A Blackboard Architecture for Query Optimization in Object Bases

Alfons Kemper* Guido Moerkotte+ KlausPeithner*

*Fakultät für Mathematik und Informatik
Universität Passau
W-8390 Passau, F.R.G.
kemper
peithner @db.fmi.uni-passau .de

+Fakultät für Informatik
Universität Karlsruhe
W-7500 Karlsruhe, F.R.G.
moer@ira.uka.de

Abstract

Adopting the blackboard architecture from the area of Artificial Intelligence, a novel kind of optimizer enabling two desirable ideas will be proposed. Firstly, using such a well-structured approach backpropagation of the optimized queries allows an evolutionary improvement of (crucial) parts of the optimizer. Secondly, the A^* search strategy can be applied to harmonize two contrary properties: Alternatives are generated whenever necessary, and straight-forward optimizing is performed whenever possible, however.

The generic framework for realizing a blackboard optimizer is proposed first. Then, in order to demonstrate the viability of the new approach, a simple example optimizer is presented. It can be viewed as an incarnation of the generic framework.

1 Introduction

Query optimizers—no matter whether relational or object-oriented—are among the most complex software systems that have been built. Therefore, it is not surprising that the design of query optimizers is still a "hot" research issue—especially in object-oriented database systems. The following is a list of desiderata that one may expect of a "good" query optimizer:

1. *extensibility and adaptability*: As new, advanced query evaluation techniques and/or index structures become available the optimizer architecture should facilitate extension or an adaptation—without undue effort.
2. *evolutionary improvability*: It should be possible to *tune* the query optimizer after gathering experience over a longer sequence of queries being optimized. Ultimately, a self-tuning optimizer could be envisioned.
3. *predictability of quality*: Especially when optimizing interactive queries, a tradeoff between the time used for optimization and the quality of the optimized result has to be taken into account. It is, therefore, most useful if we could estimate the quality of the optimization outcome relative to the allocated time for optimization.
4. *graceful degradation under time constraints*: This desideratum is strongly correlated to the preceding one. Allocating less time for optimization should only gracefully degrade the quality of the optimized queries. This, of course, precludes any optimizer that first generates all possible alternatives—without any qualitative ordering—and then evaluates each alternative in turn.
5. *early assessment of alternatives*: The performance of an optimizer strongly depends on the number of alternatives generated. Typically, a heuristics is used to restrict the search space. However, a better, since more flexible, approach is to abandon the less promising alternatives as soon as possible. For that, a cost model which enables an estimate of the potential quality of an alternative already in an early stage of optimization is required.

**Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993.**

6. *specialization*: As in areas of (human) expertise the optimizer architecture should support the integration of highly specialized knowledge to deal with particular (restricted) parts of the optimization process and/or with particular subclasses of queries, e.g., conjunctive or non-recursive queries.

In order to achieve—some of—these desiderata, different query optimizer architectures have been proposed. Unfortunately, all of the proposals fall short of meeting *all* criteria. It even appears that in the attempt of fulfilling some of the desiderata others had to be neglected, e.g., rule-based systems emphasize the extensibility, on the other hand the predictability of the quality in relation to allocated optimization time becomes extremely difficult.

To support extensibility, rule-based systems were proposed [5, 22, 13, 3]. Adaptability is the main concern of the EXODUS query optimizer generator [6], the VOLCANO optimizer generator [7], and the GENESIS tool box system [2]. Structuring the query optimizer for maintenance and specialization is a major concern of proposal [19].

A well-structured architecture will be gained, if the optimization process is subdivided into single, small steps [24]. The "wholistic" approaches, e.g., [26, 4], consider an optimization graph—logical or physical—representing the entire query. That is, at each stage a complete query evaluation plan exists. Then, rules are applied to transform this representation. However, in our opinion it is better to segment the query into building blocks and operations, in order to compose a query evaluation plan step by step. The *building block* approach has already been proposed by Lohman [18].

The cost model is an essential part of a query optimizer in order to assure high-quality output. Since it is not generally obvious which transformation has to be applied for approaching the optimal plan, alternatives are generated [6, 22]. The alternatives are graded by a cost function which has to be continually improved [18]. In [6] an "expected-cost-factor", which is controlled by monitored results of the optimization, is added to each rule. We extend that idea by introducing a mechanism of backpropagation into our architecture.

The right choice of the search strategy is essential for the performance and the extensibility of an optimizer. Randomized optimization algorithms as proposed in, e.g., [10], are very effective, if the shape of the cost function forms a *well*, as pointed out in [9]. Further, the search strategy should be independent from the search space [17]. The search strategy—also proposed for multi query optimization [25]—that will be applied in our sample optimizer is a slight modification of A^*, a search technique which, in its pure form, guarantees to find the optimal solution [20].

In this paper, we present a new architecture for query optimization, based on a *blackboard* approach, which facilitates—in combination with a building block, bottom-up assembling approach and early assessment by utilizing future cost estimates—to address all the desiderata. Our approach is a general one as far as we first devise the generic blackboard-based architecture which can be utilized for any kind of optimizer construction. The viability of the proposed generic optimizer architecture is demonstrated by an example query optimizer which, tough quite simple, demonstrates the main—that is, we describe one sample instantiation of the generic framework which, though still incomplete, adheres to the main principles of the blackboard architecture.

The rest of the paper is organized as follows. In Section 2, the basic framework of the optimizer blackboard is introduced. We conceptually show how the optimization process works and how evolutionary improvability is integrated into the blackboard architecture. In Section 3, the running example—i.e., an object base and an associated query—is given. In order to establish the general ideas in our specific GOM optimizer, the basics as, e.g., the algebra, the organization of our optimizer, and the search strategy are explained in Section 4. Since the cost model is essential for every optimizer generating alternatives, it is outlined in Section 5. Having sketched our Blackboard Optimizer. Section 6 demonstrates a sample optimization process. Section 7 concludes the paper.

2 Generic Framework

2.1 The Pure Blackboard

The optimizer *blackboard* is organized into r successive *regions* $R_0, \ldots, R_{r-1}$. Each region contains a set of *items* representing the advances of the optimizer to derive an optimal evaluation plan for a given query. The original query is translated into some initial internal format which is identified by ϵ and placed into region R_0—as its only item.

A *knowledge source* KS_i is associated with each pair (R_i, R_{i+1}) of successive regions. Each knowledge source KS_i retrieves items to process from region R_i. For each such item, the knowledge source KS_i may generate several alternative items which are emitted—in an order determined by KS_i—into the region R_{i+1}.

Note that there is no restriction concerning the additional data read by a knowledge source. They are

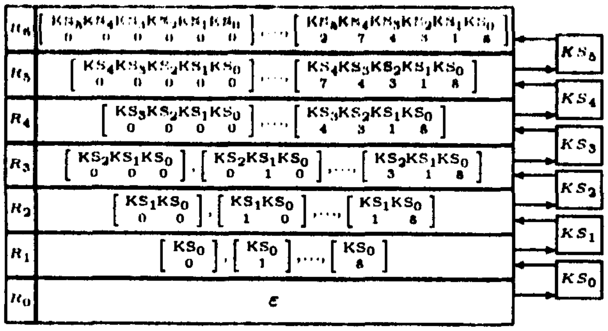


Figure 1: Blackboard Architecture

allowed to read any information at any region, all statistical data, schema data, indexing information, and so forth.

The knowledge sources generate sequences of alternatives. Therefore, the order in which the alternative items are generated can be used for identification. For our abstract blackboard architecture shown in Figure 1, the items at region R_6 are identified by six pairs each consisting of the knowledge source identifier—i.e., $KS_0, \ldots, KS_5$—and the sequence number indicating the position at which the particular item was generated. For example, the identifier

$$\#I = \begin{bmatrix} KS_5 & KS_4 & KS_3 & KS_2 & KS_1 & KS_0 \\ 1 & 0 & 2 & 3 & 4 & 1 \end{bmatrix}$$

of an item I in region R_6 indicates that this particular item I—whose identifier is denoted $\#I$—is the fifth alternative generated by KS_1 from the second item generated by KS_0, etc.

In Section 2.3 we will see that this particular identification mechanism is essential for evaluating the quality and for adapting/calibrating the optimizer blackboard.

2.2 Search Strategy

The blackboard optimizer utilizes a building block approach for generating the (alternative) ***query evaluation plans*** (*QEPs*). Thus, for a given query Q the successive regions of the optimizer blackboard contain more and more complete query evaluation plans—finally, the top-most region R_{r-1} contains complete (alternative) evaluation plans that are equivalent to the user-query Q.

It is essential to control the search space of the optimizer in order to avoid the exhaustive search over all possible query evaluation plans. Therefore, items at all regions have associated costs. There exist two cost functions, $cost_h$ and $cost_f$, which estimate the ***history*** and ***future*** costs for evaluating a certain item. With each item two sets of operations are associated: the set of operations which are already integrated into the item (representing a still incomplete *QEP*) and the set of operations which still have to be integrated. The former set determines $cost_h$ and the latter $cost_f$. Based on these cost functions, the optimizer blackboard is ideally controlled by A^* search [20]. That is, at any given time the knowledge source being applicable to the item with lowest total cost ($cost_h + cost_f$) is allowed to emit further alternatives.

If $cost_h$ corresponds to the actual costs for evaluating the operations of the first set and $cost_f$ is a *close* lower bound of the future costs, A^* search guarantees to find an optimal *QEP* efficiently. However, for query optimization a lower bound estimate of the future costs is always based on the best case for each operation, i.e., the least cost for evaluation is assumed. Hence, the total estimate of the future costs can be (far) lower than the actual costs. Then, the A^* search could possibly degenerate to an (almost) exhaustive search which leads to unacceptable optimization times. In order to straighten the optimization, the proposed A^* search strategy is enhanced by the subsequently described ***ballooning component***.

As explained before, knowledge sources retrieve an item I from their associated region and generate an ordered sequence of items $I_1, \ldots, I_j$ which are emitted into the successor region. It is one of the major objectives in the design and subsequent calibration—cf. Section 2.3, below—of a knowledge source to ensure that the most promising alternatives are generated first. Such-like sophisticated knowledge sources entail the incorporation of the ballooning control component to expedite the optimization process. The basic idea of the ballooning control is to ***periodically*** and ***temporarily*** "switch off" the A^* control and to process the first few alternatives generated by the knowledge sources without any cost control. Thereby, some "balloons" will "rise" through successive regions—possibly all the way up to the top-most region where items constitute complete *QEPs*.

When switching back to A^* search only the balloons at the top of the derivation chains are further considered; intermediate steps generated during ballooning are discarded—thereby reducing the resulting search space and "straightening" the optimization. Since the blackboard approach allows to assess the sequence of the items generated by a knowledge source with respect to its quality for the global optimization, it is expected that the integration of the ballooning component into the A^* search does not substantially degrade the quality of the optimization. Ballooning will only process

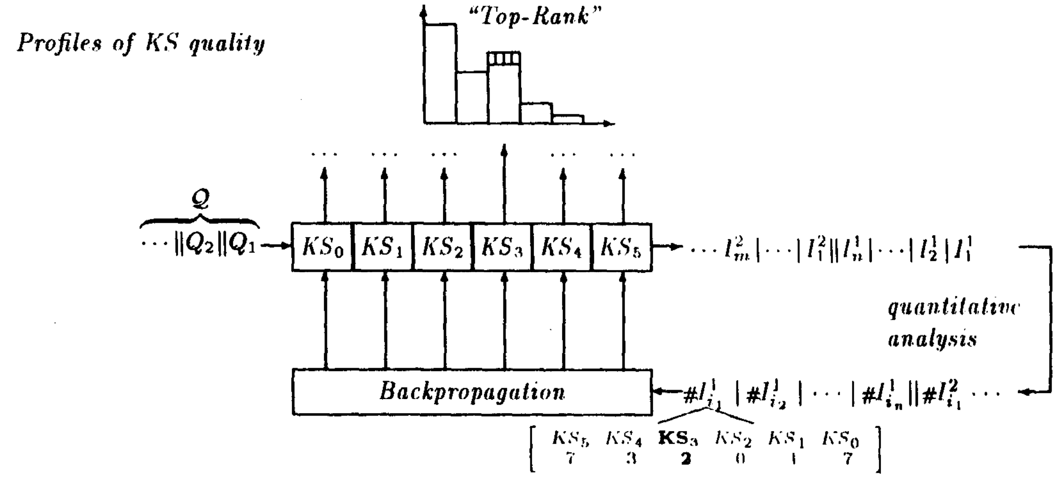


Figure 2: Evaluation and Calibration by Backpropagation

highly-promising items very efficiently —without backtracking. Further, a reconciliation of the time allocated for optimization and the quality of the solution—recall Desideratum 4. of the Introduction—can be achieved by increasing or decreasing the share of ballooning.

A simplified version of the search algorithm used in the GOM Blackboard Optimizer is given in Section 4.4.

2.3 Backpropagation

The structuring of our optimizer blackboard imposed by the knowledge sources operating on successive regions enables the thorough quantitative evaluation and subsequent calibration of the quality of the knowledge sources. This is achieved by backpropagating the outcome of an extensive set of benchmark queries. The principle of backpropagating is depicted in Figure 2.

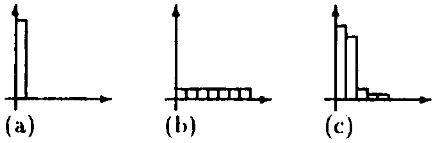
Let $Q = \{Q_1, Q_2, \ldots\}$ be a large set of representative queries—which are either extracted from user supplied queries or are generated by a query generator. For these queries let the optimizer generate **all** possible alternative query evaluation plans, i.e., for this purpose all items are expanded at regions $R_0, \ldots, R_{r-2}$. It is, however, essential that the optimizer obeys the control imposed by the pure A^* search—except that the search continues even after the optimum has been generated. For a query Q_j a sequence $I^j_n, \cdots, I^j_2, I^j_1$ of alternative items specifying a complete *QEP* at region R_{r-1}—the right-most item being generated first and the left-most last—is obtained. Note that the alternatives are already sorted by their cost. More specifically, $\#I^j_{i_1}$ is the cheapest *QEP* identifier and $I^j_{i_n}$ is the most expensive one for a query Q_j.

This ordered sequence of plan identifiers is propagated back to the blackboard optimizer in order to evaluate the individual knowledge sources' quality. The quality of a knowledge source is measured in terms of the relative position at which an alternative was generated in comparison to the position of this alternative in the *QEP* sequence ordered by their running times. By evaluating a representative number of queries, a so called "Top-Rank" profile can be derived. In Figure 2, e.g., the backpropagation of Q_1 increases the third column of the Top-Rank profile of KS_3 since the identifier $\#I^1_{i_1}$ of the top rank *QEP* states that the appropriate *QEP* was generated as the third alternative by KS_3.

In Figure 2, the Top-Rank profile of knowledge source KS_3 indicates that almost all top rank *QEPs* emerge from the first three alternatives of this knowledge source. Actually, in practice we are usually more interested in the so-called "Top-δ" profiles in which all those query evaluation plans with running time within δ% of the actual optimum are considered semi-optimal—where δ may be some application domain specific threshold value.

Quantitative analysis of the profiles facilitates predicting the average quality of the optimization—as envisioned in Desideratum 3. stated in the Introduction. Let $BAP(KS_i, n_i)$ denote the probability that the first n_i alternatives emitted by knowledge source KS_i include the optimal one—under the condition that KS_i starts with the alternative from knowledge source KS_{i-1} which ultimately leads to the optimum. This function can easily be computed from the "Top-Rank" profile. Furthermore, let b_{KS_i} denote a (limiting) branching factor of knowledge source KS_i, i.e., the maximal number of alternatives that knowledge source KS_i is allowed to generate. Then, the following calculation $\prod_{i\in\{0,\ldots,r-2\}} BAP(KS_i, b_{KS_i})$ derives the probability that the optimal *QEP* is among the $\prod_{i\in\{0,\ldots,r-2\}} b_{KS_i}$ alternatives that emerge at the top-most region R_{r-1}.

Further, a more qualitative analysis of the profiles facilitates tuning the individual knowledge sources—as demanded in Desideratum 2. To give an idea of how the optimizer can be improved, the three following "hypothetical" profiles are depicted:



An ideal profile is Profile (a)—no improvement can be made. The worst one can think of is Profile (b). It looks like the profile of a "no-knowledge knowledge source". Usually, a profile like (c) is worth striving for. It displays that the knowledge source has only to generate few alternatives in order to carry the creation of the optimal (Top-Rank) or a semi-optimal *QEP* (Top-δ).

Ultimately, we envision that the profiles can be used by the optimizer for self-tuning—Desideratum 2—since the analysis of the profiles as well as the generation of the hints may be carried out automatically.

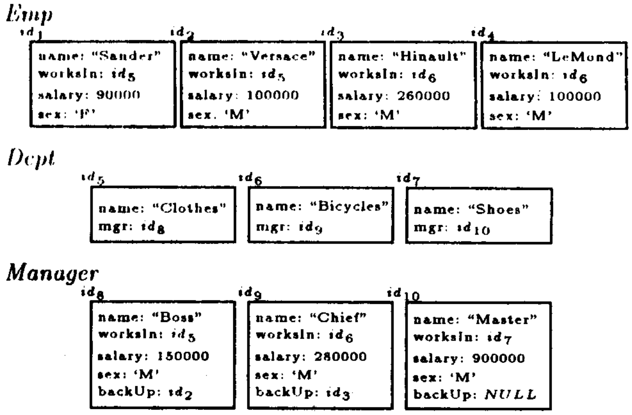
2.4 Generalized Optimizer Blackboard

In the discussion of the hypothetical knowledge source profiles we already observed that it might be useful to classify queries within the regions. This allows to process them more specifically by particular highly customized knowledge sources. The classification of queries depends on the region. As an example, consider classification of recursive vs. non-recursive queries which is important to know for applying the right algorithm to compute join orderings.

In the pure architecture a knowledge source reads items from region R_i and emits the outcome into the next higher region R_{i+1}. We extend this concept such that an item leaving a special region R_{i_o} is allowed to re-enter the blackboard at a lower level R_{i_e} ($i_e \leq i_o$). Thus, items can iterate over the regions R_{i_e} to R_{i_o}. An item will leave that iteration if it comes back to R_{i_o} without being modified.

3 Running Example

In this section, an example object base—called *Company*—is presented. In Figure 3, ten objects belonging to types *Emp*, *Dept*, and *Manager* are shown. The type definitions are omitted—for the further discussion it is only of importance that each object of type *Emp* has the attributes ***name*** : *String*, ***worksIn*** : *Dept*, ***salary*** : *Float*, and ***sex*** : *Char*, and each object of type *Dept* the attributes ***name*** : *String* and ***mgr*** : *Manager*. Since *Manager* is a subtype of *Emp* it contains all the attributes of *Emp* and, furthermore, it has one attribute ***backUp*** : *Emp* additionally. Further, a type-associated function ***skill*** computing a ranking number for individual *Emp*loyees is assumed.



name: "Sander"
worksIn: id_5
salary: 90000
sex: 'F'

name: "Versace"
worksIn: id_5
salary: 100000
sex: 'M'

name: "Hinault"
worksIn: id_6
salary: 260000
sex: 'M'

name: "LeMond"
worksIn: id_6
salary: 100000
sex: 'M'

name: "Clothes"
mgr: id_8

name: "Bicycles"
mgr: id_9

name: "Shoes"
mgr: id_{10}

name: "Boss"
worksIn: id_5
salary: 150000
sex: 'M'
backUp: id_2

name: "Chief"
worksIn: id_6
salary: 280000
sex: 'M'
backUp: id_3

name: "Master"
worksIn: id_7
salary: 900000
sex: 'M'
backUp: *NULL*

Figure 3: Example Extension of *Company*

The labels id_i for $i \in \{1, 2, 3, \ldots\}$ denote the system-wide unique object identifiers (*OIDs*). References via attributes are maintained uni-directionally in GOM—as in almost all other object models. For example, in the extension of *Company* there is a reference from *Emp*loyee id_1 to *Dept* id_5 via the ***worksIn*** attribute.

The Example Query For the object model GOM, a QUEL-like query language called GOMql [13] was developed. As an example query, we want to know whenever there is a *Manager*—usually called "MCP"—who pays a female less than a male *Emp*loyee (in one of his *Dept*s) even though the female is better qualified. We want to retrieve the manager and as evidence the female, the male, and the difference of their salaries. In GOMql the query can be formulated as follows:

```
range    u : Emp, o : Emp
retrieve [mcp : u.worksIn.mgr, underPaid : u,
          overPaid : o, difference : o.salary - u.salary]
where    u.worksIn.mgr = o.worksIn.mgr and
         u.skill > o.skill and
         u.salary < o.salary and
         u.sex = 'F' and o.sex = 'M'
```

There are three clauses. The **range**-clause introduces the needed variables and binds them to finite ranges—here, the extensions of the types. The **retrieve**-clause specifies the final projection of the query, and the **where**-clause contains the selection predicate. Under the assumption that "*Sander*" has higher *skill* than "*Versace*", the relation $\{[mcp : id_8, underPaid : id_1, overPaid : id_2, difference : 10000]\}$ is the outcome of the query with respect to the object base *Company*.

At this point, we would like to stress that even though we have chosen GOM and GOMql as the example data model and query language, respectively, the results obviously apply to other object-oriented data models and query languages as well.

The Index Structures The GOM query evaluation is supported by two very general index structures tailored for object-oriented data models:

- *Access Support Relations* (*ASRs*) [12] are used to materialize (frequently) traversed reference chains, and
- *Generalized Materialization Relations* (*GMRs*) [11] maintain pre-computed function results.

Since these two index structures have to be taken into account in the optimization process, two index relations based on the schema *Company* are exemplified:

$\llbracket Emp.worksIn.mgr \rrbracket$		
$\#0 : OID_{Emp}$	$\#1 : OID_{Dept}$	$\#2 : OID_{Manager}$
id_1	id_5	id_8
id_2	id_5	id_8
...	...	...
id_{10}	id_7	id_{10}

$\langle\!\langle Emp.skill \rangle\!\rangle$	
$\#0 : OID_{Emp}$	$\#1 : int$
id_1	10
id_2	4
...	...
id_{10}	10

The extension of the *ASR* $\llbracket Emp.worksIn.mgr \rrbracket$ which contains all paths corresponding to the indicated path expression, and of the *GMR* $\langle\!\langle Emp.skill \rangle\!\rangle$ which maintains the pre-computed *skill* function for each *Emp*loyee are depicted. Note that the columns of these index relations are sequentially numbered, i.e., #0, #1, ...

4 GOM Blackboard Optimizer

4.1 The Algebra

The *query evaluation plans* (*QEPs*) are *directed acyclic graphs* (*DAGs*) consisting of algebraic operator applications. *Building blocks* standing for sets of *OIDs* of a type T (denoted by $oid(T)$), *ASRs* (denoted by $\llbracket \ldots \rrbracket$), and *GMRs* (denoted by $\langle\!\langle \ldots \rangle\!\rangle$) are the leaves of the *DAGs*. The treatment of indexes—like *ASRs* and *GMRs*—as additional sources of information is already present in the notion of shadow tables as introduced in [23]. In accordance with the building block approach [18], the *DAGs* are successively composed bottom-up—operations are added to the *DAG* and common subexpressions are factorized. In order to compute a (near-)optimal *DAG* the optimizer has to determine an optimal set of building blocks and an optimal order of the algebraic operations.

Our algebra mainly copes with relations. In order to refer to single columns of relations, we use so-called *information units* (*IUs*). We do not call them attributes, since we want to avoid any conflict with the attributes at the GOM object type level. Each *IU* is unique throughout the entire optimization process, i.e., over all alternatives which would be generated, and so an unambiguous dereferencing mechanism is obtained for the algebraic operations and the cost functions.

Besides the usual set operations ($\cup$, $\setminus$), the algebra consists of the common relational *selection* σ, *projection* π, *join* $\bowtie$, and *renaming* ϱ. Further, a mapping operator (χ)—called *expansion*—belongs to the algebra. Let T be a type, v, v_1, v_1', ..., v_n, v_n' be *IUs*, a_1, ..., a_n be attributes, $\phi \in \{=, <, >, \ldots\}$ be a comparison operator, and c be a constant. Then, the building blocks and the algebraic operators are informally defined as follows:

- *building blocks*: The extension of T $oid(T)$, an *ASR* $\llbracket \ldots \rrbracket$, and a *GMR* $\langle\!\langle \ldots \rangle\!\rangle$ are building blocks. The columns of the relations retrieved by them are denoted by **self** and $\#0, \ldots, \#n$, respectively. We assume indices on the first and last column of an *ASR* and on each column of a *GMR*.
- *expansions*: An expansion $\chi_{v_1:v.a_1,\ldots,v_n:v.a_n}$ dereferences sets of *OIDs* denoted by *IU* v such that the attribute values can be obtained and be assigned to new *IUs* v_1, ..., v_n, respectively. The input relation is expanded by new columns denoted v_1, ..., v_n. Further, the χ operator may also expand the tuples by function invocations—instead of attribute accesses. The parameters of functions are enclosed in parentheses following its name.

- ***usual relational operations***: $\bowtie_{v_1 \phi v_2}$ denotes a join, $\sigma_{v_1 \phi c}$ and $\sigma_{v_1 \phi v_2}$ selections, $\pi_{v_1,\ldots,v_n}$ a projection on the *IUs* in the subscript, and $\varrho_{v'_1=v_1,\ldots,v'_n=v_n}$ a renaming operation where the column named v_i is renamed to v'_i $(i = 1, \ldots, n)$.

Relying heavily on ordinary relational operators allows us to exploit relational optimization techniques [16, 14].

4.2 The Normal Forms

In object-oriented query processing it is common to translate the query into an internal representation as close to the original query as possible—witness, e.g., [1, 4, 13, 14]. This is also valid for relational query processing where, e.g., an SQL query is translated into a $\pi\sigma\bowtie$-expression. However, this representation exhibits another property which the initial internal representation of object-oriented queries very often lacks: It is an (expensive) well-structured term facilitating a straightforward splitting into building blocks and operations.

Our proposed starting point—called *Most Costly Normal Form* (*MCNF*) [14]—has one additional χ-expansion directly following the $\bowtie$ resulting in a $\pi\sigma\chi\bowtie$ sequence. All the extensions whose instances are needed for the query evaluation are joined with ***true*** as join predicate. χ-expansions follow enhancing each tuple of the resulting relation by further information needed to evaluate the selection predicate solely on the basis of this result. Thus, two vital concepts of object-orientation—access via *OIDs* (implicit dereferenciation) and function invocation—are integrated into the *MCNF*, and are prepared for their optimization. Then, the selections accompanied by the final projection onto the required *IUs* are appended.

The *MCNF* representation of the example query "MCP" is shown below:

$$\begin{array}{l}
\pi_{mcp:um,underPaid:u,overPaid:o,difference:osa-usa}(\\
\;\sigma_{osx=\text{'M'}}(\sigma_{usx=\text{'F'}}(\sigma_{usa<osa}(\sigma_{usk>osk}(\sigma_{um=om}(\\
\;\;\chi_{um:ud.mgr}(\chi_{om:od.mgr}(\\
\;\;\chi_{ud:u.worksIn,usa:u.salary,usx:u.sex}(\\
\;\;\chi_{od:o.worksIn,osa:o.salary,osx:o.sex}(\\
\;\;\chi_{usk:u.skill}(\chi_{osk:o.skill}(\\
\;\;\;\varrho_{u=self}(oid(Emp)) \bowtie_{true} \varrho_{o=self}(oid(Emp)) \ldots)
\end{array}$$

The *MCNF* is further enhanced [15] in order to obtain a convenient basis for composing the query evaluation plans. A table combining the building blocks and the operations with catalog information is derived such that it contains ***all*** information relevant for optimizing the query. Thus, we can, e.g., efficiently retrieve the building blocks and the operations in which a given *IU* is involved. This elaborated normal form is obtained by decomposing the *MCNF* term into its building blocks and operations. Each piece is then enriched by statistical data being relevant to the query. For example, the cardinalities of the building blocks and the selectivities of the operations are attached. The fact which columns of a building block are supported by an index is important for an exact cost estimate. Hence, this information is also maintained.

4.3 Regions and Knowledge Sources

The blackboard of our GOM Blackboard Optimizer is subdivided into seven ***regions***—each one completing the *QEP* in a particular way: R_0 (*MCNF*), R_1 (Decomposition), R_2 (Anchor Sets), R_3 (Introduce χ), R_4 (Introduce σ), R_5 (Introduce $\bowtie$), and R_6 (Introduce π). Each region supplies ***items***, each of which possesses an entry ***currentDAGs*** and an entry ***futureWork*** where the *DAGs* composed so far and the remaining operations, respectively, are stored.

The ***knowledge sources*** of type KS_i read items at region R_i and write items at region R_{i+1}. What follows is an informal description of the knowledge sources at each region. We assume that the query is represented in *MCNF* format at region R_0.

KS_0 (to "Decomposition"): The *MCNF* term is decomposed into building blocks and operations. The additional information is obtained from the schema manager which also manages the statistical data. Additionally, the ***ASRs*** and ***GMRs*** which can be integrated into the query are determined. There exists only one knowledge source of this type and it does not produce any alternatives.

KS_1 (to "Anchor Sets"): A knowledge source of this type determines which building blocks are chosen for evaluating the query. We call such a minimal (i.e., non-redundant) set of building blocks containing enough information for answering the query an ***anchor set***. KS_1 generates several anchor sets and sorts them according to special heuristics, e.g., considering the number of joins or the number of operations left in the ***futureWork*** entry.

KS_2 (to "Introduce χ"): Expansions are added to the ***currentDAGs*** entry. In the current implementation, the following heuristics is applied: An expansion—or a pair of expansions—is integrated into the *DAGs* if (and only if) a selection or a join directly depends on it, or the ***futureWork*** entry of the item only contains expansions and projections.

KS_3 (to "Introduce σ"): According to the heuristics "introduce selections as early as possible", selections are integrated into the query whenever it is possible.

KS_4 (to "Introduce $\bowtie$"): At each iteration the knowledge source of type KS_4 introduces at most one join. As a consequence, for each item a join order is obtained by repeated iterations. Alternatives might have different join orderings.

KS_5 (to "Introduce π"): Finally, projections are added to the *DAG*. We rule out the following two sequences: $\sigma\pi\bowtie$ and $\sigma\pi\chi$, since a $\pi\sigma\bowtie$ and a $\pi\sigma\chi$ sequence can be replaced by only one single physical operation.

The blackboard is re-entered from region R_5 to R_2 until all expansions, selections, and joins are processed, that is, the ***futureWork*** entry is empty except for a single projection.

In order to avoid evaluating equal expressions twice, items leaving regions R_1, R_2, R_3, R_4, and R_5 are ***factorized***. For example, if KS_1 selects $\varrho_{u=self}(oid(Emp))$ and $\varrho_{o=self}(oid(Emp))$ as elements of an anchor set, they will be factorized as follows:

$$\begin{tikzcd}
\varrho_{u=self} & & \varrho_{o=self} \\
& oid(Emp) \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

The full set of factorization rules applied can be found in [15]. As a result, the optimizer generates a *DAG* which is a "logical" query evaluation plan.

4.4 Search Algorithm

The search strategy in the GOM Blackboard Optimizer consists of two parts. On the one hand, A^* ***search*** advances the alternative with the minimal sum of history ($cost_h$) and future costs ($cost_f$), and on the other hand, ***ballooning*** proceeds the alternative(s) emitted first by a knowledge source. The actual search strategy combines these two techniques by allowing a certain ratio of optimization steps to be done under A^* search and under the ballooning control, respectively. The search strategy is outlined as follows:

1. Insert the starting state (item) ε into the list OPEN of unexpanded states.
2. Sort the elements I of $OPEN$ by increasing $f(I) := cost_h(I) + cost_f(I)$ values.
3. If the ballooning flag is raised, do
   (a) remove the first $b_{initial}$ elements from OPEN and insert them into the set B
   (b) perform the following steps $b_{iterations}$ times
      i. expand each $I \in B$ by its appropriate knowledge source to $I_1, \ldots, I_j$ for $j \leq b_{branch}$
      ii. remove I from B and insert the item into CLOSED
      iii. insert $I_1, \ldots, I_j$ into B
   (c) transfer the items in B to OPEN, and go to Step 2.
4. Remove the left-most item I from OPEN —i.e., the item for which $f(I) := cost_h(I) + cost_f(I)$ is minimum (ties broken arbitrarily)— and place it on CLOSED.
5. If I is a goal state, i.e., $I.FW = \emptyset$, exit successfully with the solution I.
6. Let the appropriate knowledge source expand state I, generating all its successors.
7. For every successor I' of I:
   (a) insert I' into OPEN unless
   (b) there exists $I'' \in OPEN \cup CLOSED$ with $I'.FW = I''.FW$ then
      i. if $cost_h(I') < cost_h(I'')$, then insert I' into OPEN and transfer I'' to PRUNED
      ii. else, if $cost_h(I') \geq cost_h(I'')$, then insert I' into PRUNED
8. Go to Step 2.

The A^* search algorithm is a *best first algorithm* [20]. It starts with inserting ε, the initial state, into *OPEN*. *OPEN* contains all states which have been reached but have not been fully expanded, i.e., it contains all items waiting for their further processing. In each iteration, A^* search continues with the item of *OPEN* which has the least *f-value*, i.e., the minimal sum of $cost_h$ and $cost_f$. That item is expanded, i.e., its successors are put into *OPEN*, and then it is promoted to *CLOSED*, the set of all fully expanded states. The algorithm will successfully terminate as soon as an item is generated whose future work—denoted by *FW*—is empty and whose costs are minimal.

In Step 3. the control is temporarily switched from A^* search to ballooning. Ballooning might, for example, be triggered after a certain number of iterations in the A^* search have been performed. Then, the first $b_{initial}$ items of *OPEN* are expanded $b_{iterations}$ times, i.e., the items are expanded to lists, the first, at most b_{branch} — which should be one in most cases — elements of each list are then expanded, and so on. The numbers $b_{initial}$, $b_{iterations}$, and b_{branch} can be set depending on the analysis of the entire query and the current state of the optimizing process. For example, the optimizing process of a query containing many χ-expansions and selections may be expedited by low $b_{initial}$, high $b_{iterations}$, and low b_{branch} parameters, since generating many alternatives is unnecessary for integrating these operations. Thus, by ballooning fast optimizing can be switched on whenever it seems acceptable.

For the *pruning conditions* in Step (7b), a special case of the optimality criterion [20] is presupposed: If there are two items I_1 and I_2 with equal future work entries both containing an operation op and, further, $cost_h(I_1) < cost_h(I_2)$ holds, then integrating op into the history work entry of I_1 and I_2 will keep the cost order between the two items invariant. Therefore, all items (states) which produce higher costs than an item with the same future work are pruned by the pruning condition (7b) and transferred to a set $PRUNED$ since, due to the optimality criterion, they cannot possibly yield a better item. Thus, the successor item I' will cause the pruning of some items $I'' \in OPEN \cup CLOSED$, if it is less "expensive", and it will be pruned itself by an item $I'' \in OPEN \cup CLOSED$, if it is more "expensive".

The pruning conditions can be strengthened, if some further properties are ensured by the cost functions [15].

5 Cost Model

From specific data extracted from the object base, the costs for scanning the building blocks and evaluating the operations are estimated.

For the calculation of the *history costs* as well as the *future costs*, two parameters are assigned to each DAG node: the cardinality $\#o$ of the output relation, and the numbers $\#e = (e_{v_1}, \ldots, e_{v_n})$ of distinct values belonging to the IUs $v_1, \ldots, v_n$ of the output relation — called *e-values*. Their calculation from so-called basic numbers is explained below. The number of page faults $\#p$ and the CPU costs $\#c$ — additionally to $\#o$ and $\#e$ assigned to each DAG node — are derived from $\#o$, $\#e$, and the basic numbers. For estimating $\#p$, the well-known formula of Yao [27] is used.

The estimate for $\#c$ is based on system-dependent functions which estimate the CPU costs for the building blocks and the appropriate operations with $\#o$ and $\#e$ as input.

Thus, the calculation of the history costs is fairly straight-forward. The future cost estimate of an operation is demanded to be a lower bound of the actual costs. For that, we derive a lower bound of the size and the e-values of the input relations (see below). Then, we can calculate the future costs in basically the same way as the history costs.

Assigning a quadruple $\tau = (\#p, \#c, \#o, \#e)$ to each DAG node, the costs of a DAG are computed by summing up the costs of its nodes. Then, we compute the history cost of an item by adding up the costs of the $DAGs$ in the *currentDAGs* entry of the item and the future costs by adding up the costs of the operations in the *futureWork* entry.

The data used for the cost calculations is stored as *basic numbers* in three levels: "*Values from the Object Base*", "*Single Selectivities*", and "*Combined Selectivities*".

For every object type T, the cardinality c_T of its extension and the values p_T^{oid} and p_T^{object} — which denote the number of pages occupied by the extension, i.e., the set of $OIDs$, and by the objects, respectively — are available as values from the object base. Let a be an attribute of an object type T. If a refers to an object type, $def_{T,a}$ denotes the probability that the attribute is defined ($\neq NULL$). For each attribute a of type T, the parameter $c_{T,a}$ denotes the size of its range. For each method m, the size of its range $c_{T,m}$ and its average execution time[1] $exec_{T,m}(n)$ — for executing n times the method m on $OIDs$ of type T — is maintained. The cardinality of an ASR $[\![\ldots]\!]$ and a GMR $\langle\!\langle\ldots\rangle\!\rangle$ — which is denoted $c_{[\![\ldots]\!]}$ and $c_{\langle\!\langle\ldots\rangle\!\rangle}$, respectively — and the number of pages they occupy — denoted $p_{[\![\ldots]\!]}$ and $p_{\langle\!\langle\ldots\rangle\!\rangle}$ — are also available as values from the object base.

The selectivity s for a unary operation $op_1(R)$ is defined as $s(op_1(R)) = |op_1(R)|/|R|$, and for a binary operation op_2 as $s(op_2(R_1, R_2)) = |op_2(R_1, R_2)|/(|R_1| * |R_2|)$. These single selectivities can be estimated in three different ways with increasing accuracy:

1. As in [24], the selectivities might be derived from simple estimates. Thus, if the basic numbers $c_{Emp,skill} = 10$, $c_{Emp,salary} = 10.000$, and $c_{Manager} = 150$ are given, the selectivity for $\sigma_{usk>osk}$, $\sigma_{usa<osa}$, and $\sigma_{um=om}$ will be $(1 - (1/c_{Emp,skill}))/2 = 0.45$, $(1 - (1/c_{Emp,salary}))/2 \approx 0.5$, and $1/c_{Manager} \approx 0.007$, respectively.

2. The selectivities can also be determined by histograms [21]. For that, histograms are generated by sampling the object base. The selectivities for

---

[1] We know that this is only a rough estimate. Future versions of the cost model will refine this.

$\sigma_{osx='F'}$ and $\sigma_{usx='M'}$ can be determined in this way.

3. During the evaluation of a query, one can gain more accurate selectivity estimates for use in future query optimization by monitoring.

Since, in the current implementation, the independence of attribute values is presupposed, combined selectivities are the product of their single selectivities. In the future, this will be refined.

Knowing the selectivity s of an operation, we are able to derive the output size $\#o$ of that operation by multiplying s with the cardinality of the input relation(s). The output size of a building block, i.e., type extensions, *ASRs*, and *GMRs*, is given by the basic numbers. Thus, the cardinalities of the (intermediate) relations of a *DAG* are calculated bottom-up.

Since not the total number, but the number of *distinct OIDs* is essential for cost estimates considering χ-expansions and retrieving building blocks with an index, an e-value e_v defined by $|\pi_v(R)|$ is assigned to each *IU* v in a relation R. The bottom-up calculation of the e-values is performed as follows: The initialization is done by the basic numbers of the building blocks. The further calculation is mainly based on a formula also used for generating join orderings [8]. For example, let an expansion $\chi_{v':v.a}$ be applied on a relation R where the e-values are known. Let $c_{T_v,a}$ be the cardinality of the range of the attribute/type-associated function a and e_v be equal to $|\pi_v(R)|$. Then, the following formula determines the number e'_v of values being referenced:

$$e_{v'} = c_{T_v,a} * (1 - (1 - 1/c_{T_v,a})^{e_v})$$

Since the e-values decrease with each operation application, we can determine a non-trivial lower bound on all e-values. Let R be the relation obtained by evaluating the *DAG* of the *MCNF* where the last projection is cut off. Then, $|\pi_v(R)|$ gives a lower bound on all e-values of the *IU* v in all (possibly unfinished) *DAGs* representing the query. Using the formulas for history costs and applying these to the operations in the *futureWork* entry of an item, we arrive at a lower bound on the future costs.

6 Sample Optimization

Performing the optimization process for the running example, some decisions individually made at each region, factorization, and pruning will be demonstrated.

The normal forms were already explained in Section 4.2. Thus, the sample optimization starts at generating anchor sets. Each non-redundant set which binds the *IUs* u and o is a potential anchor set for our example. The values for the other *IUs* can be retrieved by χ-expansions. Because of symmetry of u and o, we only give the sets resulting in bindings for u:

$$\begin{aligned}
A_1 &= \{\varrho_{u=self}(oid(Emp))\} \\
A_2 &= \{\varrho_{u=\#0,usk=\#1}(\langle\!\langle Emp.skill\rangle\!\rangle)\} \\
A_3 &= \{\varrho_{u=\#0,ud=\#1,um=\#2}([\![Emp.worksIn.mgr]\!])\} \\
A_4 &= \begin{array}{l}\{\varrho_{u=\#0,usk=\#1}(\langle\!\langle Emp.skill\rangle\!\rangle), \\ \varrho_{u'=\#0,ud=\#1,um=\#2}([\![Emp.worksIn.mgr]\!])\}\end{array} \\
A_5 &= \begin{array}{l}\{\varrho_{u'=\#0,usk=\#1}(\langle\!\langle Emp.skill\rangle\!\rangle), \\ \varrho_{u=\#0,ud=\#1,um=\#2}([\![Emp.worksIn.mgr]\!])\}\end{array}
\end{aligned}$$

Due to the corresponding sets for o, the appropriate knowledge source generates at most $5 * 5 = 25$ alternative anchor sets. Because of the cost functions, the GOM Blackboard Optimizer favors the following anchor set $A_{2,2}$ originated from A_2:

$$A_{2,2} = \begin{array}{l}\{\varrho_{u=\#0,usk=\#1}(\langle\!\langle Emp.skill\rangle\!\rangle), \\ \varrho_{o=\#0,osk=\#1}(\langle\!\langle Emp.skill\rangle\!\rangle)\}\end{array}$$

Though A^* search might backtrack to one of the alternative anchor sets the example optimization is limited to $A_{2,2}$. Factorizing this anchor set results in the following *currentDAGs* entry:

Now, we want to sketch the search space originating in the item I_0 containing the *DAG* above. In order to simplify the following consideration, the future work for that item is reduced to the operations $\chi_{usx:u.sex}$, $\chi_{osx:o.sex}$, $\sigma_{usx='F'}$, and $\sigma_{osx='M'}$. The GOM Blackboard Optimizer doesn't usually open the whole search space as it is depicted in Figure 4. There, the possible paths leading from I_0 to an item I_1 containing the future work of I_0 in its *currentDAGs* entry are illustrated. If pure A^* search is applied and the evaluation costs of the operations differ hardly, all six paths from I_0 to I_1 are examined. Although some of the six alternatives are pruned every time edges come together, a further reduction of the expense can be achieved. Since for integrating expansions and selections, the knowledge sources deliver a good sequence of the items, the trigger condition of the ballooning component can be set to true and the branching factor b_{branch} to one. Then, only one alternative is produced.

The other expansions by *worksIn* and *salary* are also integrated. Since we assume that an attribute access of an object already resident in the buffer is free of cost, the expansions dereferencing u and o, respectively, are

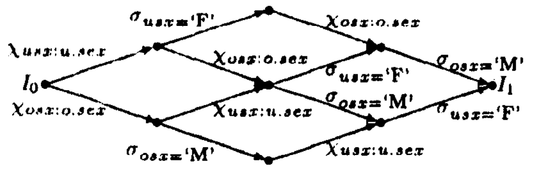


Figure 4: Example Search Space from I_0 to I_1

put together. Further, the two expansions are factorized as the lower part of the DAG in Figure 5 shows.

Two expansions, three joins, and one projection are left in the *futureWork* entry. The joins $\sigma_{usa<osa}$, $\sigma_{usk>osk}$, or $\sigma_{um=om}$ can be added to the actual *currentDAGs* entry[2]. Thus, the state expansion—Step 6 of the search strategy (cf. Section 4.4)—leads to three items I_1', I_1'', and I_1'''.

The history costs of the three items I_1', I_1'', and I_1''' differ hardly. In contrast to that, the future cost estimates differ substantially, since the selectivities and, therefore, the estimates of the cardinalities are very different. As pointed out in Section 5, the selectivity estimate of the operation $\sigma_{um=om}$ is far less than the other two selectivities. Thus, the future costs and consequently the f-value of the item where that operation is integrated into its *CurrentDAGs* entry is lowest. Hence, this item is further processed and the two remaining joins are added to its *CurrentDAGs* entry as selections.

The final projection completes the *DAG*. Furthermore, projections which reduce the size of the intermediate relations are integrated into the *DAG*.

The resulting *DAG* is given in Figure 5. Further optimizations will map the operations to physical operations. Since every $\pi\,\sigma\,\chi$ and every $\pi \bowtie \chi$ sequence entails only one physical operation, the resulting *DAG* is divided by dashed horizontal lines.

7 Conclusion

A novel architecture for query optimization based on a blackboard which is organized in successive regions has been devised. At every region knowledge sources are activated consecutively completing alternative query evaluation plans. Starting from basic building blocks a finite set of algebraic operations is added such that a *DAG* finally results in a (logical) query evaluation plan.

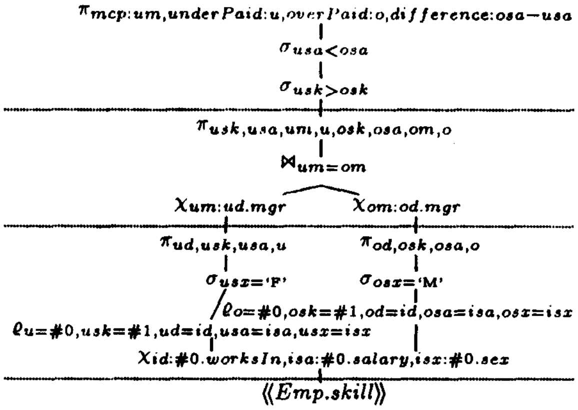


Figure 5: Resulting DAG of the Sample Optimization

Due to this well-structured approach, the optimizer can continually be improved. By backpropagating the optimized queries, each knowledge source can be calibrated and assessed. Thus, the weak points of the optimizer can be determined and eliminated. An evolutionary improvement takes place.

As a search strategy, A^* search enriched by ballooning has been proposed. By subdividing the costs for each alternative into history and future costs, A^* search is able to compare the possibly unfinished plans with each other. However, even in states where the way of building efficient plans is obvious, pure A^* search might generate a large number of alternatives. To alleviate this, ballooning was designed to accelerate the optimizer without degrading its quality.

The viability of our approach was shown by the GOM Blackboard Optimizer. Based on an object-oriented algebra, a blackboard optimizer was specified. It was shown how a blackboard, its regions, and its knowledge sources could be designed. The search algorithm was explained and the basics of a cost model were described.

For illustration purpose a sample optimization was demonstrated.

Acknowledgement This work was supported by the German Research Council DFG under contracts Ke 401/6-1 and SFB 346/A1.

We thank the participants of the Dagstuhl seminar on query processing organized by J. C. Freytag, D. Maier, and G. Vossen, and the attendees of a talk one of the authors gave on invitation by U. Dayal for fruitful discussions. We also gratefully acknowledge our students K. Hauten, A. Roemer, S. Voss, and R. Waurig who have implemented the first prototype.

---

[2] Actually, in order to introduce $\sigma_{um=om}$ the expansions χ_{ud} and χ_{od} have to be added before. This detail is omitted, since the comparison of the items obtained after incorporating the joins gives an idea about the importance of the future cost estimates.

References

[1] J. Banerjee, W. Kim, and K. C. Kim. Queries in object-oriented databases. In *Proc. IEEE Conf. on Data Engineering*, pages 31–38, L.A., USA, Feb 1988.

[2] D. S. Batory. Extensible cost models and query optimization in GENESIS. *IEEE Database Engineering*, 10(4), Nov 1987.

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

[4] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–392, San Diego, USA, Jun 1992.

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

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

[7] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. IEEE Conf. on Data Engineering*, pages 209–218, Wien, Austria, Apr 1993.

[8] T. Ibaraki and T. Kameda. Optimal nesting for computing N-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.

[9] Y. E. Ioannidis and Y. C. Kang. Cost wells in random graphs. Submitted for publication, Jun 1992.

[10] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 9–22, San Francisco, USA, 1987.

[11] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 258–268, Denver, USA, May 1991.

[12] 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, USA, May 1990.

[13] 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.

[14] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In J.-C. Freytag, D. Maier, and G. Vossen, editors, *Query Optimization in Next-Generation Database Systems.* Morgan-Kaufmann, 1993. (forthcoming).

[15] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architekture for query optimization in object bases. Technical Report #92-31, RWTH Aachen, 1992.

[16] R. Lanzelotte and J.-P. Cheiney. Adapting relational optimisation technology for deductive and object-oriented declarative database languages. In *Database Programming Languges Workshop*, pages 322–336, Nafplion, Greece, August 1991.

[17] R. S. G. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 363–373, Barcelona, Spain, Sep 1991.

[18] 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, Chicago, USA, 1988.

[19] G. Mitchell, S. B. Zdonik, and U. Dayal. An architecture for query processing in persistent object stores. In *Proc. Hawaii Intl. Conference on System Sciences*, 1992.

[20] J. Pearl. *Heuristics.* Addison-Wesley, Reading, Massachusetts, 1984.

[21] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 256–276, Boston, USA, Jun 84.

[22] A. Rosenthal and U. S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 230–239, L.A., USA, Sep 1988.

[23] A. Rosenthal and D. Reiner. An architecture for query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 246–255, Jun 1982.

[24] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, Jun 1979.

[25] T. K. Sellis. Multiple-query optimization. *ACM Trans. on Database Systems*, 13(1):23–52, Mar 1988.

[26] D. D. Straube and M. T. Özsu. Execution plan generation for an object-oriented data model. In *Proc. of the Intl. Conf. on Database Theory (ICDT)*, pages 43–67, Munich, F.R.G., Dec 1991, LCNS # 470, Springer-Verlag.

[27] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, Apr 1977.