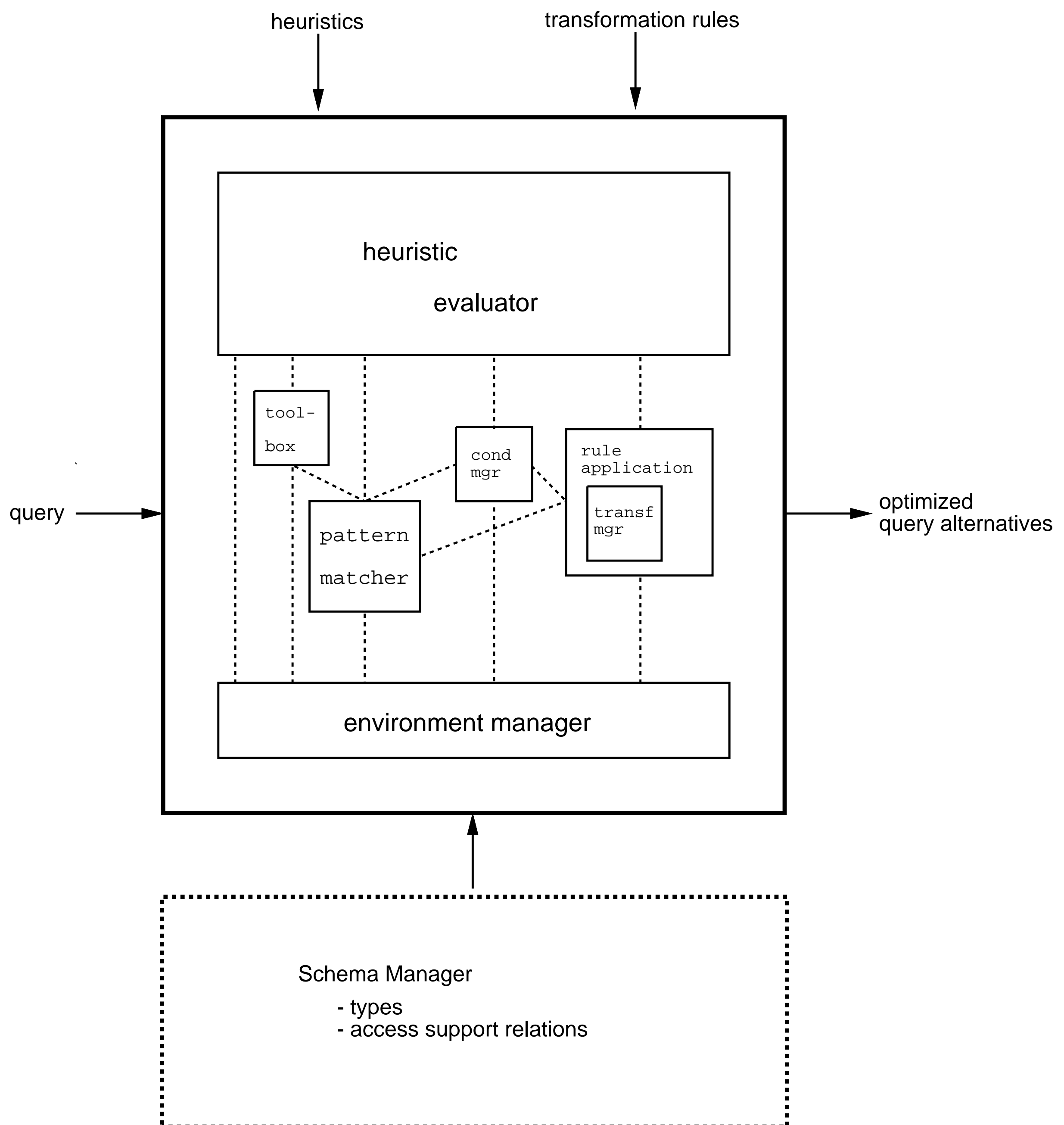

Figure 3: The Module Structure of the Rule Interpreter

IF

1. $Q/\overset{*}{r}0 \;=\;$ **retrieve**
2. $Q/\overset{*}{r}B \;=\; ((x)\; ASRexpr)$
3. $Q/\overset{*}{r}S \;=\; true$
4. $Q/\overset{*}{r}P \;=\; x$

THEN [T14]

1. $Q/\overset{*}{r} \longrightarrow ASRexpr$

---

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 architecture of the rule interpreter is outlined in Figure 3. The query optimizer was designed in a highly modular, and thus extensible fashion. Therefore, the heuristics as well as the transformation rules that constitute the essential ingredients of the optimizer, can be adapted and extended. This is indicated by the "*input-arrows*" associated with the two components.

The query optimizer contains a general *heuristic evaluator* which can interpret any heuristic that was built according to some specifications. This enables us to experiment with vastly different heuristics in order to analyze their performance. The *heuristic evaluator* governs the optimization process. It mainly interfaces with three modules:

- the *tool box*, which provides an extensive set of operations to analyse a term. For example, the algorithms that transform the selection predicate into a directed graph (cf. Section 7.1) which better supports the detection of "introducible" access support relations are part of the *tool box*.
- the *pattern matcher*, which is used to match patterns against the query term, and
- the *condition manager* which is used to evaluate whether the preconditions for applying certain transformation rules are satisfied.

In general, the optimization process has to be performed in the context of the existing database types and access support relation schemes, both of which are maintained by the *schema manager*.

The *environment manager* constitutes a set of global variables in which the positions of "interesting" patterns within the query term are maintained. Furthermore, the *environment manager* stores bookkeeping variables that are used in the process of *rule application*. Also, the list of alternative query terms is maintained by the *environment manager*.

The *rule application manager* controls the transformation rules and ensures that, after transforming the query term by the *transformation manager*, the necessary polishing steps are performed.

Because of space limitations we cannot provide a detailed description of the query optimization process. Rather, we pick a few interesting topics and discuss these in more detail: the detection of introducible access support relations and the organization of rules into rule groups.

IF

1. $Q/\overset{*}{r}0 \ = \ \mathbf{getasr}$
2. $Q/\overset{*}{r}30 \ \neq \ \mathbf{or}$
3. $Q/\overset{*}{r}3j \ = \ (\Phi \ \#i \ TERM)$
4. i is at the border of some partition of the ASR $Q/\overset{*}{r}1$

THEN [T13]

1. $Q/\overset{*}{r}2 \longrightarrow Q/\overset{*}{r}3j$
2. $Q/\overset{*}{r}3j \longrightarrow true$

Example 6.6 Using this rule we can now move the selection predicates of our example into the restriction clause.

```
(retrieve :B ((d m) (getasr [[EMP.WorksIn.Mgr]]_can
                      :R true
                      :S (in #0 (untuple (getasr [[EMP.SALARY]]_can
                                            :R true
                                            :S (> #1 200000)
                                            :P #0)))
                      :P (#1 #2)))
          :S (and (< (path d Profit) 0))
          :P m)
                                                          --T13-->
(retrieve :B ((d m) (getasr [[EMP.WorksIn.Mgr]]_can
                      :R (in #0 (untuple (getasr [[EMP.SALARY]]_can
                                            :R (> #1 200000)
                                            :S true
                                            :P #0)))
                      :S true
                      :P (#1 #2)))
          :S (and (< (path d Profit) 0))
          :P m)
```

◇

6.13 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 T14 is valid.

7 The Rule Interpreter and Search Heuristics

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

IF

1. $Q/\overset{*}{r}0 \;=\; \textbf{retrieve}$
2. $Q/\overset{*}{r}S0 \;\neq\; \textbf{or}$
3. $Q/\overset{*}{r}Si \;=\; (\textbf{in}\ (x_1 \ldots x_n)\ ASRexpr)$
4. $Q/\overset{*}{r}Bb_j \;=\; (x_j\ type_j) \qquad \text{for } 1 \leq j \leq n$

THEN [T12]

1. $append(Q/\overset{*}{r}B,\ ((x_1\ \ldots\ x_n)\ ASRexpr))$
2. $remove(Q/\overset{*}{r}Si)$
3. $remove(Q/\overset{*}{r}Bb_j) \qquad \text{for } 1 \leq j \leq n$

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

                                    :P (#1 #2)))
                  (< (path d Profit) 0))
          :P m)
 T12
--->
        (retrieve :B ((d m) (getasr [[EMP.WorksIn.Mgr]]_can
                                    :R true
                                    :S (in #0 (untuple (getasr [[EMP.SALARY]]_can
                                                               :R true
                                                               :S (> #1 200000)
                                                               :P #0)))

                                    :P (#1 #2)))
                  :S (and (< (path d Profit) 0))
                  :P m)
```

◇

6.12 Introduction of Restriction Predicates

The following rule group 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 T13 can be applied if the term concerns only the entry attribute of the **getasr** operation (cf. section 5.1 for the semantics of the **:R** clause). Exploiting the restriction predicates for efficiently evaluating an access support relation operation on the basis of the (redundant B^+ tree) storage model of the ASR partitions is the task of the *ASR-Manager*—and not further elaborated here.

```
(retrieve :B ([(e EMP)] (m MANAGER) (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] (untuple (getasr ⟦EMP.SALARY⟧can
                                            :R true
                                            :S (> #1 200000)
                                            :P #0))))]
          :P m)
```

$\xrightarrow{T10,T11}$

```
(retrieve :B ((m MANAGER) (d DEPT))
          :S (and (in (d m) (getasr ⟦EMP.WorksIn.Mgr⟧can
                                   :R true
                                   :S [(in #0 (untuple (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 T11) from the binding and from the projection list of the **getasr** term because e is no longer referenced within the selection predicate. ◇

6.11 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 general transformation rule T12 can be applied for this purpose. The meta-variable *ASRexpr* matches any kind of access support relation operation, e.g., a **getasr** or **joinasr** expression.

Example 6.5 The following transformation that utilizes rule T12 almost concludes the optimization of our example query:

In the formulation of T10 we used two metavariables: *RESTR* matching any restriction predicate and *ASRselect* which matches a selection predicate within the **getasr** expression.

6.10 Removal of Range Variables

There are several transformations after which a range variable of the query term may become obsolete. Examples are the path substitution rules (T1 and T2) and the preceding rule T10 for propagating selection predicates into the **getasr** expression. If the predicate propagated into the **getasr** term constituted the second last reference to the range variable 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 removed from the **:P** clause. Furthermore, the range variable e_l may be removed from the binding clause of the enclosing **retrieve** term—thereby reducing the number of nested loops of the associated query evaluation plan. The rule for removing range variables is stated as rule T11.

IF

1. $Q/\overset{*}{r}0 \; = \; \textbf{retrieve}$
2. $Q/\overset{*}{r}Bb \; = \; (\boxed{e_l}\; BINDING)$
3. $Q/\overset{*}{r}S\overset{*}{a}i \; = \; (\textbf{in}\ (e_1 \ldots \boxed{e_l} \ldots e_k)\ (\textbf{getasr}\ [\![t_0.A_1.\cdots.A_n]\!]_X$
   $\textbf{:R}\ RESTR$
   $\textbf{:S}\ ASRselect$
   $\textbf{:P}\ (\#i_1 \ldots \boxed{\#i_l} \ldots \#i_k)))$
4. $nrOccurrences(Q/\overset{*}{r}, e_l) \; = \; 2$

THEN [T11]

1. $Q/\overset{*}{r}S\overset{*}{a}i \longrightarrow (\textbf{in}\ (e_1 \ldots e_{l-1}\ e_{l+1} \ldots e_k)\ (\textbf{getasr}\ [\![t_0.A_1.\cdots.A_n]\!]_X$
   $\textbf{:R}\ RESTR$
   $\textbf{:S}\ ASRselect$
   $\textbf{:P}\ (\#i_1 \ldots \#i_{l-1}\ \#i_{l+1} \ldots \#i_k)))$
2. $remove(Q/\overset{*}{r}Bb)$

Analogous rules exist for the other operations on access support relations, like **mkasr**, **appendasr**, **joinasr**.

Example 6.4 Consider the following transformation steps which illustrate the full use of T10 in combination with T11.

6.8 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. Rule T9 constitutes one such "last resort" to evaluate a disjunction.

IF

1. $Q/\overset{*}{r}0$ = **retrieve**
2. $Q/\overset{*}{r}S$ = (**or** $orArg_1 \ldots orArg_n$)

THEN [T9]

1. $Q/\overset{*}{r}$ $\longrightarrow$ (**union** (**retrieve :B** Q/B **:S** $orArg_1$ **:P** Q/P)
   ...
   (**retrieve :B** Q/B **:S** $orArg_n$ **:P** Q/P)

6.9 Moving Selection Predicates Inwards

The following rule group can be applied to move selection predicates "inwards." In rule T10 the meta-variable C can be any set valued term, e.g., a term with outer operator **getasr**. The predicate that matches the range variable e_l against the constant (set) C is

---

IF

1. $Q/\overset{*}{r}$ = **retrieve**
2. $Q/\overset{*}{r}S\overset{*}{a}0$ = **and**
3. $Q/\overset{*}{r}S\overset{*}{a}i$ = (**in** $(e_1 \ldots e_l \ldots e_k)$ (**getasr** $[\![t_0.A_1.\cdots.A_n]\!]_X$
   :R *RESTR*
   :S *ASRselect*
   :P $(\#i_1 \ldots \#i_l \ldots \#i_k)$)))
4. $Q/\overset{*}{r}S\overset{*}{a}j$ = (**in** e_l C)

THEN [T10]

1. $Q/\overset{*}{r}S\overset{*}{a}i$ $\longrightarrow$ (**in** $(e_1 \ldots e_l \ldots e_k)$ (**getasr** $[\![t_0.A_1.\cdots.A_n]\!]_X$
   :R *RESTR*
   :S (**and** (**in** $\#i_l$ C)
   ASRselect)
   :P $(\#i_1 \ldots i_l \ldots \#i_k)$)))
2. *remove*($Q/\overset{*}{r}S\overset{*}{a}j$)

---

moved inside the **getasr** expression. This has two distinct advantages:

1. the *ASR-Manager* has to move fewer tuples to the query evaluator (during run-time).
2. one occurrence of the range variable e_l is eliminates. This brings us one step closer to the goal to completely eliminate range variables from the query—remember, each range variable incured one level of nesteing in the evaluation.

6.7 Further Operators on Access Support Relations

6.7.1 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 most straightforward representative from this rule group is formulates as T7. The rules for utilizing such a temporary access suppport relation are analogous to counterparts in the preceding sections.

IF

1. $Q/\overset{*}{v} \ = \ (\textbf{path} \ x \ A_1 \ldots A_m)$
2. $\textit{set-valued}((\textbf{path} \ x \ A_1 \ldots A_m))$

THEN [T7]

1. $Q/\overset{*}{v} \longrightarrow (\textbf{untuple} \ (\textbf{mkasr} \ [\![\textit{type}(x).A_1.\cdots.A_m]\!]$
   $\textbf{:S} \ (= \ \#0 \ x)$
   $\textbf{:P} \ \#m))$

6.7.2 Joining Access Support Relations

A predicate based on the comparison of two path expressions for both of which an applicable access support relation exists may be transformed into the join of the two access support relations as exemplified in rule T8.

IF

1. $Q/\overset{*}{r}0 \ = \ \textbf{retrieve}$
2. $Q/\overset{*}{r}S\overset{*}{v} \ = \ (\Phi \ (\textbf{path} \ x \ A_i \ldots A_j) \ (\textbf{path} \ y \ B_k \ldots B_l))$ with $\Phi \ \in \{=, \neq, \leq, <, >, \geq\}$
3. $\textit{Applicable}([\![s_0.A_1.\cdots.A_m]\!]_X, \ \textit{type}(x).A_i.\cdots.A_j)$
4. $\textit{Applicable}([\![t_0.B_1.\cdots.B_n]\!]_{X'}, \ \textit{type}(y).B_k.\cdots.B_l)$

THEN [T8]

1. $Q/\overset{*}{r}S\overset{*}{v} \longrightarrow (\textbf{in} \ (x \ y) \ (\textbf{joinasr} \ \ [\![s_0.A_1.\cdots.A_m]\!]_X$
   $[\![t_0.B_1.\cdots.B_n]\!]_{X'}$
   $\textbf{:J} \ (\Phi \ \#j \ \#(m+1+l))$
   $\textbf{:P} \ (\#(i-1) \ \#(m+1+k)))$

The **:J** denotes the join predicate which, in this case, it matches the jth column of the first ASR with the $(m+1+l)^{th}$ column of the second ASR.

Since the join may be a very costly operation one should try every other possibility before committing this transformation T8.

Having committed the join operation one should try to optimize the join as much as possible. If the enclosing retrieve term contains a selective binding for x and y, e.g.,

$$\textbf{:B} \ (\ldots(x \ C_1) \ldots (y \ C_2) \ \ldots)$$

then these should be propagated inside the **joinasr** operation by replacing the ASRs by equivalent **getasr** terms. The selective bindings should then be moved into the **:S** clauses of the respective **getasr** term in order to minimize the number of joined tuples.

interconnected paths shown in the precondition (3) of the rule T6. Thereby, rule T6 may be essential to achieve the utilization of an existing ASR over the segmented path expression. In rule T6 the (k+1)-ary tuple $(e_0\ e_1 \ldots e_k)$ is matched against the set of

---

IF

1. $Q/\overset{*}{r}0\ =$ **retrieve**
2. $Q/\overset{*}{r}S0\ =$ **and**
3. The following terms exist for $l_j \in I\!N$ with $\Phi_j \in \{=, \textbf{in}\}$ $(1 \leq j \leq k)$:

   (i) $Q/\overset{*}{r}Sl_1\ =\ (\Phi_1\ e_1\ (\textbf{path}\ e_0\ A_{i_0+1} \ldots A_{i_1}))$

   (ii) $Q/\overset{*}{r}Sl_2\ =\ (\Phi_2\ e_2\ (\textbf{path}\ e_1\ A_{i_1+1} \ldots A_{i_2}))$

   $\vdots$

   (k) $Q/\overset{*}{r}Sl_k\ =\ (\Phi_k\ e_k\ (\textbf{path}\ e_{k-1}\ A_{i_{k-1}+1} \ldots A_{i_k}))$
4. $\mathit{Applicable}([\![s_0.A_1.\cdots.A_n]\!]_X,\ \mathit{type}(e_0).A_{i_0+1}.\cdots.A_{i_k})$

THEN [T6]

1. $Q/\overset{*}{r}Sl_1\ \longrightarrow$ (**in** $(e_0\ e_1 \ldots e_k)$ (**getasr** $[\![s_0.A_1.\cdots.A_n]\!]_X$
   :R true
   :S true
   :P $(\#i_0\ \#i_1 \ldots \#i_k)$)))
2. $\mathit{remove}(Q/\overset{*}{r}Sl_j)$ for $2 \leq j \leq k$

---

tuples obtained in the **getasr** expression.

Example 6.3 The rule T6 can be applied to our running example in order to match the range variables e, d, and m against the access support relation $[\![\text{EMP.WorksIn.Mgr}]\!]_{can}$.

```
(retrieve :B ((e EMP) (m MANAGER) (d DEPT))
          :S (and  ┌(= d (path e WorksIn))┐
                   └(= m (path d Mgr))    ┘
                   (< (path d Profit) 0)
                   (in e (untuple (getasr [[EMP.SALARY]]can        --T6-->
                                          :R true
                                          :S (> #1 200000)
                                          :P #0))))
          :P m)
```

```
(retrieve :B ((e EMP) (m MANAGER) (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 (untuple (getasr [[EMP.SALARY]]can
                                         :R true
                                         :S (> #1 200000)
                                         :P #0))))
          :P m)
```
◇

```
(retrieve :B ((e EMP) (m MANAGER) (d DEPT))
          :S (and (= d (path e WorksIn))
                  (= m (path d Mgr))
                  (< (path d Profit) 0)
                  (in e (untuple (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*)—therefore this term cannot be substituted by a more efficient **getasr** operation. ◇

6.6 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).

6.6.1 Bi-Connected Expressions

A two-target expression may, for example, have the form (**in** y (**path** x $A_i \dots A_j$)), where y and x are both range variables. If at all possible, one should try by using rules like T1 and T2 (path substitution) to eliminate one of the two range variables. But this is not always possible. Therefore, the next rule provides for utilizing an existing ASR to support the evaluation of such a two-target expression. Note however, that this rule T5 should only be applied if there is no chance to eliminate one of the range variables. This is controlled by the heuristics (cf. Section 7).

IF

1. $Q/\overset{*}{r}0$ = **retrieve**
2. $Q/\overset{*}{r}S\overset{*}{v}$ = (Φ y (**path** x $A_i \dots A_j$)) with $\Phi \in \{=, \textbf{in}\}$
3. $Applicable([\![s_0.A_1.\cdots.A_n]\!]_X,\ type(x).A_i.\cdots.A_j)$

THEN [T5]

1. $Q/\overset{*}{r}S\overset{*}{v}$ $\longrightarrow$ (**in** (x y) (**getasr** $[\![s_0.A_1.\cdots.A_n]\!]_X$
   :R true
   :S true
   :P (#($i-1$) #j)))

In this rewrite rule T5, the range variables x and y are grouped to form tuples that are matched against the tuples returned by the **getasr** expression.

6.6.2 Multipily-Connected Paths

The rule T5 can be generalized to multipily-connected path expressions. Note that further references to the range variables e_i for $(1 \leq i \leq k)$ may obstacle the prolonging of the

together with an intermediate simplification to remove the nested **and**s.

```
(retrieve :B ((e EMP) (m MANAGER))                          (retrieve :B ((e EMP) (m MANAGER)
      :S (and (= m (path [e WorksIn] Mgr))        T3,T1                    [(d DEPT)])
              (< (path [e WorksIn] Profit) 0)     ---->          :S (and [(= d (path e WorksIn))]
              (> (path e Salary) 200000))                                 (= m (path [d] Mgr))
      :P m)                                                               (< (path [d] Profit) 0)
                                                                          (> (path e Salary) 200000))
                                                                 :P m)
```

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

6.5 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. Rule T4 provides the basis for this transformation:

IF

1. $Q/\overset{*}{v} = (\Phi$ (**path** $x\ A_i \ldots A_j)\ c)$ with $\Phi \in \{ =, >, \geq, <, \leq \}$
2. $isConst(c)$
3. $Applicable([\![s_0.A_1.\cdots.A_m]\!]_X,\ type(x).A_i.\cdots.A_j)$

THEN [T4]

1. $Q/\overset{*}{v} \longrightarrow$ (**in** x (**untuple** (**getasr** $[\![s_0.A_1.\cdots.A_m]\!]_X$
   :R true
   :S $(\Phi\ \#j\ c))$
   :P $(\#(i-1))))$

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 $\#0$).

Example 6.2 Application of the above rule yields for our running example:

```
(retrieve :B ((e EMP) (m MANAGER) (d DEPT))
      :S (and (= d (path e WorksIn))
              (= m (path d Mgr))                   T4
              (< (path d Profit) 0)               ---->
              [(> (path e Salary) 200000))]
      :P m)
```

IF

1. $Q/\overset{*}{r}0 \ = \ \textbf{retrieve}$
2. $Q/\overset{*}{r}S0 \ \neq \ \textbf{or}$
3. $Q/\overset{*}{r}Si \ = \ (\Phi \ arg_1 \ arg_2)$
4. $arg_p \ = \ (\textbf{path} \ x \ A_1 \ldots A_m B_1 \ldots B_n)$ with $m \geq 1, \ n \geq 0, \quad p \in \{1, 2\}$
5. $single\text{-}valued((\textbf{path} \ x \ A_1 \ldots A_m))$
6. $nrOccurrences(Q, \ y) \ = \ 0$

THEN [T3]

1. $append(Q/\overset{*}{r}B, \ (y \ type((\textbf{path} \ x \ A_1 \ldots A_m))))$
2. $Q/\overset{*}{r}Sip \longrightarrow (\textbf{path} \ y \ B_1 \ldots B_n)$
3. $Q/\overset{*}{r}Si \longrightarrow (\textbf{and} \ Q/\overset{*}{r}Si$
   $(= \ y \ \ (\textbf{path} \ x \ A_1 \ldots A_m)))$

We had to add "$(y \ type((\textbf{path} \ x \ A_1 \ldots A_m)))$" to the binding list of the directly enclosing retrieve. Then, in further transformation steps, y may be substituted for any other path prefix "$(\textbf{path} \ x \ A_1 \ldots A_m)$" giving rise to the following abstract example:

$$
\begin{array}{lcl}
\ldots & & \ldots \\
(\textbf{and} \ (= y \ (\textbf{path} \ x \ A_1 \ldots A_m)) & & (\textbf{and} \ (= y \ (\textbf{path} \ x \ A_1 \ldots A_m)) \\
\quad (\Phi \ TERM \ (\textbf{path} \ x \ A_1 \ldots A_m \ D_r \ldots D_q)) & \longrightarrow & \quad (\Phi \ TERM \ (\textbf{path} \ y \ D_r \ldots D_q)) \\
\quad \ldots) & & \quad \ldots) \\
\ldots & & \ldots
\end{array}
$$

The combination of T3 (splitting) and T1 and T2 (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 7).

Example 6.1 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
- $[\![\text{EMP.WorksIn.Mgr}]\!]_{can}$

Then the first step of our optimization is to factor out common subpaths that could be supported by an existing access support relation. Here, the rules T3 and T1 are applied

$$
\begin{array}{l}
(\textbf{retrieve :B}\ ((e\ \text{BINDING})\ \text{BINDING_LIST}) \\
\quad \textbf{:S}\ (\textbf{and}\ (\textbf{in}\ e\ (\textbf{path}\ v\ A_i \ldots A_j)) \\
\qquad\qquad (\textbf{in}\ \text{T}\ (\textbf{path}\ e\ A_{j+1} \ldots A_l)) \\
\qquad\qquad \text{SEL_PREDICATE}) \\
\quad \textbf{:P}\ \text{PROJ})
\end{array}
\longrightarrow
\begin{array}{l}
(\textbf{retrieve :B}\ (\text{BINDING_LIST}) \\
\quad \textbf{:S}\ (\textbf{and}\ (\textbf{in}\ \text{T}\ (\textbf{path}\ v\ A_i \ldots A_j\ A_{j+1} \ldots A_l)) \\
\qquad\qquad \text{SEL_PREDICATE}) \\
\quad \textbf{:P}\ \text{PROJ})
\end{array}
$$

The substitution may be applied if e does not occur free in *SEL_PREDICATE*, T (standing for a term) and *PROJ*. The general rule for this substitution is stated as T2—again, the rule is generalized to handle nested **retrieve** terms as well. Precondition (7) requires that e occurs exactly three times in the **retrieve** term at position $Q/\overset{*}{r}$: once in the binding list, and in the two predicates at positions $Q/\overset{*}{r}S\overset{*}{a}p$ and $Q/\overset{*}{r}S\overset{*}{a}q$. In precondition (5) we require that the "flattened" range of the path expression is a subset of the range of the variable e. This is determined on the basis of the types of the path expression and the range variable, i.e., it is deduced from the subtype/supertype relationship determined from the database schema.

IF

1. $Q/\overset{*}{r} \ = \ \textbf{retrieve}$
2. $Q/\overset{*}{r}S\overset{*}{a}0 \ = \ \textbf{and}$
3. $Q/\overset{*}{r}S\overset{*}{a}p \ = \ (\textbf{in}\ e\ (\textbf{path}\ v\ A_i \ldots A_j))$
4. $Q/\overset{*}{r}S\overset{*}{a}q \ = \ (\textbf{in}\ arg_1\ (\textbf{path}\ e\ A_{j+1} \ldots A_l))$
5. $\mathit{flatRange}((\textbf{path}\ v\ A_i \ldots A_j)) \ \subseteq \ \mathit{range}(e)$
6. $Q/\overset{*}{r}Bb \ = \ (e\ \mathit{BINDING})$
7. $\mathit{nrOccurrences}(Q/\overset{*}{r},\ e) \ = \ 3$

THEN [T2]

1. $Q/\overset{*}{r}S\overset{*}{a}q \ \longrightarrow \ (\textbf{in}\ arg_1\ (\textbf{path}\ v\ A_i \ldots A_j\ A_{j+1} \ldots A_l))$
2. $\mathit{remove}(Q/\overset{*}{r}S\overset{*}{a}p)$
3. $\mathit{remove}(Q/\overset{*}{r}Bb)$

There is another rule which allows prolonging if the intermediate (connecting) range variable e is further qualified only in a disjunction, i.e., in a term of the form

$$(\textbf{or}\ \mathit{TERM}_1\ \mathit{TERM}_2\ \ldots)$$

In this case e may be eliminated even if it occurs in more than one disjunct TERM_i.

6.4 Splitting Path Expressions

Splitting of path expressions may be needed to "factor out" the subpath that is supported by an existing access support relation $[\![t_0.A_1.\cdots.A_n]\!]_X$. We will provide the rule for linear paths only—an analogous rule exists for set-valued path expressions.

Let y be a new variable not occurring anywhere in the entire term Q—this condition is stated in condition (6) of rule T3.

abstract example the range variable e is itself generalized to a linear path expression (**path** $x\ A_1 \dots A_m$). Furthermore, as opposed to the abstract example rule T1 is generalized to nested **retrieve** terms.

IF

1. $Q/\overset{*}{r}0 \ = \ \textbf{retrieve}$
2. $Q/\overset{*}{r}S\overset{*}{a}0 \ = \ \textbf{and}$
3. $Q/\overset{*}{r}S\overset{*}{a}i \ = \ (= \ (\textbf{path}\ x\ A_1 \dots A_m)\ (\textbf{path}\ y\ B_1 \dots B_n))$
4. $Q/\overset{*}{r}S\overset{*}{a}j \ = \ (\Phi\ arg_1\ arg_2)$ with $\Phi \in \{ =, \textbf{in}, <, \leq, >, \geq, \dots\}$
5. $i \neq j$
6. $arg_p \ = \ (\textbf{path}\ y\ B_1 \dots B_n\ A_{m+1} \dots A_{m+k})$ with $p \in \{1, 2\}$

THEN [T1]

1. $Q/\overset{*}{r}S\overset{*}{a}jp \longrightarrow (\textbf{path}\ x\ A_1 \dots A_m\ A_{m+1} \dots A_{m+k})$

In this rule the path expression "(**path** $y\ B_1 \dots B_n\ A_{m+1} \dots A_{m+k}$)" is replaced by the equivalent path expression "(**path** $x\ A_1 \dots A_n\ A_{m+1} \dots A_{m+k}$)". The equivalence of the two path expressions is ensured by the term at position $Q/\overset{*}{r}S\overset{*}{a}i$. The motivation behind this rewriting rule is indicated by the choice of meta-variable names for attributes, i.e., $A_1, \dots, A_m, A_{m+1}, \dots, A_{m+k}$. One uses this rule T1 to "line up" a path expression for which an access support relation $[\![t_0.A_1.\cdots.A_n]\!]_X$ exists.

6.3.2 Prolonging a Set-Valued Path Expression

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

(**retrieve :B** ((m MANAGER) (c CAR))
  :S (**and** (**in** c (**path** m Cars))
      (= 'Jaguar' (**path** c Make))
      (= 150 (**path** c HorsePower)))
  :P m)

$\not\equiv$

(**retrieve :B** ((m MANAGER))
  :S (**and** (**in** 'Jaguar' (**path** m Cars Make))
      (**in** 150 (**path** m Cars HorsePower)))
  :P m)

The left-hand **retrieve** term finds all *MANAGER*s who own a 'Jaguar' with 150 *HorsePowers*. The right-hand term, however, retrieves the *MANAGER*s 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 linear paths has to be properly 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 e is qualified only once in the **:S** clause. An abstract example is as follows:

8. analogously, *flatRange*(p) returns for a set-valued path expression p the "flattened" range.

Additionally, the infix functions $+, -$, etc. are evaluated at rule application time if they specify a column number in a projection clause.

Each of the subsequent subsections will present a rule group—except for the next subsection. For each rule group we exemplify the rules therein with a representative rewriting rule which, whenever possible, is illustrated by application to our running example. The optimization is separated into three main phases. First is a preprocessing phase introduced in the next subsection which is followed by the main optimization phase where the different rules are applied (subsequent subsections).

6.2 Preprocessing

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 7).

6.3 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. 3.6)—depending on the extension X of the respective ASR. The reader should recall that, for example, the cannonical extension of an access support relation $[\![t_0.A_1.\cdots.A_n]\!]_{can}$ can only be utilized if the path expression originates in t_0 and leads to t_n via the attribute chain $A_1.\cdots.A_n$.

6.3.1 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 (all capitalized words denote term variables, and e and v denote term variables for range variables only):

$$T \equiv \left[\begin{array}{l} (\textbf{retrieve :B } ((e \text{ BINDING}) \text{ BINDING_LIST}) \\ \qquad\qquad \textbf{:S } (\textbf{and} \quad (= \ e \ (\textbf{path} \ v \ A_i \ \ldots \ A_j)) \\ \qquad\qquad\qquad\qquad \text{SEL_PRED}) \\ \qquad\qquad \textbf{:P } \text{PROJ_LIST}) \end{array}\right]$$

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

$$(\textbf{path } e \ A_{j+1} \ldots A_l) \quad \longrightarrow \quad (\textbf{path } v \ A_i \ldots A_j \ A_{j+1} \ldots A_l)$$

A further simplification is possible if—after the transformation—the range variable e is not further qualified in T. In this case "(e BINDING)" 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.

Having motivated prolonging on the above (abstract) example let us now formulate the rule T1 which handles the general case of linear path expressions—that is, in terms of our

iff $\overset{*}{v} = \overset{*}{w}i$, and $L/\overset{*}{w} = (L_0 \ldots L_n)$. The last two functions needed are *append* and *frontappend* which append a list to the end or the front of a list, respectively. Thus

$$append(L/\overset{*}{v},\ L') := L/\overset{*}{v} \ \rightarrow\ (L_1 \ldots L_n\ L')$$

and

$$frontappend(L/\overset{*}{v},\ L') := L/\overset{*}{v} \ \rightarrow\ (L'\ L_1 \ldots L_n)$$

if $L/\overset{*}{v} = (L_1 \ldots L_n)$.

Our rule system does not contain rules which apply unrestrictedly. Rather, in order to guarantee semantic correctness they have the form

IF *condition-part* THEN *rewrite-part*

The *condition-part* consists of a list of conditions logically connected by **and**. The *rewrite-part* consists of a list of rewrite specifications which are all executed if the rule is applied to a term. This is done only if the *condition-part* is satisfied.

The following additional evaluable functions may occur in a rule specification. They allow the computation of arguments to operators which play a role in the applicability test of a rule or within the rewrite part. Analogous to the above rewriting functions they are written in italics. The number of occurrences of a list L' in a list L will play a major role as a precondition to the rewrite rules specified below. The number of occurrences

$$nrOccurrences : \mathbb{L} \times \mathbb{L} \rightarrow \mathbb{N}$$

is defined as

$$nrOccurrences(L, L') := |\{\overset{*}{v} \mid L/\overset{*}{v} = L'\}|.$$

where $\mathbb{L}$ denotes the set of all lists.

The list of evaluable functions further contains:

1. *Applicable*(a, p), returning *true* if the access support relation a is applicable for evaluating the path p.

2. *single-valued*(p), returning *true* if the path p is single-valued.

3. *set-valued*(p), returning *true* if the path p is set-valued.

4. *type*(x) returns for an expression x its type.

5. *FlatTargetType*(p) returns for a set-valued path p the type of the elements of the result set.

6. *isConst*(c) returns *true* if c is a constant or a set of constants

7. *range*(p) constitutes the result set of evaluating the path expression p. However, the *range* predicate is—in the term rewriting system—used to relate two path expressions in the form

   $$range(p) \subseteq range(p')$$

   where the satisfaction can be deduced based on the syntactical super/sub-type relationship without actually evaluating the range of either path expression p or p'.

6 Transformation Rules to Optimize Term Expressions

6.1 Preliminaries

The query optimization steps are described as transformation rules or term rewriting rules [10]. A rule is given in the form $l \to r$ which specifies that the term l is to be replaced by the term r. Since our term language consists only of lists we recast the major definitions of [10] to fit them for lists. But first we introduce some notational conventions used throughout the rest of the paper. Let $I\!N$ denote the set of natural numbers, and $I\!N^*$ the set of finite sequences of natural numbers. These sequences will be used to denote positions within our terms. Consequently, we call them positions. The elements of $I\!N^*$ are denoted by $\overset{*}{a}, \ldots, \overset{*}{z}$, or ϵ which denotes the empty sequence. For an element $\overset{*}{a} \in I\!N^*$ $|\overset{*}{a}|$ defines the length of $\overset{*}{a}$. Concatenation of elements $\overset{*}{v}, \overset{*}{w} \in I\!N^*$ is denoted by $\overset{*}{v}\overset{*}{w}$. We further define a partial ordering ($\succeq$) on the elements of $I\!N^*$ by

$$\overset{*}{u} \succeq \overset{*}{w} \Leftrightarrow \exists \overset{*}{v} \in I\!N^* : \overset{*}{u}\overset{*}{v} \;=\; \overset{*}{w}.$$

Thus $\overset{*}{u} \succeq \overset{*}{w}$ iff $\overset{*}{u}$ is a prefix of $\overset{*}{w}$.

Next we define the infix operator / defined for a list and a position. The result is the sublist occurring at the specified position. More formally we define for a list $L = (L_0 \ldots L_n)$ with sublists L_i $(0 \le i \le n)$:

$$L/\overset{*}{v} := \begin{cases} L & \text{if } \overset{*}{v} = \varepsilon \\ (L_i)/\overset{*}{w} & \text{if } 0 \le i \le n \text{ and } \overset{*}{v} = i\overset{*}{w} \\ \uparrow & \text{else} \end{cases}$$

We speak of $L/\overset{*}{v}$ as L at the position $\overset{*}{v}$.

In order to make the notation more mnemonic we use the following constants, B for 1, S for 2, and P for 3. These constants are used to identify the position of the **:B**, **:S**, and **:P** subterm of a **retrieve** expression, respectively. Thus, for example, $Q/\overset{*}{v}S\overset{*}{w}$ stands for $Q/\overset{*}{v}2\overset{*}{w}$ (and means the position $\overset{*}{w}$ within the selection predicate in the **retrieve** term at position $\overset{*}{v}$ of the Query Q. Note that we do not count the labels within a list. Thus for the definition of the position they are treated as non-existent.

It is sometimes hazardous to specify the rewrite process using the simple notation $L \to L'$ since we need functions with specific side-effects on our terms. Thus, we define some abbreviation with mnemonic identifiers. This will also enhance readability. For two lists $L = (L_0 \ldots L_n)$ and L', and a position $\overset{*}{v}$ we define the function *replace*:

$$replace(L,\ \overset{*}{v},\ L') := \begin{cases} L' & \text{if } \overset{*}{v} = \varepsilon \\ (L_0 \ldots L_{i-1}\ replace(L/i,\ \overset{*}{w},\ L')\ L_{i+1} \ldots L_n) & \text{if } \overset{*}{v} = i\overset{*}{w} \end{cases}$$

which is normally abbreviated by $L/\overset{*}{v} \;\to\; L'$.

The function *remove* is defined as

$$remove(L/\overset{*}{v}) := \Big(L/\overset{*}{w} \to (L_0 \ldots L_{i-1}\ L_{i+1} \ldots L_n)\Big)$$

5.1.7 Terms

Terms, as used for the selection predicate, *SELPRED* and *RESTR*, are of the form (**op** $t_1 \ldots t_n$) where $t_1 \ldots t_n$ are terms. Here, **op** is a boolean connector, e.g., **and, or, not** (in this case, $n = 1$), and the terms t_i are either terms, that again, represent selection predicates, or they consist of 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** being a comparator, e.g., $=$, $>$, **in**, **seteq**, etc.

Additionally to the main operators introduced so far there exist some more technical operators like **unset** and **untuple**. The operator **unset** turns a singleton, i.e., a set with a single element, into this element. The **untuple** operator returns for a set of tuples with a single column the projections onto this column.

5.2 Translation of Retrieve Expressions into Term Representation

The initial translation of a **retrieve** expression into a term is straightforward. 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 5.1 To make things more concrete we give the translation of the query in Example 2.2 into the corresponding term representation:

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

This not yet optimized term expression gives way to 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.

In the term resulting from the translation of a query all negations are eliminated in the usual way using de Morgan's law and reverting the comparators.

5.3 Evaluation of Term Expressions

In this short subsection we give a sneak preview of how terms are evaluated. In order to evaluate a term it is translated into executable code which then is executed. This translation follows the nested-loop paradigm. For the above example term we get the following code.

```
foreach e in EMP do
   foreach m in MANAGER do
      if
         m = e.WorksIn.Mgr and e.WorksIn.Profit > 0 and e.Salary > 200000
      then
         output(m);
```

The problem of translating terms into executable code is discussed in more detail in Section 8.

(**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 [6], the *RESTR* predicate can only refer to attributes at the left and/or right of an access support relation partition.

5.1.3 Creating Temporary Access Support Relations

In order to avoid the *nested loop* evaluation of queries it may be more advantageous to create a temporary access support relation:

(**mkasr** ASRSPEC **:S** SELPRED **:P** PROJ)

This operator materializes a new temporary access support relation. *ASRSPEC* is the specification of an access support relation as defined in Definition 3.4. The result contains the projection specified by *PROJ* of those tuples of the specified access support relation which satisfy the selection predicate *SELPRED*.

5.1.4 Extending an Existing ASR

(**appendasr** ASR PATHEXPR **:S** SELPRED **:P** PROJ)

The **appendasr** operator is utilized to extend an existing access support relation beyond the originally defined attribute chain

5.1.5 Joining Two Access Support Relations

Each of the two operators above 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, semijoin, union, etc.

(**joinasr** ASR ASR **:J** JOINPRED **:S** SELPRED **:P** PROJ)

The same syntax as for the join term applies for the semijoin operator.

(**semijoinasr** ASR ASR **:J** JOINPRED **:S** SELPRED **:P** PROJ)

In the latter case the projection is restricted to columns of the first *ASR*.

5.1.6 Scanning Type Extensions

The scan operator scans the extension of the type *TYPE* for objects satisfying the selection predicate *SELPRED*. Note that the selection predicate must be restricted to concern only attributes within an object of type *TYPE*. Thus, it is not allowed to reference objects other than of type *TYPE*.

(**scan** *TYPE* **:S** SELPRED)

[6] This limitation is due to the storage structure of ASRs—discussed in Section 3 on page 13.

5 The Term Language: A Neutral Query Representation Language

5.1 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 queries in the QUEL-like language into the term representation. Of course, this step may be more complicated for other query languages than the one used here. But then at least the indepencence of the term language from the query language guarantees that only the preprocessing phase of the query optimizer has to be redesigned.

In the following we represent terms in prefix notation utilizing (general) lists with the operator being the first element of the list. The other list elements represent the arguments of the operator. We will often use mnemonic labels (denoted by **:L** for a label **L**) to increase readability. To avoid representing an abstract grammar for our term language we discuss the main operators together with its intended semantics within this section. Note that the list of introduced operators is not complete. This is due to the fact that we cannot discuss all rules contained in our query optimizer. As a consequence we thus concentrate on the "kernel" of the optimizer emphasizing the exploitation of access support relations. The discussion of the term language is organized top down.

5.1.1 The Retrieve Operator

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

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

This term—directly—yields a nested loop evaluation of the query specified by the parameters. In the **:B** clause the variables are bound from left to right to every possible object (or value) of the corresponding set. This is denoted by the list of pairs in the **:B** clause each one consisting of (a) range variable(s) and a type name—which, more precisely, stands for the *extension* of the type—or a set valued expression. During the evaluation of the **retrieve** term the selection predicate following the label **:S** is evaluated on each binding, and in the case of success the binding of the variable corresponding to the one in the **:P** clause is gathered. For an example query evaluation plan see Section 5.3. 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 discussed now.

5.1.2 Utilizing Access Support Relations

The main goal of our optimization is to exploit existing access support relations for query evaluation. This is facilitated mainly by the **getasr** operator:

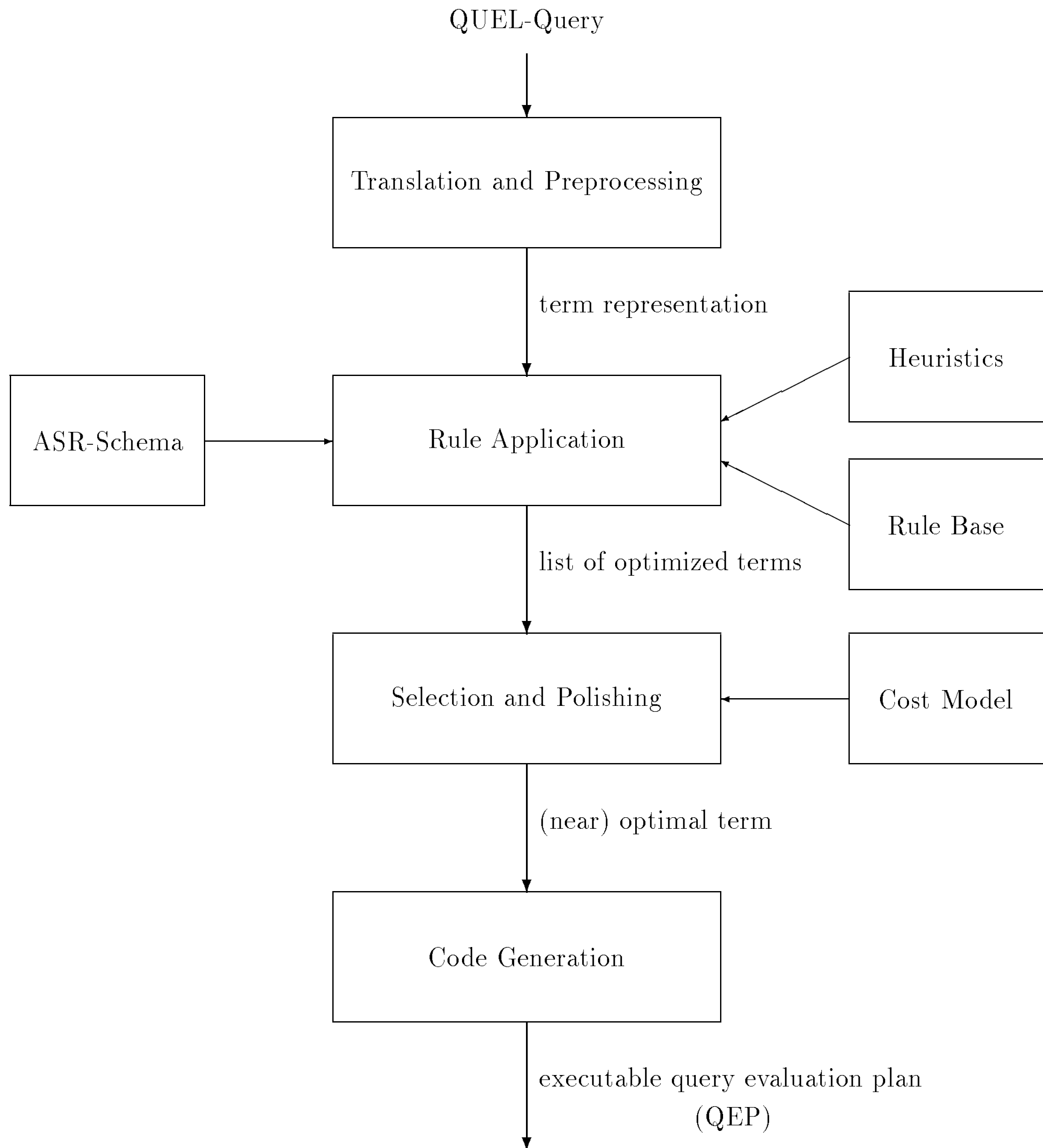

Figure 2: Outline of the GOM Query Processing Architecture

The storage structure of access support relations is borrowed from the binary join index proposal by Valduriez [23]. 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 ([13] contains 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.

Definition 3.6 (Applicability) *An access support relation* $[\![t_0.A_1.\cdots.A_n]\!]_X$ *under extension* X *is applicable for a path* $s.A_i.\cdots.A_j$ *with* $s \leq t_{i-1}$ *under the following condition—depending on the extension* X*:*

$$Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_i.\cdots.A_j) = \begin{cases} X = full & \wedge \quad 1 \leq i \leq j \leq n \\ X = left & \wedge \quad 1 = i \leq j \leq n \\ X = right & \wedge \quad 1 \leq i \leq j = n \\ X = can & \wedge \quad 1 = i \leq j = n \end{cases}$$

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

4 Overview of the GOM Architecture

In Figure 4 the architecture of the GOM query processing system is outlined. Currently, our optimizer supports only the declarative QUEL-like query language. In the future we intend to support other declarative query languages as well as the optimization of procedurally specified database access.

The declarative QUEL query is validated and translated into a neutral query representation language: the *term language* which is the subject to the next section. The term rewriting system is applied to this term representation by the *Rule Application* module. The *Rule Base* as well as the *Heuristics* module are designed in a highly modular and extensible fashion. The *Rule Base* currently comprises 90 non-trivial term rewriting rules. Application of the rewriting rules is governed by the existing access support relations which are determined from the *ASR-Schema*. The *Rule Application* module generates a list of (equivalent) optimized terms of which the most efficient is chosen in the *Selection and Polishing* phase. The *Cost Model* forms the basis for this selection since it allows to estimate costs incurred by different query evaluation plans. The chosen term representation is directed to the *Code Generation* module that transforms the term into an executable query evaluation plan (QEP).

Example 3.2 For the path of Example 3.1 the full extension, which is denoted as $[\![EMP.WorksIn.Mgr.Cars.Make]\!]_{full}$, looks as follows:

$[\![$**EMP.WorksIn.Mgr.Cars.Make**$]\!]_{\mathbf{full}}$				
OID_{EMP}	OID_{DEPT}	$OID_{MANAGER}$	OID_{CAR}	$STRING$
...	...	...	...	...
id_1	id_5	id_8	id_{11}	"Jaguar"
id_2	id_5	id_8	id_{11}	"Jaguar"
id_2	id_5	id_8	id_{12}	"BMW"
id_3	id_6	id_9	—	—
—	id_7	id_{10}	id_{13}	"Benz"
...	...	...	...	...

◇

This extension contains all paths and subpaths corresponding to the underlying path expression. The first three tuples actually constitute complete paths which would be present in the canonical extension as well; however the last two paths would be omitted in the canonical extension. In the left-complete extension the first four tuples would be present, whereas the last one would be omitted since it does not originate in *EMP*. Analogously, the right-complete extension would contain the first three and the last tuple and omit the fourth tuple since it does not "go all the way through" to a *STRING* representing the *Make* of some *CAR*.

Aside from extensions, we also allow decomposition of access support relations. The following formally defines valid (lossless) decompositions:

Definition 3.5 (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*

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

are called a decomposition of $[\![t_0.A_1.\cdots.A_n]\!]_X$. The relations $[\![t_0.A_1.\cdots.A_n]\!]_X^{(i_j,i_{j+1})}$ are called partitions for $(0 \le j \le k)$[5] They are materialized by projecting the corresponding attributes of $[\![t_0.A_1.\cdots.A_n]\!]_X$:

$$[\![t_0.A_1.\cdots.A_n]\!]_X^{(i_j,i_{j+1})} := \Pi_{(S_{i_j},S_{i_j+1},\ldots,S_{i_{j+1}})}\Big([\![t_0.A_1.\cdots.A_n]\!]_X\Big)$$

If every partition is a binary relation the decomposition is called binary. The above decomposition is denoted by $(0, i_1, i_2, \ldots, i_k, n)$. □

[5] For notational convenience let $i_0 := 0$ and $i_{k+1} := n$.

$[\![$EMP.WorksIn$]\!]$	
OID_{EMP}	OID_{DEPT}
id_1	id_5
id_2	id_5
id_3	id_6
id_4	id_6
id_8	id_5
...	...

$[\![$DEPT.Mgr$]\!]$	
OID_{DEPT}	$OID_{MANAGER}$
id_5	id_8
id_6	id_9
id_7	id_{10}
...	...

$[\![$MANAGER.Cars$]\!]$	
$OID_{MANAGER}$	OID_{CAR}
id_8	id_{11}
id_8	id_{12}
id_{10}	id_{13}
id_{10}	id_{14}
...	...

$[\![$CAR.Make$]\!]$	
OID_{CAR}	$STRING$
id_{11}	"Jaguar"
id_{12}	"BMW"
id_{13}	"Benz"
id_{14}	"Toyota"
...	...

◇

Note, that $[\![$Emp.WorksIn$]\!]$ contains also tuples like (id_8, id_5) that relate a MANAGER instance to the DEPT instance that is referenced via WorksIn. This is due to the fact that MANAGER is a subtype of EMP.

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 (all the way) 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 3.4 (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:*

$$
\begin{aligned}
[\![t_0.A_1.\cdots.A_n]\!]_{can} &:= [\![t_0.A_1]\!] \bowtie \ldots \bowtie [\![t_{n-1}.A_n]\!] \\
[\![t_0.A_1.\cdots.A_n]\!]_{full} &:= [\![t_0.A_1]\!] ⟗ \cdots ⟗ [\![t_{n-1}.A_n]\!] \\
[\![t_0.A_1.\cdots.A_n]\!]_{left} &:= \left(\cdots\left([\![t_0.A_1]\!] ⟕ [\![t_1.A_2]\!]\right) ⟕ \cdots [\![t_{n-1}.A_n]\!]\right) \\
[\![t_0.A_1.\cdots.A_n]\!]_{right} &:= \left([\![t_0.A_1]\!] ⟖ \cdots \left([\![t_{n-2}.A_{n-1}]\!] ⟖ [\![t_{n-1}.A_n]\!]\right)\cdots\right)
\end{aligned}
$$

□

The second part of the definition is useful to support access paths through sets[4]. 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.

As we will see the information contained in a path can be hold in a relation. Consequently, 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 3.2 (Access Support Relation) *Let $t_0, \ldots, t_n$ be types, $t_0.A_1.\cdots.A_n$ be a path expression. 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 of definition 3.1 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 auxiliary relations $[\![t_0.A_1]\!], \ldots, [\![t_{n-1}.A_n]\!]$.

Definition 3.3 (Auxiliary Binary Relations) *For each i $(1 \leq i \leq n)$—that is, for each attribute in the path expression—we construct the auxiliary binary relation $[\![t_{i-1}.A_i]\!]$. The relation $[\![t_{i-1}.A_i]\!]$ contains the tuples $(id(o_{i-1}), id(o_i))$ for every object o_{i-1} of type t_{i-1} and o_i of type t_i such that*

- *$o_{i-1}.A_i = o_i$ if A_i is a single-valued attribute.*
- *$o_i \in o_{i-1}.A_i$ if A_i is a set-valued attribute.*

If t_n is an atomic type then $id(o_n)$ corresponds to the value $o_{n-1}.A_n$. Note, however, that only the last type t_n in a path expression can possibly be an atomic type. □

Example 3.1 Let us re-consider the path expression of our schema *Company* (now we indicate the types of the subpaths by the underbraces):

$$P \equiv \underbrace{\underbrace{\underbrace{\underbrace{EMP.WorksIn}_{DEPT}.Mgr}_{MANAGER}.Cars}_{CAR}.Make}_{STRING}$$

For this path expression the auxiliary binary relations have the following extensions:

[4] Note, however, that we do not permit powersets.

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

Even though, the predicate of this query is much more complex, the QUEL formulation is still easy to understand. For our example object base *Company* the result of this query is the one *MANGER* object with OID id_9. ◇

3 Access Support Relations

In an earlier paper [14] 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:

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

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 3.1 (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:*

- *The 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*[3]*.*
- *The 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 $t_0.A_1.\cdots.A_n$.* □

For simplicity of the presentation we assumed that the involved types are not being defined as a subtype of some other type. This—of course— would be possible; it would only make the definition a bit more complex to read.

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

2.4 The Query Language

For our object model we developed a QUEL-like query language along the lines of the EXCESS object query language that was designed as the declarative query language for the EXTRA object model [5].

Let x_i be variables, T_i set typed expressions or type names, 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 $S(x_1, \ldots, x_n)$

The selection predicate S in variables $x_1, \ldots, x_n$ may consist of path expressions, comparison operators, set operators, boolean connectors and may also contain a (nested) **retrieve** statement. Note, however, that our current implementation of the GOM query language facilitates single-target queries only.

Example 2.1 In this example query we want to utilize the path expression which was already described above: *EMP.WorksIn.Mgr.Cars.Make* The query considered is as follows:

> "Retrieve the *EMP*loyees whose *MANGER* drives, among other *CAR*s, a Jaguar"

range e : EMP
retrieve e
where "Jaguar" **in** e.WorksIn.Mgr.Cars.Make

In terms of our example database shown in Figure 2.1 the result of this query is the set $\{id_1, id_2\}$, i.e., the OIDs of the two qualifying *EMP* instances named "Versace" and "Lagerfeld".

In an object base that provides no access support—other than the uni-directional references—the number of pages to be accessed in the evaluation of this query is proportinal to[2]

$$\#(EMP) + \#(DEPT) + \#(MANAGER) + \#(CAR)$$

where $\#(t)$ denotes the cardinality of the extension of type t. This cost is induced because—no matter how good the query evaluation algorithm performs—every instance of the respective type has to be visited at least once. ◇

Example 2.2 The following example query will be used later on in this paper for demonstrating our query transformation rules. Again, the query is based on our *Company* object base:

> "Retrieve the managers of departments which generate losses and, at the same time, pay at least one of their employees an exorbitant salary exceeding 200000."

[2] If we assume that every *DEPT* has at least one *EMP* and every *MANAGER* manages at least one *DEPT* and every *CAR* is used by at least one *MANAGER*.

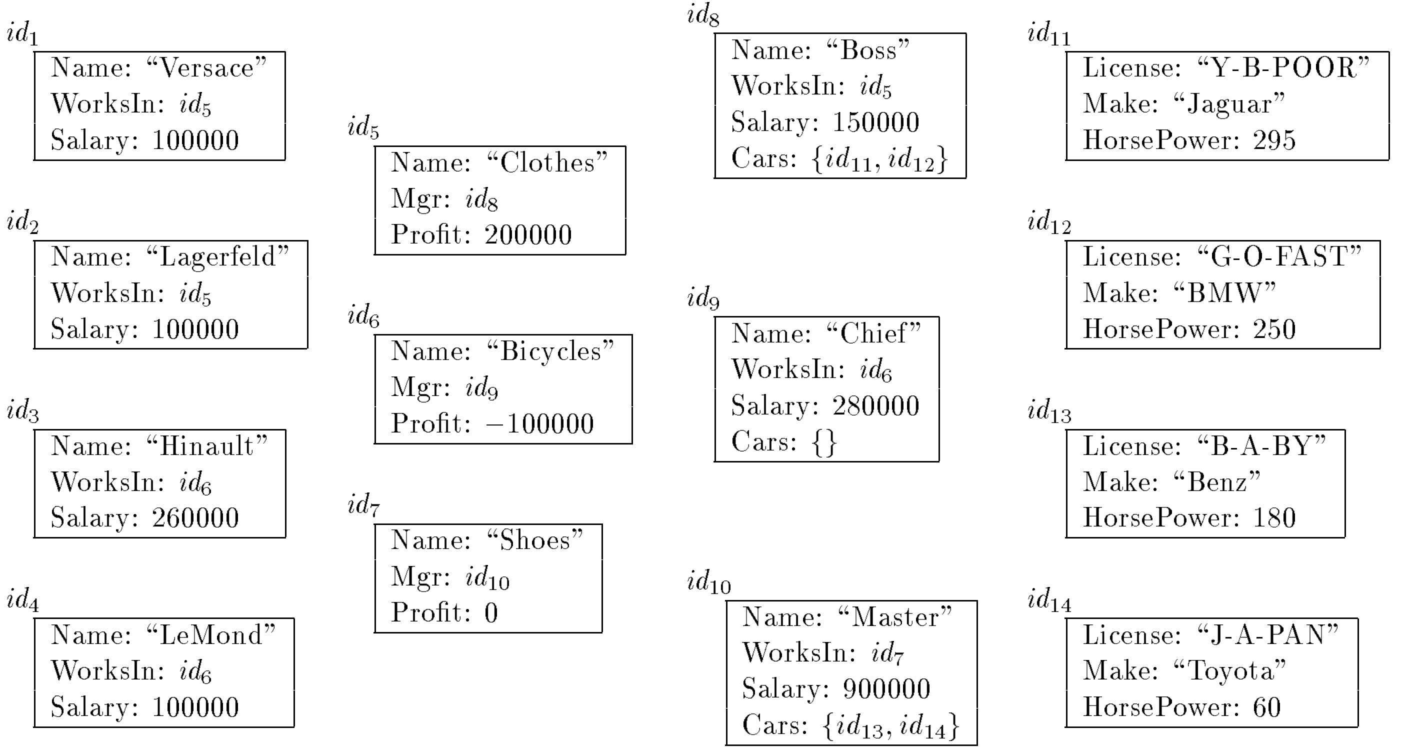

Figure 1: Example Extension of the Object Base *Company*

A (small) database extension based on the above schema is shown in Figure 1.

The id_j for $j = \{1, 2, 3, \ldots\}$ denote the object identifiers (OID) which are system-wide unique. References via complex attributes are—as mentioned before—maintained unidirectionally in GOM. 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. These references are maintained by storing the unique OID of the referenced object.

Based on the example schema, we want to give an intuitive description of a path expression—the formal definition is provided in Definition 3.1 on page 9. The following is a path expression on the *Company* schema:

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

This path expression starts in some object type and—implicitly—traverses all objects referenced by the intermediate attributes. These attributes are either single-valued, as *WorksIn*, *Mgr*, and *Make*, or set-valued, as *Cars*. Graphically, the above path expression can be outlined as follows:

$$\text{EMP} \xrightarrow{\text{WorksIn}} \text{DEPT} \xrightarrow{\text{Mgr}} \text{MANAGER} \xrightarrow{\text{Cars}} \twoheadrightarrow \text{CAR} \xrightarrow{\text{Make}} \text{STRING}$$

The single arrows represent single-valued attributes, the double arrows denote set-valued attributes.

strong typing GOM is strongly typed, meaning that all database components, e.g., attributes, set elements, etc., are constrained to a particular type and, further, that the type correctness can be verified at compile time.

instantiation Types can be instantiated to render a new object instance.

uni-directional references Even though this is actually an implementation issue, because of its relevance to our indexing scheme we want to mention here, that GOM—like almost all other object models—maintains references from one object to another only uni-directionally.

2.2 Type Definitions

If $s, t, t_1, \ldots, t_n$ are types, and $a_1, \ldots, a_n$ are pairwise distinct attribute names then

type t [**supertype** s] **is**
$[a_1 : t_1, \ldots, a_n : t_n]$

is a tuple structured type definition[1]. In this case, the **supertype s** must itself be a tuple-structured type. The type t is called a (direct) **subtype** of s and inherits all attributes of s (including those that s inherited from its supertype, if any).

Aside from tuple-structured types GOM provides built-in support for two *collection types*: *sets* and *list* which are defined as follows:

type t **is** $\{s\}$ **type** t **is** $< s >$

Here, s has to be a tuple-structured (i.e., complex) object type or an atomic type. At the present we do not deal with nested set types with respect to our indexing structures. Since the access support on ordered collection, i.e., lists, is analogous to sets we will not elaborate on list-structured types in the remainder of this paper.

2.3 Running Example

In this subsection we will introduce an example object base, called *Company*. This database will be used throughout the paper to illustrate our optimization concepts. The type definitions are as follows:

type EMP **is**
[Name: STRING,
WorksIn: DEPT,
Salary: INT];

type MANAGER
supertype EMP **is**
[Cars: {CAR}];

type DEPT **is**
[Name: STRING,
Mgr: MANAGER,
Profit: INT];

type CAR **is**
[License: STRING,
Make: STRING,
HorsePower: INT]

[1] We presented only the structural parts of our object type definitions; of course, there are type-specific operations that can be defined by the database user.

considered to derive a near-optimal evaluation plan.

Related work on object-oriented query processing is reported in [12, 16] 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. Our object model together with its declarative query language is presented in Section 2. In Section 3 we outline the access support relations as a means for access support along reference chains. Section 4 outlines the architecture of the GOM query processing system. Then in Section 5 we develop a term representation into which the QUEL-like queries are translated for the optimization process. The transformation rules are discussed in Section 6. In order to reduce the search costs we develop heuristics for the sequence of applying the transformation rules in Section 7. In Section 8 our approach to query evaluation is outlined. Section 9 concludes the paper with a summary and a discussion of future developments.

2 GOM and its Declarative Query Language

2.1 Main Concepts of GOM

This research is based on an object-oriented model that unites the most salient features of many recently proposed models in one coherent framework. In this respect, the objective of GOM can be seen as providing a syntactical framework of the essential object-oriented features identified in the "Manifest" [1]—albeit the GOM model was developed much earlier. Similarly, Zdonik and Maier developed the so-called Reference Model in [24]. The features that GOM provides are relatively *generic* such that the results derived for this particular data model can easily be applied to a variety of other object-oriented models. This helps to overcome the diversity of existing object-oriented models which is due to the lack of a commonly adhered to base model which, for example, helped in the relational database area to focus the research in one direction.

GOM provides the following object-oriented concepts:

object identity Each object instance has an identity that remains invariant throughout its lifetime. The object identifier is invisible for the database user; it is used by the system to reference objects. This allows for shared subobjects because the same object may thus be associated with many database components.

values GOM has a built-in collection of elementary (value) types, such as *char*, *string*, *integer*, etc. Instances of these types do not possess an identity.

type constructors The most basic type constructor is the tuple constructor—denoted as []—which aggregates attributes to one object. In addition, GOM has the two built-in collection type constructors set, denoted as {}, and list, denoted as <>.

subtyping A tuple-structured type t may be defined as the subtype of one other tuple-structured type t' which means that t inherits all attributes of the supertype t'.

1 Introduction

Record-oriented database systems, e.g., those based on the pure relational or the CODASYL network model, are for a variety of reasons believed to be inappropriate for engineering applications. Object-oriented database systems are emerging as the next generation DBMSs for—at least—the non-standard application domains. However, these systems are still not adequately optimized: for applications which involve a lot of associative search for objects on secondary memory they still have problems even to keep up with the performance achieved by, for example, relational DBMSs [7]. 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 couple of 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., [12]—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., [21, 11]) and semantic query optimizer techniques, e.g., [6, 22], can be applied to object-oriented databases. However, the full potential of the object-oriented paradigm can only be exploited for optimization if new access support structures and their utilization in query evaluation 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 [14]—described in this paper constitute one such approach. Access support relations are a generalization of an indexing technique for path expressions first proposed for the GemStone data model [20] and, later, applied to ORION [3]. 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 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., [8, 11, 18]. [9] reports on a rule-based query optimizer generator, which was designed for their database generator EXODUS [4]. 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

7 The Rule Interpreter and Search Heuristics **32**
7.1 Detection of "Usable" Access Support Relations 35
7.2 Rule Organization 36

8 Evaluating Optimized Terms **38**
8.1 Translation of Terms into a Graph Representation 38
8.2 Translating the Graph Representation into Executable Code 39

9 Conclusion **40**

Contents

1 Introduction 4

2 GOM and its Declarative Query Language 5
- 2.1 Main Concepts of GOM . . . 5
- 2.2 Type Definitions . . . 6
- 2.3 Running Example . . . 6
- 2.4 The Query Language . . . 8

3 Access Support Relations 9

4 Overview of the GOM Architecture 13

5 The Term Language: A Neutral Query Representation Language 15
- 5.1 The Term Language . . . 15
  - 5.1.1 The Retrieve Operator . . . 15
  - 5.1.2 Utilizing Access Support Relations . . . 15
  - 5.1.3 Creating Temporary Access Support Relations . . . 16
  - 5.1.4 Extending an Existing ASR . . . 16
  - 5.1.5 Joining Two Access Support Relations . . . 16
  - 5.1.6 Scanning Type Extensions . . . 16
  - 5.1.7 Terms . . . 17
- 5.2 Translation of Retrieve Expressions into Term Representation . . . 17
- 5.3 Evaluation of Term Expressions . . . 17

6 Transformation Rules to Optimize Term Expressions 18
- 6.1 Preliminaries . . . 18
- 6.2 Preprocessing . . . 20
- 6.3 Prolonging Path Expressions . . . 20
  - 6.3.1 Prolonging a Linear Path Expression . . . 20
  - 6.3.2 Prolonging a Set-Valued Path Expression . . . 21
- 6.4 Splitting Path Expressions . . . 22
- 6.5 Utilization of ASRs for Single-Target Path Expressions . . . 24
- 6.6 Multi-Target Expressions . . . 25
  - 6.6.1 Bi-Connected Expressions . . . 25
  - 6.6.2 Multipily-Connected Paths . . . 25
- 6.7 Further Operators on Access Support Relations . . . 27
  - 6.7.1 Creating Temporary Access Support Relations . . . 27
  - 6.7.2 Joining Access Support Relations . . . 27
- 6.8 Introduction of Union . . . 28
- 6.9 Moving Selection Predicates Inwards . . . 28
- 6.10 Removal of Range Variables . . . 29
- 6.11 Moving Predicates into the Binding List . . . 30
- 6.12 Introduction of Restriction Predicates . . . 31
- 6.13 Deletion of the Retrieve Operator . . . 32

Abstract

Object-oriented database systems are emerging as the next generation databases for non-standard applications, e.g., VLSI-design, mechanical CAD/CAM, software engineering, etc. While 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 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.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General; H.2.2 [**Database Management**]: Physical Design—access methods; H.2.4 [**Database Management**]: Systems—query processing

General Terms: Algorithms, Design, Performance

Additional Keywords: object-oriented databases, access support, indexing, query optimization, query transformation, term rewriting system, query evaluation, performance enhancement

Advanced Query Processing in Object Bases:

A Comprehensive Approach to Access Support, Query Transformation and Evaluation

Alfons Kemper *Guido Moerkotte*

Interner Bericht Nr. 27/90 ∗ September 1990