[Stonebracker 1985] Stonebraker M., editor. *"The INGRES Papers: Anatomy of a Relational Database System."* Addison Wesley, Reading.

[Ullmann 1987] Ullman J. D. "Database theory: Past and future." In *Proc. ACM Symp. on Principles of Database Systems*, (PODS), pp. 1–10.

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

[von Bültzingsloewen 1990] von Bültzingsloewen G. *"SQL Anfragen: Optimierung für parallele Bearbeitung (Optimization of SQL-queries for parallel processing)."* FZI Berichte Informatik, Springer-Verlag, 1992.

[Zdonik and Maier 1989] Zdonik S. and Maier D. "Fundamentals of object-oriented databases." In Zdonik S. and Maier D., editors, *Readings in Object-Oriented Databases*, pp. 1–32. Morgan-Kaufman Publ. Co.

[Kemper, Kilger, and Moerkotte 1991] Kemper A., Kilger C., and Moerkotte G. "Materialization of functions in object bases." In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pp. 258–268, Denver, CO.

[Kemper, Kilger, and Moerkotte 1992] Kemper A., Kilger C., and Moerkotte G. "Function materialization in object bases: Design, Realization, and Evaluation." To appear in: *IEEE Trans. Knowledge and Data Engineering.*

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

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

[Kemper and Moerkotte 1990c] Kemper A. and Moerkotte G. "Advanced query processing in object bases: A comprehensive approach to access support, query transformation and evaluation." Technical report No. 27/90, University of Karlsruhe, Computer Science Department, 7500 Karlsruhe, F. R. G.

[Kemper and Moerkotte 1991] Kemper A. and Moerkotte G. "A formal model for object-oriented databases." Technical report, University of Karlsruhe.

[Kemper and Moerkotte 1992] Alfons Kemper und Guido Moerkotte. Access Support Relations: An Indexing Method for Object Bases. *Information Systems*, **17**(2), pp. 117–145.

[Kemper et. al 1991] Kemper A., Moerkotte G., Walter H. D., and Zachmann A. "GOM: a strongly typed, persistent object model with polymorphism." In *Proc. of the German Conf. on Databases in Office, Engineering, and Science (BTW)*, pp. 198–217, Kaiserslautern, Springer-Verlag, Informatik-Fachberichte Nr. 270.

[Keßler and Dadam 1991] Keßler U. and Dadam P. "Auswertung komplexer Anfragen an hierarchisch strukturierte Objekte mittels Pfadindexen." In *Proc. of the German Conf. on Databases in Office, Engineering, and Science (BTW)*, pp. 218–237, Kaiserslautern, Springer-Verlag, Informatik-Fachberichte Nr. 270.

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

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

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

Bibliography

[Atkinson et. al 1989] Atkinson M., Bancilhon F., DeWitt D. J., Dittrich K. R., Maier D., and Zdonik S. "The object-oriented database system manifesto." In *Proc. of the Conference on Deductive and Object-Oriented Databases (DOOD)*, pp. 40–57, Kyoto, Japan.

[Banerjee, Kim, and Kim 1988] Banerjee J., Kim W., and Kim K. C. "Queries in oject-oriented databases." In *Proc. Int. Conf. on Data Engineering*, pp. 31–38.

[Becker and Güting 1992] Becker L. and Güting R. H. "Rule-based optimization and query processing in an extensible geometric database system." *ACM Trans. on Database Systems*, **17**(2), pp. 247–303.

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

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

[Cluet and Delobel 1992] Cluet S. and Delobel C. "Towards a unifiction of rewrite based optimization techniques for object-oriented queries." In *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 383–392, San Diego, CA.

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

[Härder 1978] Härder T. "Implementing a generalized access path structure for a relational database system." *ACM Trans. Database Syst.*, **3**(3), pp. 285–298.

[Jarke and Koch 1984] Jarke M. and Koch J. "Query optimization in database systems." *ACM Computing Surveys*, **16**(2), pp. 111–152.

[Jenq et. al 1990] Jenq P., Woelk D., Kim W., and Lee W. "Query processing in distributed ORION." In *Proc. Int. Conf. on Extended Database Technology (EDBT)*, Venice, Italy.

the costs of accessing a non-index relation. A further step would be to also reflect physical properties within the algebra allowing to discard the two-step approach of first algebraic optimization and then index exploitation. There would be two advantages of this approach. First, one could avoid the generation of many alternatives, since indexes are not considered during algebraic optimization and their utilization may depend on its outcome. Second, the query optimizers would have to deal with a single representation—which would be amenable to a well-structured design.

Acknowledgment: This work was supported by the German Research Council DFG under contracts SFB 346 and Ke 401/6-1.

We thank K. Peithner, A. Roemer, and R. Waurig for careful reading of a first draft and for many fruitful discussions. R. Waurig implemented the term-language-based GOM query optimizer. Further, we acknowledge gratefully the fruitful comments of the editors J. C. Freytag, D. Maier, and G. Vossen.

$$\begin{array}{l}\pi_{\lambda x.[x.3,x.5/fetch(\pi_{\lambda y.(y.2)}\sigma_{\lambda y.(y.1=x.2)}(\langle\langle \text{avg_skill}\rangle\rangle))]} \\ \quad (\sigma_{\lambda y.(y.1.\text{Salary}>200000 \wedge y.2.\text{Profit}<0)}([\![\text{EMP.WorksIn.Mgr}]\!]_{can}) \\ \quad \bowtie_{\lambda x.(x.3=x.4)} \\ \quad \sigma_{\lambda y.(y.2<4)}(\langle\langle \text{skill}\rangle\rangle))\end{array} \tag{5.b}$$

Step 6 recombines selections. In our example there is no recombination possible. The algebraic expression (5.b) mostly resembles the result of the previous subsection. For the derivation of the equivalent of (5.b) in the previous subsection, tranformations in all six steps were necessary. Here, the number of relevant steps was diminished to only three steps. Paying attention to the fact that the introduction of GMRs and ASRs accounted for one step in the latter optimization process the gain is still 2/3. Taking into account that the search space increases roughly exponential in the number of transformations to be carried out, the saving above becomes highly relevant.

For the evaluation of the above expressions note that a sequence of type $\pi\sigma\pi(ext(t))$ can be performed by a single *scan* operator applied to the objects of type t.

23.7 Conclusion

In this paper we investigated to what extent the relational query optimization techniques can be exploited for object-oriented query optimization. It turned out that the so-called "semantic integration" through the *most costly normal form* results in a highly structured optimization strategy.

Since it should be clear by now that object-oriented query processing can benefit a lot from relational query processing, the question arises whether relational query processing can benefit from object-oriented query processing as well. Of course, since there is no behavior within relational databases, the methods developed for operator materialization cannot be applied. But things look different for the access support relations. They can be seen as the materialization of n-way joins where the projection takes place on the primary key of each relation. The different extensions of ASRs allow a broad utilization of ASRs, even if the query to be optimized does not span all n joins.

Further, it should be explored whether every index access can be expressed as an access to a special relation with a cost possibly different from

This step might again be called "subscript driven", which is unavoidable since the equivalence of path expressions and joins has to be expressed. We have a flow of the relevant information from the joins to the subscripts. Obviously, this step complements the creation of the most costly normal form.

There is no path to be reassembled for (3.a.2) but we can eliminate the redundant access to the extension of *MANAGER*, which results in:

$$
\begin{array}{l}
\pi_{\lambda x.[x.4,x.8/x.6]} \\
\quad \sigma_{\lambda y.(y.1=y.3)} \\
\quad\quad \sigma_{\lambda y.(y.2>200000)} \\
\quad\quad\quad \sigma_{\lambda y.(y.5<0)} \\
\quad\quad\quad\quad \sigma_{\lambda y.(y.8<4)} \\
\quad\quad\quad\quad\quad \sigma_{\lambda y.(y.3=y.9)} \\
\quad\quad\quad\quad\quad\quad \sigma_{\lambda y.(y.4=y.7)} \\
\quad\quad\quad\quad\quad\quad\quad \pi_{\lambda z.[z.4.\mathrm{WorksIn},z.4.\mathrm{Salary},z.3,z.3.\mathrm{Mgr},z.3.\mathrm{Profit},z.6,z.1,z.2,z.5]} \\
\quad\quad\quad\quad\quad\quad\quad\quad (\langle\langle \mathrm{skill} \rangle\rangle \bowtie ext(\mathrm{DEPT}) \bowtie ext(\mathrm{EMP}) \bowtie \langle\langle \mathrm{avg_skill} \rangle\rangle)
\end{array}
\qquad (4.a.2)
$$

There is no path to be reassembled nor are there any redundant joins in (3.b). Let (4.b) equal (3.b).

Step 5 This step consists of pushing the selections inwards and moving the access to the GMR $\langle\langle \mathrm{avg_skill} \rangle\rangle$ to the outer projection.

$$
\begin{array}{l}
\pi_{\lambda x.[x.2,x.3/fetch(\pi_{\lambda y.(y.2)}\sigma_{\lambda y.(y.1=x.1)}(\langle\langle \mathrm{avg_skill} \rangle\rangle))]} \\
\quad ((\sigma_{\lambda y.(y.\mathrm{Profit}<0)} \\
\quad\quad \pi_{\lambda z.(z.\mathrm{WorksIn})} \\
\quad\quad\quad \sigma_{\lambda y.(y.\mathrm{Salary}>200000)}(ext(\mathrm{EMP}))) \\
\quad \bowtie_{\lambda x.(x.1.\mathrm{Mgr}=x.2)} \\
\quad (\sigma_{\lambda y.(y.2<4)}(\langle\langle \mathrm{skill} \rangle\rangle))))
\end{array}
\qquad (5.a.1)
$$

$$
\begin{array}{l}
\pi_{\lambda x.[x.1,x.2/fetch(\pi_{\lambda y.(y.2)}\sigma_{\lambda y.(y.1=x.3)}(\langle\langle \mathrm{avg_skill} \rangle\rangle))]} \\
\quad (\sigma_{\lambda y.(y.2<4)}(\langle\langle \mathrm{skill} \rangle\rangle) \\
\quad \bowtie_{\lambda x.(x.1=x.3.\mathrm{Mgr})} \\
\quad \sigma_{\lambda y.(y.\mathrm{Profit}<0)}(ext(\mathrm{DEPT})) \\
\quad \bowtie_{\lambda x.(x.3=x.4.\mathrm{WorksIn})} \\
\quad \sigma_{\lambda y.(y.\mathrm{Salary}>200000)}(ext(\mathrm{EMP})))
\end{array}
\qquad (5.a.2)
$$

Of course, the last join in the above expression could be reduced to a semijoin utilizing the $\in$ predicate.

$$\pi_{\lambda x.[x.7,x.8/x.6]}\ \sigma_{\lambda y.(y.1=y.3)}\ \sigma_{\lambda y.(y.4=y.7)}\ \sigma_{\lambda y.(y.2>200000)}\ \sigma_{\lambda y.(y.5<0)}\ \sigma_{\lambda y.(y.8<4)}\ \sigma_{\lambda y.(y.3=y.9)}\ \sigma_{\lambda y.(y.7=y.10)}\ \pi_{\lambda z.[z.1.\text{WorksIn},z.1.\text{Salary},z.2,z.2.\text{Mgr},z.2.\text{Profit},z.5,z.3,z.7,z.4,z.6]}\ (ext(\text{EMP}) \bowtie ext(\text{DEPT}) \bowtie ext(\text{MANAGER}) \bowtie \langle\langle\text{avg_skill}\rangle\rangle \bowtie \langle\langle\text{skill}\rangle\rangle) \tag{3.a.1}$$

$$\pi_{\lambda x.[x.7,x.8/x.6]}\ \sigma_{\lambda y.(y.1=y.3)}\ \sigma_{\lambda y.(y.4=y.7)}\ \sigma_{\lambda y.(y.2>200000)}\ \sigma_{\lambda y.(y.5<0)}\ \sigma_{\lambda y.(y.8<4)}\ \sigma_{\lambda y.(y.3=y.9)}\ \sigma_{\lambda y.(y.7=y.10)}\ \pi_{\lambda z.[z.4.\text{WorksIn},z.4.\text{Salary},z.3,z.3.\text{Mgr},z.3.\text{Profit},z.7,z.5,z.2,z.6,z.1]}\ (\langle\langle\text{skill}\rangle\rangle \bowtie ext(\text{DEPT}) \bowtie ext(\text{EMP}) \bowtie ext(\text{MANAGER}) \bowtie \langle\langle\text{avg_skill}\rangle\rangle) \tag{3.a.2}$$

We do not consider any reordering for (2.b), which thus equals (3.b).

Step 4 In query (3.a.1) we reassemble the path *EMP.WorksIn.Mgr* since a forward traversal of references is cheaper than performing the joins. Reassembling the path yields:

$$\pi_{\lambda x.[x.5,x.6/x.4]}\ \sigma_{\lambda y.(y.1>200000)}\ \sigma_{\lambda y.(y.3<0)}\ \sigma_{\lambda y.(y.6<4)}\ \sigma_{\lambda y.(y.2=y.7)}\ \sigma_{\lambda y.(y.5=y.8)}\ \pi_{\lambda z.[z.1.\text{Salary},z.1.\text{WorksIn},z.1.\text{WorksIn.Profit},z.3,z.1.\text{WorksIn.Mgr},z.5,z.2,z.4]}\ (ext(\text{EMP}) \bowtie \langle\langle\text{avg_skill}\rangle\rangle \bowtie \langle\langle\text{skill}\rangle\rangle) \tag{4.a.1}$$

GMRs clearly results in a performance increase whereas this is uncertain for the introduction of the ASR. Then two alternatives are generated:

$$
\begin{array}{l}
\pi_{\lambda x.[x.7,x.8/x.6]} \\
\quad \sigma_{\lambda y.(y.1=y.3)} \\
\quad\quad \sigma_{\lambda y.(y.4=y.7)} \\
\quad\quad\quad \sigma_{\lambda y.(y.2>200000)} \\
\quad\quad\quad\quad \sigma_{\lambda y.(y.5<0)} \\
\quad\quad\quad\quad\quad \sigma_{\lambda y.(y.8<4)} \\
\quad\quad\quad\quad\quad\quad \sigma_{\lambda y.(y.3=y.9)} \\
\quad\quad\quad\quad\quad\quad\quad \sigma_{\lambda y.(y.7=y.10)} \\
\quad\quad\quad\quad\quad\quad\quad\quad \pi_{\lambda z.[z.1.\mathrm{WorksIn},z.1.\mathrm{Salary},z.2,z.2.\mathrm{Mgr},z.2.\mathrm{Profit},z.5,z.3,z.7,z.4,z.6]} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (ext(\mathrm{EMP}) \bowtie ext(\mathrm{DEPT}) \bowtie ext(\mathrm{MANAGER}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \bowtie \langle\langle \mathrm{avg_skill} \rangle\rangle \bowtie \langle\langle \mathrm{skill} \rangle\rangle)
\end{array}
\tag{2.a}
$$

and

$$
\begin{array}{l}
\pi_{\lambda x.[x.5,x.6/x.4]} \\
\quad \sigma_{\lambda y.(y.1>200000)} \\
\quad\quad \sigma_{\lambda y.(y.3<0)} \\
\quad\quad\quad \sigma_{\lambda y.(y.6<4)} \\
\quad\quad\quad\quad \sigma_{\lambda y.(y.2=y.7)} \\
\quad\quad\quad\quad\quad \sigma_{\lambda y.(y.5=y.8)} \\
\quad\quad\quad\quad\quad\quad \pi_{\lambda z.[z.1.\mathrm{Salary},z.2,z.2.\mathrm{Profit},z.5,z.3,z.7,z.4,z.6]} \\
\quad\quad\quad\quad\quad\quad\quad (\llbracket \mathrm{EMP.WorksIn.Mgr} \rrbracket \bowtie \langle\langle \mathrm{avg_skill} \rangle\rangle \bowtie \langle\langle \mathrm{skill} \rangle\rangle)
\end{array}
\tag{2.b}
$$

Within the above subscripts of the inner projection we have ordered the entries of the tuple to be created such that it first contains all necessary information on employees, followed by the information on departments and then the information on managers. Last, the first columns of the generalized materialization relations are included in order to provide the necessary information for the join predicates.

Step 3 We consider for expression (2.a) two different join orderings corresponding to forward traversal and mixed traversal of [Kim et. al 1988] and [Jenq et. al 1990]:

The control strategy we propose is a hybrid of the current GOM optimizer strategy and the one described in [Becker and Güting 1992]:

1. Decompose any operation. For example, selection predicates with **and** are decomposed into two selections and so forth.
2. Introduce access support relations and generalized materialization relations, which may necessitate some join reordering first. If it is not clear that their introduction results in a performance gain, alternatives have to be generated.
3. Order the join operations—possibly generating alternatives.
4. Reassemble the path expressions and remove any redundant joins.
5. Improve the algebraic expression by pushing selections and the projections inwards and removing redundant join expressions.
6. Recombine operations—which is the inverse of the first step.

Again, we illustrate the optimization process starting with the most costly normal form and following the strategy above with the running example. This time, we will also illustrate that generating alternatives — which is the key to any effective optimization procedure — can be performed very systematically.

Step 1 results in:

$$\begin{array}{l}\pi_{\lambda x.[x.7,x.8/x.6]}\\ \quad\sigma_{\lambda y.(y.1=y.3)}\\ \quad\quad\sigma_{\lambda y.(y.4=y.7)}\\ \quad\quad\quad\sigma_{\lambda y.(y.2>200000)}\\ \quad\quad\quad\quad\sigma_{\lambda y.(y.5<0)}\\ \quad\quad\quad\quad\quad\sigma_{\lambda y.(y.8<4)}\\ \quad\quad\quad\quad\quad\quad\pi_{\lambda z.[z.1.\mathrm{WorksIn},z.1.\mathrm{Salary},z.2,z.2.\mathrm{Mgr},z.2.\mathrm{Profit},z.2.\mathrm{avg_skill},z.3,z.3.\mathrm{skill}]}\\ \quad\quad\quad\quad\quad\quad\quad(ext(\mathrm{EMP}) \bowtie ext(\mathrm{DEPT}) \bowtie ext(\mathrm{MANAGER}))\end{array} \tag{1}$$

Step 2 Access support relations are introduced by replacing a sequence of joins of extensions by the appropriate access support relation—if the latter is applicable. For this step there may be some reordering of joins necessary. Further, the selection predicates have to be checked for the occurrence of the correct join predicates. For the example, assume that the introduction of the

then explores several ways down to the cost valleys. If one is not sure about the right way to the deepest valley, alternatives have to be explored. We will adopt this idea for object-oriented query processing and propose a very costly starting point. There exists another argument of equally importance for $\pi\sigma\bowtie$-expressions as starting points: they permit more transformations to be applied easily.

Parts of the idea can already be found in [Cluet and Delobel 1992] where some queries are translated into a $\pi\sigma\bowtie$ expression. In the context of extensible query optimization the idea is also utilized in [Becker and Güting 1992]. Our starting point will exhibit one additional projection directly following the $\bowtie$. Thus, it consists of a sequence $\pi\sigma\pi\bowtie$. First all the extensions whose instances are needed for the query evaluation are joined. More specifically, the cross product is taken. Then a projection follows which enhances each tuple of the resulting relation by further information needed in order to evaluate the selection predicate solely on the basis of this result. Then the selection follows evaluating the selection predicate of the query on each tuple resulting from the projection. Last, a projection onto the required result takes place. We call the thus derived query representation the *most costly normal form (MCNF).*

For our example we get

$$\pi_{\lambda x.[x.7, x.8/x.6]}\ \sigma_{\lambda y.(y.1=y.3 \wedge y.4=y.7 \wedge y.2>200000 \wedge y.5<0 \wedge y.8<4)}\ \pi_{\lambda z.[z.1.\text{WorksIn}, z.1.\text{Salary}, z.2, z.2.\text{Mgr}, z.2.\text{Profit}, z.2.\text{avg_skill}, z.3, z.3.\text{skill}]}\ (ext(\text{EMP}) \bowtie ext(\text{DEPT}) \bowtie ext(\text{MANAGER}))$$

with all join predicates being $\lambda x.true$. For the rest of this section we will stick to this convention.

23.6.3 Optimizer Strategy and Generating Alternatives

This normal form allows a clear optimizer strategy, and thus a clear optimizer design. Note that we are not talking about heuristics that are used to prune the search space of alternatives. Instead, a strategy divides the optimization process into several steps performed sequentially or under any other control procedure. Whenever a step may generate alternatives, heuristics may be useful to favor or discard some of them.

Another aspect treated within this section is the systematic generation of alternatives; a problem often neglected in object-oriented query processing.

$(\sigma_{\lambda x.(x.2<4)}$
$(\langle\langle \text{skill} \rangle\rangle)))$

We will now have a closer look at both optimizations, especially at the one performed within the algebra. Here, there is one important point to note: the optimization rules are mostly performed within the selection predicate of the selection operator or within the projection operations within the projection operator, i.e., in both cases the rewriting takes place within the subscripts of the algebraic operators. This subscript rewriting is not the clean algebraic optimization we are used to in the relational context. One goal of the next subsection is to remedy this situation.

Another observation is that often new domains (extensions) are introduced together with variables which are subsequently dropped and then reintroduced again. This seems to be a rather unstructured approach which will be remedied in the next subsection, as well.

23.6.2 The Most Costly Normal Form

In the previous section we have seen that algebraic optimization is not as readable as expected. One could argue that readability is no problem since the terms are only to be read by the optimizer; but, nevertheless, there exist reasons why readability is important. Not only the optimizer becomes harder to debug and errors more difficult to find, but also the search for optimization rules and the organization of the optimizer, i.e., the development of its strategy and pruning heuristics, is hindered.

As seen in the previous section, one of the main drawbacks is that the optimization takes place mostly within the subscripts. This rewriting of subscripts will be overcome by selecting another starting point, the term derived initially from the translation of the query into the algebra. This choice is crucial for the optimization process. A good starting point can be seen as a good heuristic to organize the optimization.

In object-oriented query processing it seems quite common to translate the query into an internal representation as close to the original query as possible ([Banerjee, Kim, and Kim 1988], [Cluet and Delobel 1992], and also [Kemper and Moerkotte 1990b]). This observation is also true for relational query processing where, for example, an SQL query is translated into a $\pi\sigma\bowtie$-expression. But this representation exhibits another property that the initial internal representation of object-oriented queries lacks: it is a very costly term. Considering the following metaphor this seems a good heuristic. Optimization starts from the highest mountain from a cost point of view and

Moving projections inwards and removing retrieves Again, this step consists of several rule applications. Within the term language several parts of the selection predicate can be moved to the binding list, resulting in smaller bindings to be explored if executed by a nested loop. This step is always required for variables that are bound to "infinite" sorts such as integer. If the selection predicate becomes empty (true) by this step, the outer retrieve can be removed and the resulting term representing the query is gained by moving the projection of the retrieve to the projection of the expression within the binding list.

Within our example there exists only one term within the selection predicate. It expresses variable bindings for all variables bound within the binding list. Its removal from the binding list results in a selection predicate equivalent to true. Since it is possible to move the projection of the outer retrieve to the expression within the binding list, the retrieve can totally be removed. We get

```
(join (getasr [[EMP.WorksIn.Mgr]]
              :S (and (< (path #2 Profit) 0)
                      (> (path #1 Salary) 200000))
              :P (#2 #3))
      (getgmr ⟨⟨skill⟩⟩
              :S (< #2 4)
              :P (#1 #2))
      :J (= #2 #3)
      :P (#2 (/ #4 (fetch (getgmr ⟨⟨avg_skill⟩⟩
                                  :B (d #1)
                                  :S (= d #1)
                                  :P #2)))))
```

Here, we see an example for the use of the binding clause **:B** within a non-**retrieve** term. Its usage is necessary here since the column number #1 would otherwise be ambiguous.

Within the algebra the equivalent of the expressions above replaces the cross product of the extensions by the access support relation access as performed within the selection operation of the selection. The result is as follows:

$$\pi_{\lambda x.[x.2,(x.4/fetch(\pi_{\lambda y.(y.2)}\sigma_{\lambda y.(y.1=x.1)}(\langle\langle \text{avg_skill}\rangle\rangle)))]}$$
$$((\pi_{\lambda x.[x.2,x.3]}$$
$$\sigma_{\lambda x.(x.1.\text{Salary}>200000\ \textbf{and}\ x.2.\text{Profit}<0)}$$
$$([\![\text{EMP.WorksIn.Mgr}]\!]))$$
$$\bowtie_{\lambda x.(x.2=x.3)}$$

where

$$P \equiv \lambda x.[x.3, (x.4/fetch(\pi_{\lambda y.(y.2)}\sigma_{\lambda y.(y.1=x.2)}(\langle\langle \text{avg_skill}\rangle\rangle)))]$$

$$S \equiv \lambda x.(x \in \pi_{\lambda x.[x.1,x.2,x.3,x.5]}([\![\text{EMP.WorksIn.Mgr}]\!] \bowtie_{\lambda x.(x.3=x.4)} \langle\langle \text{skill}\rangle\rangle)$$
$$\textbf{and } x.2.\text{Profit} < 0$$
$$\textbf{and } x.1.\text{Salary} > 200000$$
$$\textbf{and } x.4 < 4)$$

Moving selection predicates inwards We now come to the more conventional optimization techniques as already employed within the relational optimization process. Moving selection predicates inwards is one of the common heuristics that can often be applied successfully. Applying it to the example yields:

```
(retrieve :B ((m MANAGER) (d DEPT) (s INT))
          :S (in (d m s) (join (getasr [[EMP.WorksIn.Mgr]]
                                       :S (and (< (path #2 Profit) 0)
                                               (> (path #1 Salary) 200000))
                                       :P (#2 #3))
                               (getgmr <<skill>>
                                       :S (< #2 4)
                                       :P (#1 #2))
                               :J (= #2 #3)    !! revised column numbering
                               :P (#1 #2 #4)))
          :P (m (/ s (fetch (getgmr <<avg_skill>>
                                    :S (= d #1)
                                    :P #2)))))
```

and

$$\pi_P\sigma_S(ext(\text{DEPT}) \bowtie ext(\text{MANAGER}) \bowtie ext(\text{INT}))$$

where

$$P \equiv \lambda x.[x.2, (x.3/fetch(\pi_{\lambda y.(y.2)}\sigma_{\lambda y.(y.1=x.1)}(\langle\langle \text{avg_skill}\rangle\rangle)))]$$

$$S \equiv \lambda x.(x \in \pi_{\lambda x.[x.1,x.2,x.4]}((\pi_{\lambda x.[x.2,x.3]}\sigma_{\lambda x.(x.1.\text{Salary}>200000 \textbf{ and } x.2.\text{Profit}<0)}([\![\text{EMP.WorksIn.Mgr}]\!])) \bowtie_{\lambda x.(x.2=x.3)} (\sigma_{\lambda x.(x.2<4)}(\langle\langle \text{skill}\rangle\rangle))))$$

where

$P \equiv \lambda x.[x.2, (x.2.\text{skill}/x.1.\text{WorksIn.avg_skill})]$

$S \equiv \lambda x.(x \in \pi_{\lambda x.[x.1, x.3]}([\![\text{EMP.WorksIn.Mgr}]\!])$ **and**
$x.1.\text{WorksIn.Profit} < 0$ **and**
$x.1.\text{Salary} > 200000$ **and**
$x.2.\text{skill} < 4)$

Utilization of GMRs After having introduced the access support relation, the replacement of function invocations by GMR accesses is performed. The ordering among these two steps is arbitrary since they do not interfere. For the introduction of the GMR access two variables are necessary. The first, d, iterates over the departments whereas the second, s, holds as a result the skill of a manager. Introducing the variables and replacing the calls to *skill* and *avg_skill* by GMR accesses yields—after several elementary steps—the following term expression:

```
(retrieve :B ((e EMP) (m MANAGER) (d DEPT) (s INT))
          :S (and (in (e d m s) (join (getasr [[EMP.WorksIn.Mgr]]
                                      :S (true)
                                      :P (#1 #2 #3))
                                      (getgmr ⟨⟨skill⟩⟩
                                              :S (true)
                                              :P (#1 #2))
                                      :J (= #3 #4)
                                      :P (#1 #2 #3 #5)))
                  (< (path d Profit) 0)
                  (> (path e Salary) 200000)
                  (< s 4))
          :P (m (/ s (fetch (getgmr ⟨⟨avg_skill⟩⟩
                                    :S (= #1 d)
                                    :P #2)))))
```

Note that the binding of s to INT does not result in any problem if the term is processed further. Actual execution of the term above would run into problems due to the huge range of (computer representable) integers. Actually, it is impossible to directly mimic this step within the algebra. Nevertheless, ignoring the problems we will introduce a cross product with the extension of the integers and get rid of this construct in later steps. Thus, we obtain:

$\pi_P \sigma_S(ext(\text{EMP}) \bowtie ext(\text{DEPT}) \bowtie ext(\text{MANAGER}) \bowtie ext(\text{INT}))$

:S clause. This concentration prevents the introduction of the applicable access support relation $[\![\text{EMP.WorksIn.Mgr}]\!]_{can}$ into the projection (**:P**) clause of the term above. Within the selection clause the access support relation is not yet applicable since the call to the method *skill* immediately succeeds the path. Thus, we split at this location:

```
(retrieve :B ((e EMP) (m MANAGER))
          :S (and (< (path e WorksIn Profit) 0)
                  (> (path e Salary) 200000)
                  (= (path e WorksIn Mgr) m)
                  (< m.skill 4))
          :P (m (/ m.skill (path e WorksIn).avg_skill)))
```

Contrary to the extending step where a variable could be eliminated from the term, we had to introduce a new variable m to hold the manager obtained at "the end" of the path (**path** e WorksIn Mgr) in order to be able to compute his or her skill, subsequently.

Within the algebraic representation, this step results in the following:

$$\pi_{\lambda x.[x.2,(x.2.\text{skill}/x.1.\text{WorksIn.avg_skill})]} \sigma_{\lambda x.(x.2=x.1.\text{WorksIn.Mgr } \textbf{and } x.1.\text{WorksIn.Profit}<0 \textbf{ and } x.1.\text{Salary}>200000 \textbf{ and } x.2.\text{skill}<4)} (ext(\text{EMP}) \bowtie ext(\text{MANAGER}))$$

Again, this step constitutes a "subscript driven" transformation.

Utilization of the ASR Due to the above two steps we are now able to replace the path traversal (**path** e WorksIn Mgr) by an access to the access support relation $[\![\text{EMP.WorksIn.Mgr}]\!]_{can}$. For the term representation this replacement results in:

```
(retrieve :B ((e EMP) (m MANAGER))
          :S (and (in (e m) (getasr [[EMP.WorksIn.Mgr]]
                                    :S (true) :P (#1 #3)))
                  (< (path e WorksIn Profit) 0)
                  (> (path e Salary) 200000)
                  (< m.skill 4))
          :P (m (/ m.skill (path e WorksIn).avg_skill)))
```

And in the algebra we obtain:

$$\pi_P \sigma_S (ext(\text{EMP}) \bowtie ext(\text{MANAGER}))$$

omit the subscript "*can*" within the ASR-specification. Further, the optimization process will take advantage of the generalized materialization relations ⟨⟨skill⟩⟩ and ⟨⟨avg_skill⟩⟩.

The first phase of the optimization process consists of the two steps of extending and splitting path expressions such that the given access support relation becomes applicable (see Def. 23.5). Loosely speaking, these steps cut and paste path expressions until the materialized path *EMP.WorksIn.Mgr* is derived.

Extending paths/variable removal In the term representation extending leads to the following (equivalent) term expression:

```
(retrieve :B ((e EMP))
          :S (and (< (path e WorksIn Profit) 0)
                  (> (path e Salary) 200000)
                  (< (path e WorksIn Mgr).skill 4))
          :P ((path e WorksIn Mgr)
              (/ (path e WorksIn Mgr).skill (path e WorksIn).avg_skill)))
```

Note that not only all occurrences of d within the **:S** clause have been replaced by (**path** e WorksIn) but also the occurrence of d within the **:B** clause has been removed. This step could, of course, be deferred until the end of this optimization phase.

The same transformation step applied to the algebraic representation yields:

$$\pi_{\lambda x.[x.\text{WorksIn.Mgr},(x.\text{WorksIn.Mgr.skill}/x.\text{WorksIn.avg_skill})]} \; \sigma_{\lambda x.(x.\text{WorksIn.Profit}<0 \textbf{ and } x.\text{Salary}>200000 \textbf{ and } x.\text{WorksIn.Mgr.skill}<4)} \; (ext(\text{EMP}))$$

If the removal of d had been deferred the whole transformation would have taken place within the subscripts. Nevertheless, we can speak of a "subscript driven" optimization step since the transformations at the algebra level are only adjustments, whereas the relevant information and the triggering part are contained in the subscript. We will see that this finding is also true for the subsequent optimization steps. It is important to note that this is a phenomenon not present only within the GOM optimizer.

Splitting and variable insertion Remember that the transformation process first concentrates as much as possible on the selection predicate, i.e., the

relevant to the example is given. The optimization process for the running example will pass through 4 phases:

1. extending and splitting path expressions such that the available access support relations become applicable,
2. introducing the applicable access support relations and generalized materialization relations,
3. moving selections inwards,
4. moving projections inwards.

Each phase consists of possibly several steps. For example, Phase 1 consists of the two steps extending and splitting. Each step has a certain set of rules attached to it. Since the statement of the rules requires a good deal of formalism to be introduced, only the results after each application are illustrated—we trust that the reader is able to roughly figure out how the rules are specified. Note that several rules may be applied within one step. For further details we refer to the original literature.

Another characteristics of the GOM optimization process is that rewriting first takes place at inner terms and then proceeds to outer terms. In particular, the rule application first concentrates as much as possible on the selection clause (**:S**) of the **retrieve** term.

Original query We restate the original query in term and algebraic representation:

```
(retrieve :B ((e EMP) (d DEPT))
          :S (and (= d (path e WorksIn))
                  (< (path d Profit) 0)
                  (> (path e Salary) 200000)
                  (< (path d Mgr).skill 4))
          :P ((path d Mgr) (/ (path d Mgr).skill  d.avg_skill)))
```

$$\pi_{\lambda x.[x.2.\text{Mgr}, x.2.\text{Mgr.skill}/x.2.\text{avg_skill}]}\ \sigma_{\lambda x.(x.2=x.1.\text{WorksIn}\ \textbf{and}\ x.2.\text{Profit}<0\ \textbf{and}\ x.1.\text{Salary}>200000\ \textbf{and}\ x.2.\text{Mgr.skill}<4)}\ (ext(\text{EMP}) \bowtie ext(\text{DEPT}))$$

For the subsequent optimization we assume the existence of the access support relation $[\![\text{EMP.WorksIn.Mgr}]\!]_{can}$. For simplicity of notation we will

possibilities to do so. First, one could perform a technical integration by simply changing the internal representation of an existing object-oriented query optimizer to an algebra, transforming its rewrite rules to the algebraic representation and then adding the proposed rewrite rules for relational query optimization to its rule base. This approach will be pursued within the next subsection. A consequence of this approach is that one must be able to mirror the object-oriented optimization within the algebra. We will do so and show the result. However, we will identify several disadvantages of this approach preventing the useful integration of relational optimization knowledge. One problem occurring is that the relational algebra transformations treat all the expressions which are subscripts to the algebra operators as black boxes. Therefore, additional transformations are needed to transform the subscripts into algebra operators or to transform the subscripts themselves. We will refer to this approach as the *syntactic integration* of relational and object-oriented query processing.

In the second approach, the integration does not take place at the technical level but must first take place at the "intellectual" level. By this we mean that the designer of the object-oriented query optimizer must first learn—at an abstract level—some basic ideas of relational query optimization in order to grasp its spirit. Next one is asked to transform this spirit to object-oriented query optimization. From then on the integration of both approaches into one useful approach to object-oriented query processing is straightforward. As we will see this approach results in a very clear optimization process allowing a clean optimizer and optimizer strategy design. We call this approach the *semantic integration*. This second experiment is carried out in Subsections 23.6.2 and 23.6.3.

23.6.1 Mimicking Term-Based Optimization Within the Algebra

In this section it is shown that it is possible to mimic the object-oriented query optimization process performed within the GOM optimizer within the algebra. This indicates the possibility of integrating relational and object-oriented query processing.

In order to facilitate the understanding of the optimization process it is useful to state the overall optimization strategy. The GOM optimizer strategy as currently employed is represented as a heuristic net. It is interpreted at optimization time. This interpretation enhances flexibility. A full account of this subject is beyond the scope of the paper, hence, only the restricted part

is defined as

$$\pi_{\lambda x.proj_list'}(\sigma_{\lambda x.sel_pred'}([\![t_0.A_1.\cdots.A_n]\!]_X))$$

where the expressions *proj_list'* and *sel_pred'* are modified such that they reflect the correct redirection of attribute accesses, i.e., each attribute access to the access support relation is preceded by an x.

In fact, we are a little bit sloppy in the usage of the getasr operator since in the original version it is able to cope with the actual physical representation including the utilization of indexes on the accessed ASR. The **getgmr** operator, which retrieves data from a generalized materialization relation (GMR), is defined analogously.

The Running Example

The translation of our example query into the GOM term language results in the following initial term:

```
(retrieve :B ((e EMP) (d DEPT))
          :S (and (= d (path e WorksIn))
                  (< (path d Profit) 0)
                  (> (path e Salary) 200000)
                  (< (path d Mgr).skill 4))
          :P ((path d Mgr) (/ (path d Mgr).skill  d.avg_skill)))
```

Note that *skill* is not an attribute but instead is an operation to be computed. Thus it cannot be included in the **path**-expression. The same holds for *avg_skill*.

23.6 The Optimization Process

As should be clear from the definition of our algebra, all the techniques developed for relational query optimization can be considered for object-oriented query optimization as well. Note that it is not the goal of this paper to review the optimization techniques for relational query optimization. (For a more thorough survey see, e.g., [von Bültzingsloewen 1990, Freytag 1987, Jarke and Koch 1984, Lehnert 1988].)

Instead, we come to the most critical experiments of this paper. The main question we are trying to answer by the following two experiments is how to provide a basis for exploiting the knowledge of relational query optimization for object-oriented query optimization. Obviously, there exist two

Path expressions constitute a major difference between relational and object-oriented databases. In the former they are not supported whereas in the latter they are among the most basic and most frequently used expressions. Thus, the above equivalence is not present in relational databases but can be exploited for optimization purposes in object-oriented databases.

On the other hand, note that the standard relational technique of reordering joins applied to the second expression coincides with optimizations proposed in several papers ([Banerjee, Kim, and Kim 1988, Jenq et. al 1990, Kim et. al 1988]). Although the latter is more general in that it can deal with arbitrary or generalized path expressions that are not just linear chains of attributes, the stated optimization techniques can still be derived from relational join reordering as we will see in Section 23.6.2.

Retrieve Term The **retrieve** operator is often seen as one of the reasons for the term language being low level as opposed to algebras. The reason is that the proposed implementation of the **retrieve** operator's evaluation is a nested loop. Nevertheless, the definition of this operator can be given in purely algebraic terms. More specifically, we can define

(**retrieve :B** $((x_1\ t_1)\ldots(x_n\ t_n))$ **:S** *sel_pred* **:P** *proj_list*)

as

$$\pi_{\lambda x.proj_list'}\sigma_{\lambda x.sel_pred'}(ext(t_1)\bowtie\cdots\bowtie ext(t_n))$$

where $proj_list'$ and sel_pred' are derived from *proj_list* and *sel_pred*, respectively, by modifying them such that the accesses to variables are replaced by accesses to components of the tuples obtained by the joins.

In the definition above we have restricted ourselves to the case where the binding list contains elements that are tuples of the form (*variable type*). In general, the second component can be any term—even a single-valued expression—to which the variable is bound. This binding mechanism can also serve a similar purpose as in SQL: resolving binding ambiguities that might occur, e.g., in the case of nesting access support relation accesses. If necessary, any operator in the term language can be enhanced by a binding clause. An example will be shown later in the paper.

Getasr Term The **getasr** operator

(**getasr** $[\![t_0.A_1.\cdots.A_n]\!]_X$ **:S** *sel_pred* **:P** *proj_list*)

will be overloaded (denoted by $\bowtie$) and always be processed from left to right. Overloading is resolved by first applying $\bowtie$. If its application is not possible we apply $\bowtie$, last we apply $\bowtie$. We will abbreviate $\bowtie_{\lambda x.true}$ by $\bowtie$.

The Running Example

Our example query can be translated into the following algebraic expression:

$$\pi_{\lambda x.[x.2.\text{Mgr}, x.2.\text{Mgr.skill}/x.2.\text{avg_skill}]}$$
$$\sigma_{\lambda x.(x.2=x.1.\text{WorksIn}\ \textbf{and}\ x.2.\text{Profit}<0\ \textbf{and}\ x.1.\text{Salary}>200000\ \textbf{and}\ x.2.\text{Mgr.skill}<4)}$$
$$(ext(\text{EMP}) \bowtie ext(\text{DEPT}))$$

23.5.2 The GOM Term Language

This section reviews the GOM term language and gives the algebraic definitions of its most important constituents. Building upon the term language, a set of transformation rules has been defined and a rule-based optimizer was implemented ([Kemper and Moerkotte 1990b, Kemper and Moerkotte 1990c]).

The Term Language Operators

Path Term A path term is used to retrieve the attribute value at the end of a path. Thus for v of type t_0, $v.A_1$ of type t_1, ..., $v.A_1.\cdots.A_n$ of type t_n the **path** term

$$(\textbf{path}\ v\ A_1\ \ldots A_n)$$

is defined as

$$fetch(\pi_{\lambda x.(x.A_1.\cdots.A_n)}(\{v\})).$$

This expression is equivalent to

$$fetch($$
$$\pi_{\lambda x.(x.n+1)}$$
$$\sigma_{\lambda x.(x.1=v)}$$
$$(ext(t_0) \bowtie_{\lambda x.(x.1.A_1=x.2)} ext(t_1) \bowtie_{\lambda x.(x.2.A_2=x.3)} \cdots$$
$$\bowtie_{\lambda x.(x.n.A_n=x.(n+1))} ext(t_n)))$$

show that it is even possible to express the utilization of the above access support methods within the algebra. In particular, we show how to incorporate ASRs and GMRs in the next section where optimization in the presence of these is discussed. Since we will not review all the relational optimization techniques we restrict the part of our algebra represented here to the minimum. It should be obvious how to include special operators to deal with, for example, NULL values and quantifiers, by incorporating the appropriate operators from the relational area.

Definitions

Aside from the basic operations to build tuple constants ("$[\cdot]$") and set constants ("$\{\cdot\}$"), the operators for tuple access ($x.n$ retrieves the n-th attribute of tuple x) and for set manipulation ("$\cup$", "$\setminus$", "$\cap$", "*fetch*", which retrieves the element of a singleton set) the algebra consists of the usual *selection*, *projection*, and *join* operations as well as a new operator *ext* that retrieves the extension, i.e., the set of object identifiers of the instances of the given type. Their declarations as polymorphic operators are as follows ([von Bültzingsloewen 1990]):

$$
\begin{array}{lcll}
\sigma_{g:\tau_1 \to BOOL} & : & \{\tau_1\} & \to \{\tau_1\} \\
\pi_{g:\tau_1 \to \tau_2} & : & \{\tau_1\} & \to \{\tau_2\} \\
\bowtie\!\!\!\!\bullet_{g:[\tau_1,\tau_2] \to BOOL} & : & \{\tau_1\},\{\tau_2\} & \to \{[\tau_1,\tau_2]\} \\
\bowtie\!\!\!\!\bullet_{g:\tau_1 \circ [\tau_2] \to BOOL} & : & \{\tau_1\},\{\tau_2\} & \to \{\tau_1 \circ [\tau_2]\} \\
\bowtie\!\!\!\!\bullet_{g:\tau_1 \circ \tau_2 \to BOOL} & : & \{\tau_1\},\{\tau_2\} & \to \{\tau_1 \circ \tau_2\} \\
 & & & \\
ext(\tau) & : & & \to \{\tau\}
\end{array}
$$

Here, τ and τ_i are either sort or type variables.

The functions appearing in the subscripts of the operators will be written using λ-notation. Note that, contrary to the relational algebra, the operators appearing within the subscripts are allowed to be general functions defined as an λ-expression, and not just those accessing attribute values. As a consequence the projection operator resembles a general *map* operator.

The $\bowtie$ operations will be used to "join" the extensions of types. Since these extensions are not tuples, but instead sets of object identifiers we apply $\bowtie\!\!\!\!\bullet$ in order to join two of them. This results in a set of tuples with two components. Joining a third extension to this result requires the application of $\bowtie\!\!\!\!\bullet$. This way we can perform in a left-to-right order a join of any number of extensions. Sometimes the ordinary join $\bowtie$ is still needed. All three joins

attributes $f_1, \ldots, f_m$ store the results or, if the result type is complex, references to the result objects. In this paper we restrict our discussion to functions having complex argument types and atomic result types. However, our concepts scale up to arbitrary functions [Kemper, Kilger, and Moerkotte 1992]. To keep the overhead in case of an update as low as possible, special attention has to be paid to the invalidation and rematerialization procedures [Kemper, Kilger, and Moerkotte 1991, Kemper, Kilger, and Moerkotte 1992].

The Running Example

Consider the database extension in Figure 23.2. Recall functions *skill* and *avg_skill* associated with the object types *EMP* and *DEPT*, respectively:

skill: $EMP \rightarrow INT$
avg_skill: $DEPT \rightarrow FLOAT$

Their computation requires to iterate over the whole sets of employees and departments, hence there computation is quite complex and materialization pays off.

The extensions of the GMRs $\langle\langle skill \rangle\rangle$ and $\langle\langle avg_skill \rangle\rangle$ with all results valid are depicted below.

$\langle\langle skill \rangle\rangle$

O_1:*EMP*	*skill*:*INT*
id_1	4
...	...
id_8	5
id_9	3
id_{10}	10

$\langle\langle avg_skill \rangle\rangle$

O_1:*DEPT*	*avg_skill*:*FLOAT*
id_5	5.32
id_6	7.50
id_7	6.77

Using this information together with the information in Figure 23.2 we are able to give an answer tuple for our example query: $[id_9, 0.4]$.

23.5 The (Internal) Query Representation Formats

23.5.1 The Algebra

The main motivation for introducing the object algebra—which constitutes a slightly enhanced relational algebra—is to show that GOMql queries can be captured by it and, since it consists mainly of the relational operators, that relational optimization strategies can be applied as is. Further, we will

$[\![\mathbf{EMP.WorksIn.Mgr}]\!]_{\mathbf{can}}$		
OID_{EMP}	OID_{DEPT}	$OID_{MANAGER}$
id_1	id_5	id_8
id_2	id_5	id_8
id_3	id_6	id_9
id_4	id_6	id_9
...	...	...

This extension contains all complete paths corresponding to the underlying path expression.

23.4.2 Function Materialization

In this subsection we describe one (further) piece in the mosaic of access support structures that we incorporated in our experimental object base system GOM: the *materialization of functions*, i.e., the precomputation of function results (see [Kemper, Kilger, and Moerkotte 1991] and the more detailed presentation in [Kemper, Kilger, and Moerkotte 1992] for further information).

Materialization—just like indexing—is based on the assumption that the precomputed results are eventually utilized in the evaluation of some associative data access. Function materialization is a dual approach to *Access Support Relations*, which constitute materializations of heavily traversed path expressions that relate objects along attribute chains.

Definitions

The main definition needed within this paper concerned with the optimization of GOMql queries utilizing materialized functions is the one for *generalized materialization relations*:

DEFINITION 23.6 *Generalized Materialization Relations (GMRs)*

Let $t_1, \ldots, t_n, t'_1, \ldots, t'_m$ *be types and* $f_1, \ldots, f_m$ *side-effect-free functions with* $f_j : t_1, \ldots, t_n \rightarrow t'_j$ *for* $1 \leq j \leq m$. *The* generalized materialization relation $\langle\langle f_1, \ldots, f_m \rangle\rangle$ *for the functions* $f_1, \ldots, f_m$ *is of arity* $n + m$ *and has the following form:*

$$[O_1{:}t_1, \ldots, O_n{:}t_n, f_1{:}t'_1, \ldots, f_m{:}t'_m]$$

The attributes $O_1, \ldots, O_n$ store the arguments, i.e., values if the argument type is atomic or references to objects if the argument type is complex; the

$$[\![t_0.A_1.\cdots.A_n]\!]_{full} := [\![t_0.A_1]\!] \mathbin{⟗} \cdots \mathbin{⟗} [\![t_{n-1}.A_n]\!]$$
$$[\![t_0.A_1.\cdots.A_n]\!]_{left} := (\cdots([\![t_0.A_1]\!] \mathbin{⟗} [\![t_1.A_2]\!])\cdots \mathbin{⟗} [\![t_{n-1}.A_n]\!])$$
$$[\![t_0.A_1.\cdots.A_n]\!]_{right} := ([\![t_0.A_1]\!] \mathbin{⟕} \cdots([\![t_{n-2}.A_{n-1}]\!] \mathbin{⟕} [\![t_{n-1}.A_n]\!])\cdots)$$

Aside from extensions, we also allow decomposition of access support relations which is not treated here. We further do not discuss the storage structure of access support relations but again refer to the original literature.

Let us now define the *applicability* of a given access support relation for evaluating some path expression:

DEFINITION 23.5 *Applicability*

An access support relation $[\![t_0.A_1.\cdots.A_n]\!]_X$ *under extension* X *is* applicable *to 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.*

The Running Example

Let us reconsider the path expression of our schema *Company* (now we indicate the types of the subpaths by the underbraces):

$$P \equiv \underbrace{\underbrace{EMP.WorksIn}_{DEPT}.Mgr}_{MANAGER}$$

For this path the canonical extension $[\![EMP.WorksIn.Mgr]\!]_{can}$ looks as follows:

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 $[\![t_0.A_1]\!]$, ..., $[\![t_{n-1}.A_n]\!]$.

DEFINITION 23.3 *Auxiliary Binary Relations*

For each A_i $(1 \leq i \leq n)$, i.e., 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 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.*

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 which must not necessarily lead to an object in t_n, but possibly ending in a *NULL*-value.

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 that 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*-value.

DEFINITION 23.4 *Extensions*

Let $\bowtie$ ($\overset{\rightarrow}{\bowtie}$, $\overset{\leftarrow}{\bowtie}$, $\overset{\leftrightarrow}{\bowtie}$) *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:*

$$[\![t_0.A_1.\cdots.A_n]\!]_{can} := [\![t_0.A_1]\!] \bowtie \ldots \bowtie [\![t_{n-1}.A_n]\!]$$

23.4 Access Support Structures

23.4.1 Access Support Relations

Access support relations (ASRs) have been introduced as an index structure to support the evaluation of *path expressions*. A very special case is reviewed briefly here. More detailed information can be found in the original papers [Kemper and Moerkotte 1990a] and [Kemper and Moerkotte 1992].

Definitions

A path expression has the form

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

where o is a tuple structured object with an attribute A_1 and $o.A_1.\cdots.A_i$ refers to an object or tuple value having an attribute A_{i+1}. More formally:

DEFINITION 23.1 *Path Expression*

Let $t_0, \ldots, t_n$ be (not necessarily distinct) types. A (generic) path expression *on t_0 is an expression $t_0.A_1.\cdots.A_n$ if and only if for each $1 \leq i \leq n$ 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*[1].

Thus we concentrate on the case where none of the path attributes is set valued—even though ASRs in their general form allow set-valued attributes as well ([Kemper and Moerkotte 1990a]).

As we will see, the information contained in a path can be held in a relation. Consequently, we will use relation extensions to represent access paths. The next definition maps a given path expression to the declaration of the underlying access support relation.

DEFINITION 23.2 *Access Support Relation (ASR)*

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 23.1 for $(0 \leq i \leq n)$. If t_n is an atomic type

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

A *predicate* is either an application of a comparison operator to (a) term(s) or an application of a user-defined function returning a Boolean value. From predicates we build *Boolean expressions* by applying the usual Boolean connectives AND, OR, and NOT.

We are now ready to define our query language. In GOM a query is a **retrieve** expression returning a complex value, i.e., a relation.

Let x_i be variables, T_i type names, e_j expressions, and S a Boolean expression called the *selection predicate*. Then, a query has the following form:

range $x_1 : T_1, \ldots, x_n : T_n$
retrieve $e_1, \ldots, e_m$
where $S(x_1, \ldots, x_n)$

In fact, GOMql is more powerful in that the T_i are not restricted to type names only but may, instead, represent general set typed or set valued expressions [Kemper and Moerkotte 1991]. Note, however, that our current implementation of the GOM query language facilitates single-target queries only ([Kemper and Moerkotte 1990b]). We have chosen the query language above because it is well suited to show its mapping to the (slightly extended) relational algebra and the optimization within the algebra. The T_i could also be lists but so far we have only considered sets as results.

23.3.2 The Running Example

In our example query we want to retrieve—and subsequently fire—the managers of departments that generate losses and, at the same time, pay at least one of their employees an exorbitant salary (exceeding 200000). Since we do not want to fire highly skilled managers we additionally require that the skill of the managers is less than 4. In addition to the manager, i.e., the manager OID, we want to retrieve the manager's skill relative to the average of skill (*avg_skill*) of all employees in his or her department. In GOMql the query can be stated as follows:

range e : EMP, d : DEPT
retrieve d.Mgr, d.Mgr.skill/d.avg_skill
where $d = e$.WorksIn **and**
d.Profit < 0 **and**
e.Salary > 200000 **and**
d.Mgr.skill < 4

maintained uni-directionally in GOM—as in almost all other object models, e.g., O2, GemStone, etc. 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 in the referencing attribute or variable.

23.3 Query language

For our object model we developed a QUEL-like [Stonebracker 1985] query language along the lines of the EXCESS object query language that was designed as the declarative query language for the EXTRA object model that is described in [Carey, DeWitt, and Vandenberg 1988]. We assume that every query returns a relation (see [Ullmann 1987]). This relation is a complex value that may then be assigned as a value to an object. More detailed information on this topic and a more thorough treatment of the semantics of our object model as well as of the query language can be found in [Kemper and Moerkotte 1991].

23.3.1 Abstract syntax

As complex sorts we only have tuples denoted by $[\tau_1, \ldots, \tau_n]$ for $n > 0$ and τ_i being base sorts, sets denoted by $\{\tau\}$, and lists denoted by $< \tau >$ where τ is a base or a tuple sort (see [Kemper and Moerkotte 1991] for more details). In order not to run into any problems concerning the uniqueness of attribute names, we assume a tuple to be ordered and use numbers for their identification. On tuple sorts the operator "$\circ$" denotes concatenation. This operator will be used for both function composition and tuple concatenation. This overloading can easily be resolved by looking at the argument types.

To build *terms*, there exist basic functions for arithmetic and string manipulation as well as the possibility to apply built-in and user-defined operators. The most important built-in operator is attribute access, denoted by "."—whose successive application on tuple structured objects leads to so-called *path expressions*. To build predicates we have the following *comparison operators*:

$$\begin{aligned} \text{binary operators} &: =, \neq, <, >, \geq, \leq, \mathit{in} \\ \text{unary operator} &: \mathit{empty} \end{aligned}$$

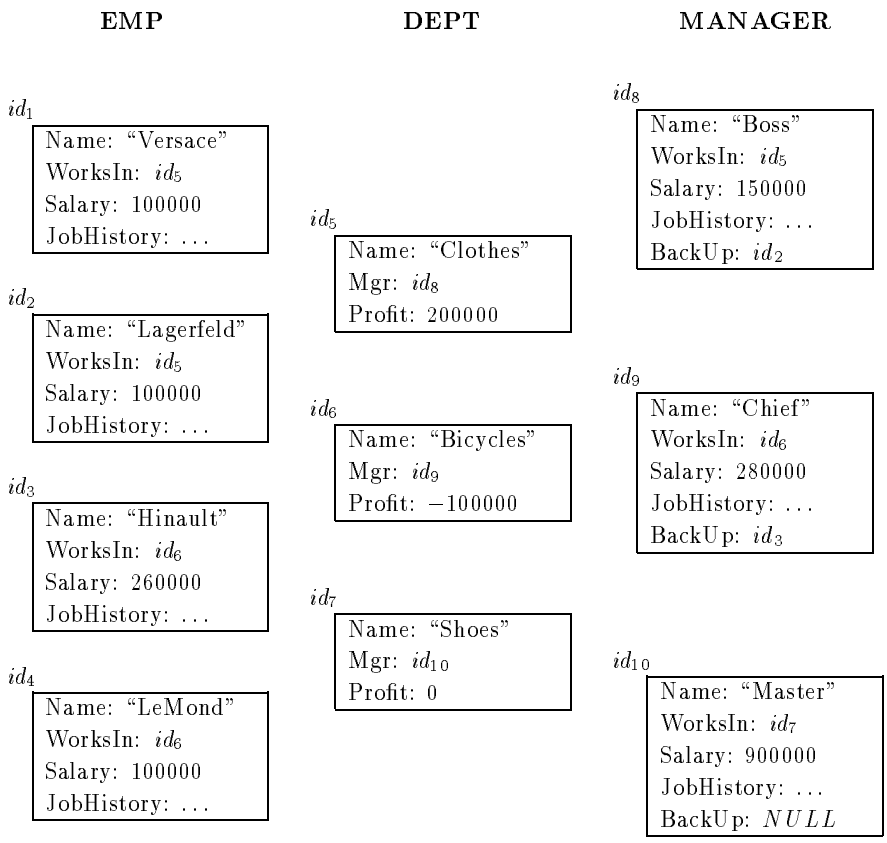


FIGURE 23.2
Example Extension of the Object Base *Company*

```
type EMP is
  body
    [Name: STRING;
     WorksIn: DEPT;
     Salary: INT;
     JobHistory: HISTORY;]   !! assuming HISTORY
  operations                 !! being defined elsewhere
    declare skill: → INT;
  implementation
    define skill is
      ...   !! derived from the JobHistory
end type EMP;
```

```
type DEPT is
  body
    [Name: STRING;
     Mgr: MANAGER;
     Profit: INT;]
  operations
    declare avg_skill: → FLOAT;
  implementation
    define avg_skill is
      ...   !! compute the average skill of
      ...   !! employees within this department
end type DEPT;
```

```
type MANAGER
  supertype EMP is
  body
    [BackUp: EMP;]
end type MANAGER;
```

FIGURE 23.1
Type Definitions of our Running Example

in the relational database area to focus the research in one direction. More specifically, GOM provides the following object-oriented concepts:

- *object identity*, i.e., each object owns an unchangeable object identifier that is used to uniquely identify (reference) the object.
- *complex values*, which—contrary to objects—do not possess an identity.
- *type constructors*, i.e., GOM supports tuple-, set-, and list-constructors. Building upon these, the user is free to define his or her own constructors utilizing the generic type concept.
- *subtyping* in conjunction with *inheritance*, i.e., the subtype inherits all features—properties and operations—of the supertype(s).
- *strong typing*, that is, the type safety of all programs is verified at compile time.
- *object instantiation*, i.e., objects are created by instantiating types.

In addition to types, GOM provides for (complex) *sorts*. The main difference between types and sorts is that the instances of the former are objects whereas the instances of the latter are values.

A traditional relation can be modeled by a sort and its extension is a value.

There exist the following *base sorts*: BOOL, INT, FLOAT, STRING, and OID. Thus, the set of object identifiers (OIDs) is viewed as a base sort. For typing purposes, it is divided into subsets containing identifiers for objects of a certain type. For more details see [Kemper et. al 1991].

23.2.2 The 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 shown in Figure 23.1. Since no **public** clause is supplied in any of the type definitions, all attributes and operations are visible. We further assume all defined types to be persistent. For each type, we maintain a type extension. These extensions also contain all instances of subtypes.

A (small) database extension based on the above schema is shown in Figure 23.2. The labels id_j for $j = \{1, 2, 3, \ldots\}$ denote the object identifiers (OIDs), which are system-wide unique. References via complex attributes are

of being more closely related to relational query optimization formats and, thus, may lead to easier incorporation of relational techniques.

The analysis of the two representation formats is based on performing the step-wise optimization of a rather complex example query. Our investigation reveals that the starting point, i.e., the initially generated internal representation of the declarative user query is crucial for a well structured query optimization process. We propose a normal form into which all queries are initially translated: the *most costly normal form (MCNF)*. This normal form has a positive impact on the design of optimizer strategies in that it allows clear definition of several steps of optimization. As a side effect, this structuring allows the easy and systematic generation of alternative branches that are to be investigated by the optimizer.

The remainder of this paper is organized as follows. In Section 23.2 our object model GOM is surveyed briefly. Then, in Section 23.3 the object-oriented query language GOMql, which is based on the relational language QUEL, is introduced. In Section 23.4 the access support structures that were developed for GOM are explained briefly. In Section 23.5 the two query representation formats, object algebra and term language, are introduced. In Section 23.6 the rule-based optimization of a term expression is illustrated by way of a complete example. The two different internal query representation formats are contrasted, the most costly normal form is defined, and the optimizer strategy is outlined. Section 23.7 concludes this paper.

23.2 Our Object Model GOM

23.2.1 Main Concepts

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 "Manifesto" [Atkinson et. al 1989]—albeit the GOM model was developed much earlier. Similarly, Zdonik and Maier developed the so-called Reference Model in [Zdonik and Maier 1989]. The features that GOM provides are relatively *generic*, so that the results derived for this particular data model can easily be applied to a variety of other object-oriented models. This genericity helps to overcome the diversity of existing object-oriented models which is due to the lack of a commonly adopted base model—which, for example, helped

In this paper we set out to investigate one of the core optimization concepts: the optimizer for declarative queries, commonly called the *query optimizer*. There exists a large body of knowledge—accumulated over a period of almost two decades—within the relational framework. Obviously, it would be worthwhile investigating whether this knowledge can be exploited in an object-oriented context.

Our work is carried out on the basis of the object-oriented database model GOM [Kemper et. al 1991]. GOM is a good candidate for this research since it incorporates the most salient features of object-oriented database models, as described in the "Manifesto" [Atkinson et. al 1989]. In addition, GOM provides two very general indexing methods, which—of course—have to be incorporated into query optimization:

- *access support relations (ASRs)* [Kemper and Moerkotte 1990a] enhance the performance of object access via arbitrarily long reference chains leading from one object to another object or a set of other objects. Access support relations generalize other indexing schemes, e.g., links [Härder 1978], binary join indexes [Valduriez 1987], path indexes for object models [Maier and Stein 1986, Bertino and Kim 1989], and path indexes for the nested relational model [Keßler and Dadam 1991].

- *generalized materialization relations (GMRs)* can be utilized to maintain precomputed results of those functions that are frequently used in search predicates. [Kemper, Kilger, and Moerkotte 1991] introduces this optimization method for object-oriented databases.

These two very general access support structures provide a good "yardstick" for evaluating the quality of a query optimizer by analyzing how well these structures are exploited in the optimized query evaluation plan.

A principal question in the design of a query optimizer concerns the internal representation of the query during the optimization process. In this paper we contrast two representation formats that were developed in GOM for rule-based query rewriting: (1) a procedurally oriented term representation [Kemper and Moerkotte 1990b] and (2) an object algebra that resembles relational algebra—with a few extensions accounting for the characteristics of object-oriented query processing. Both representations exhibit distinct advantages: the term representation is more amenable to incorporation of procedurally oriented constructs, e.g., for specifying the exploitation of access support structures. The object algebra, on the other hand, has the advantage

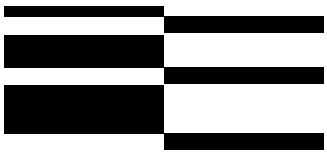
Abstract

There exists a large body of knowledge—gathered over a period of almost two decades—in relational query optimization. In this paper we investigate to what extent the relational optimization techniques can be applied to object-oriented query processing. As a "testbed" we chose our object-oriented database GOM which incorporates two very general indexing structures: *access support relations* which materialize frequently traversed path expressions and *generalized materialization relations* which maintain precomputed function results. These two indexing structures provide a good "yardstick" for analyzing a query optimizer since they subsume many other access support schemes proposed in the literature. In the analysis of rule-based query optimization we contrast two different internal query representation languages that were developed for rule-based optimization in GOM: (1) the procedurally oriented term language and (2) an object algebra representation that is based on relational algebra with a few extensions. The two representation languages are illustrated by way of optimizing an example query, step by step. Our initial approach to algebraic optimization reveals that the starting point for the optimization process is crucial. Therefore, we propose the so-called *most costly normal form (MCNF)* as the initial algebraic representation of object queries. Re-optimizing the example query illustrates that this starting point leads to a more structured approach (1) to query rewriting and (2) to generating alternatives during the optimization process.

23.1 Introduction

Object-oriented database systems are emerging as the "next generation" database technology—especially for advanced engineering applications, e.g., mechanical CAD/CAM. Compared to the relational model, the object-oriented model(s) offer substantially higher expressiveness by incorporating the *structural* and the *behavioral dimension* in a unifying framework. Nevertheless, the acceptance of this new database technology will largely depend on the performance that can be achieved. Database users, in particular engineers, are generally not willing to trade performance for functionality. Therefore, the development of effective optimization techniques for object-oriented systems will largely determine their fate in the commercial market place.

2 3



Query Optimization in Object Bases: Exploiting Relational Techniques

Alfons Kemper Guido Moerkotte