Optimizing Disjunctive Queries with Expensive Predicates*

A. Kemper† *G. Moerkotte*‡ *K. Peithner*† *M. Steinbrunn*†

†Universität Passau
Fakultät für Mathematik und Informatik
94030 Passau, Germany
(lastname)@*db.fmi.uni-passau.de*

‡Universität Karlsruhe
Fakultät für Informatik
76128 Karlsruhe, Germany
moer@ira.uka.de

Abstract

In this work, we propose and assess a technique called *bypass processing* for optimizing the evaluation of disjunctive queries with expensive predicates. The technique is particularly useful for optimizing selection predicates that contain terms whose evaluation costs vary tremendously; e.g., the evaluation of a nested subquery or the invocation of a user-defined function in an object-oriented or extended relational model may be orders of magnitude more expensive than an attribute access (and comparison). The idea of bypass processing consists of avoiding the evaluation of such expensive terms whenever the outcome of the entire selection predicate can already be induced by testing other, less expensive terms. In order to validate the viability of bypass evaluation, we extend a previously developed optimizer architecture and incorporate three alternative optimization algorithms for generating bypass processing plans.

1 Introduction

During the past few years we have witnessed tremendous efforts in optimizing "next-generation" database systems—see, e.g., [FMV93]. One particularly important aspect is the optimization and efficient processing of declarative queries. [Fre87, GD87, Loh88] made rule-based query optimization popular, which was later adopted in the object-oriented context [OS90, KM90, CD92]. Many researchers have worked on optimizer architectures that facilitate flexibility: [GD87, Bat86, BMG93, GM93] are proposals for optimizer generators; [BG92, HFLP89] described extensible optimizers in the extended relational context; [MDZ93, KMP93] proposed architectural frameworks for query optimization in object bases.

*This work was supported by the German Research Council under contracts DFG Ke401/6-1 and SFB 346.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

[HS93] pointed out that the ordering of the selection predicate evaluation is particularly important in the presence of expensive conditions. These may occur in relational systems in the form of nested subqueries or in extended relational and object-oriented systems in the form of user-defined functions. [HS93]'s work is based on ordering the conditions in a sequence according to their relative selectivity and evaluation cost[1]. This approach yields the optimal evaluation sequence for (purely) *conjunctive* selection predicates [MS79]. An obvious extension towards predicates containing disjunctions is to convert the entire selection predicate into conjunctive normal form (CNF) and apply the sorting on the resulting Boolean factors [SAC+79]. Furthermore, the disjuncts within one Boolean factor could be ordered according to selectivity and evaluation cost. Then, the evaluation within a Boolean factor can be stopped as soon as one condition evaluates to *true*, and the evaluation of the predicate is "exited" as soon as one Boolean factor evaluates to *false*—or, as soon as the last Boolean factor in the sequence evaluates to *true*.

Another proposed technique (e.g., [OS90, JK84]) of evaluating disjunctive selection predicates consists of transforming the predicate into disjunctive normal form (DNF). Then, the *or*'s are usually transformed into a multiway union, and each resulting stream is optimized as usual. This approach facilitates parallel processing. However, the top union necessitates the computationally expensive duplicate elimination—because the same object may qualify in more than one stream.

In this paper, we investigate an evaluation technique, called "*bypass processing*," which generalizes the idea of avoiding the evaluation of expensive and/or non-selective predicates whenever possible. In comparison to the CNF- and DNF-based evaluation, the search space for an efficient bypass plan is expanded. The CNF-(DNF-) based optimization cannot rearrange conditions across Boolean factors (conjunction terms). Therefore,

[1]In the presence of joins the work relies on a pre-determined join order and sorts the conditions within the join tree.

the search space is limited because the granules of the optimization, i.e., the Boolean factors (CNF) or the conjunction terms (DNF), are non-atomic. On the other hand, the "bypass optimization" consists of determining a (near) optimal evaluation sequence on the basis of atomic conditions (not restricted to Boolean factors or conjunction terms).

As an example, consider the predicate P which consists of three conditions P_1, P_2, and P_3: $P(o) = (P_2(o) \vee P_1(o)) \wedge (P_3(o) \vee P_1(o))$. For simplicity, assume that all three predicates have the same (high) evaluation cost and the same selectivity. Then, the bypass technique will first try to determine the outcome of the predicate P solely on the basis of P_1, which is the most influential in this example. That is, those objects o for which $P_1(o)$ is satisfied are to be included into the result and only those objects o' for which $P_1(o')$ is not satisfied are further processed. Let us assume that they are next checked on the basis of P_2. Objects not satisfying P_2 can be discarded since they did satisfy neither P_1 nor P_2 and therefore cannot satisfy P. Only those objects o'' that satisfy P_2 are further processed by P_3. This way the condition P_3 is evaluated on only those objects not satisfying P_1 but satisfying P_2—which is the minimal set of objects that have to be tested on P_3. For all other objects, the evaluation of P_3 is bypassed. In this paper, we present techniques for efficiently generating such bypass plans that are based on atomic predicates.

In order to verify the viability of the approach, we developed an optimizer for generating query evaluation plans for bypass processing. This optimizer is an instantiation of our previously developed architectural framework for object-oriented query optimization [KMP93]. This architecture is centered around a blackboard structure divided into regions—an idea also proposed by [MDZ93]—which contain incomplete query evaluation plans (QEPs). QEPs are composed from basic building blocks [Loh88] which are, step by step, augmented to complete evaluation plans. This building block approach—also used for global query optimization [Sel86]—incorporates the factorization of common subexpressions (cf. [CD92]) and the early pruning of non-promising alternatives. Local optimization algorithms (in our terminology called knowledge sources) are associated with the regions and carry out the augmentation of the (still incomplete) QEPs until, at the top-most region, complete QEPs are generated. Each knowledge source is allowed to restrict its search space and to use its own search strategy as it is also suggested by [LV91]. The (global) optimization process is controlled by A* search [Pea84], which advances the alternative which has the least expected cost. In contrast to the approaches [GD87, GM93], where an expected cost factor is assigned to the transformation rules, the

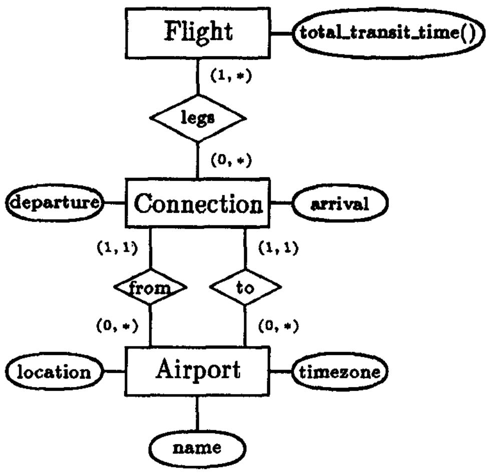


Figure 1: ER schema of the Airline Reservation System

building block approach allows us to derive the expected cost of an alternative from the state of its completion.

For purpose of comparison, we devised four alternative knowledge sources—three of which generate QEPs for bypass processing and one for DNF-based processing. In one of these knowledge sources we apply a formerly developed heuristic based on the Boolean Difference Calculus [KMS92], which ranks conditions on the basis of their evaluation cost and their significance for obtaining the predicates' outcome.

In Section 2 the idea of bypass evaluation is illuminated on an illustrative example. In comparison to conventional selection processing, the advantages of bypass processing are shown. Then, in Section 3 our optimization framework, into which the bypass optimization is integrated, is described. In Section 4 the four above-mentioned knowledge sources are explained. In Section 5 we assess the quality of these four optimization techniques based on a set of generated example queries. Section 6 concludes this paper.

2 The Bypass Technique

We illustrate the ***bypass technique*** by an example taken from the airline reservation domain; the entity relationship schema is shown in Figure 1.

Consider the following example query retrieving those flights which are extremely tiring. These can be characterized as eastward journeys spanning more than three time zones and consisting of more than three connections or with a total transit time (sum of all the times the passenger spends waiting in transit) of more than four hours. In the query language GOMql [KM94], this query can be expressed as in Figure 2, where the total transit time is computed by the function *total_transit_time*() associated with the type *Flight*.

```
range f: Flight
retrieve f
where
 f.legs.last().to.timezone − f.legs.first().from.timezone ≥ 3
   and f.legs.length() ≥ 3
 or f.total_transit_time() ≥ 4
```

Figure 2: Example Query "Tiring Flights"

Condition	Cost c	Selectivity s
P_{tz}	18	0.6
P_{length}	3	0.4
P_{time}	40	0.7

Table 1: Costs and selectivities for "Tiring Flights"

This function adds up all the transit times (time difference between arrival of a flight and the departure of the succeeding flight) in order to determine the total. As this computation requires iteration over the entire set of connections that make up the complete journey, this function is more expensive to evaluate than the other two.

The functions *first*, *last*, and *length* are built-in operations on list-structured types. For brevity, we denote the conditions of the selection predicate by P_{tz}, P_{length} and P_{time}:

$$\begin{aligned} P_{tz} &:= \text{f.legs.last().to.timezone} - \\ &\qquad \text{f.legs.first().from.timezone} \geq 3 \\ P_{length} &:= \text{f.legs.length()} \geq 3 \\ P_{time} &:= \text{f.total\_transit\_time()} \geq 4 \end{aligned}$$

Consequently, the query's selection predicate becomes $(P_{tz} \wedge P_{length}) \vee P_{time}$. Apart from the query and the schema, we assume costs and selectivities to be available to the optimizer. The evaluation costs per object are denoted by c, and the selectivities by s. For the example query, these figures are given in Table 1.

Let us now contrast the bypass technique with evaluation plans based on transforming the selection predicate into the disjunctive normal form (DNF) or the conjunctive normal form (CNF) [JK84]. The evaluation plans for both alternatives are depicted in Figure 3a–c. For the DNF-based plan, the set of objects that are to be processed is duplicated such that one of the copies of the object set is directed to the path on the left-hand side (starting with the condition P_{time}), the other to the path on the right-hand side (starting with P_{length}). Those objects that satisfy a condition are moved onwards; those that do not are immediately discarded. If the object set initially consists of 1000 objects, on average 700 pass P_{time} and 400 pass P_{length}. From these 400 objects, 240 pass P_{tz} and are subsequently united with the 700 objects that have satisfied P_{time}. Because these two remaining sets are generally not disjunct, the final union operation must eliminate duplicate objects in order to generate the result set that consists of an average of 772 objects. The average cost per object can be computed as

$$c(P_{time}) + c(P_{length}) + s(P_{length}) \cdot c(P_{tz}) = 50.2,$$

the cost for eliminating the duplicates not included.

The second approach, based on the conjunctive normal form, avoids the costly elimination of duplicates. Two possible CNF-based plans for the example predicate are depicted in Figure 3b and 3c. An object moves on to the next stage (i.e., the next "Boolean factor," cf. [SAC+79]) as soon as it is certain that it qualifies. For instance, if an object does not pass P_{length}, P_{time} is evaluated. If it does not pass P_{time} as well, it is discarded—otherwise, it is an element of the result because P_{time} also appears in the second stage. If P_{time}'s evaluation results are not saved, the sorting of Boolean factors and conditions within Boolean factors as in plan 3b is the optimum (cf. [Han77]). The average cost per object for plan 3b can be computed as

$$\begin{aligned} & c(P_{length}) + (1 - s(P_{length})) \cdot (c(P_{time}) + \\ & \quad s(P_{time}) \cdot (c(P_{tz}) + (1 - s(P_{tz})) \cdot c(P_{time}))) + \\ & \quad\quad s(P_{length}) \cdot (c(P_{tz}) + (1 - s(P_{tz})) \cdot c(P_{time})) = 54.88, \end{aligned}$$

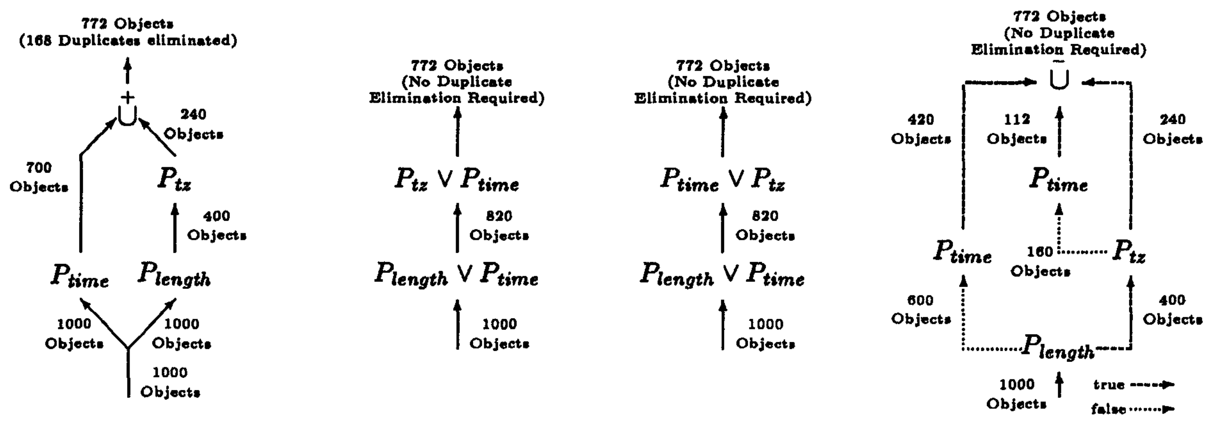
that is higher than the DNF plan's cost (neglecting duplicate elimination) chiefly because it may be necessary to evaluate P_{time} twice (namely if P_{length} is *false*, P_{time} is *true* and P_{tz} is *false*).

However, if a cache for P_{time}'s evaluation results is available as proposed in [HS93], plan 3c is the optimum, and the cost can—at best—be reduced to an average per object of

$$\begin{aligned} & c(P_{length}) + c(P_{time}) + \\ & \qquad s(P_{length}) \cdot (1 - s(P_{time})) \cdot c(P_{tz}) = 45.16 \end{aligned}$$

if zero lookup and maintenance cost for cached results is assumed.

The advantage compared to the DNF-based plan is not only the lower evaluation cost average, but also the avoidance of a duplicate-eliminating union operation. Furthermore, the optimal ordering of Boolean factors and conditions within a Boolean factor can be determined efficiently [Han77]. Objects that are not elements of the result set can be eliminated early, namely as soon as one Boolean factor evaluates to "*false*." However, for elements of the result set, it may be necessary to evaluate certain conditions repeatedly. For instance, if an object satisfies P_{time} but neither P_{length} nor P_{tz}, P_{time} must be evaluated twice



(a) DNF-based plan

(b) CNF-based plan without result caching

(c) CNF-based plan with result caching

(d) Bypass technique

Figure 3: Various evaluation plans for $(P_{tz} \wedge P_{length}) \vee P_{time}$

because it appears in both Boolean factors. This is an inherent problem of both CNF- and DNF-based plans and makes a result cache indispensable if competitive performance is desired.

The fourth evaluation plan, employing the bypass technique, pursues a similar idea as the CNF technique, namely to discard objects that are not elements of the result set as early as possible, but to avoid the CNF plans' drawbacks. In order to achieve this goal, the plan makes use of a special selection operator that does not merely determine those objects that satisfy the selection predicate, but distinguishes the input object set into two output sets consisting of objects that satisfy the predicate and those that do not, respectively. In Figure 3d, the "*false*" set is output on the left-hand side (dotted line), the "*true*" set on the right-hand side (broken line). For instance, an object that satisfies condition P_{length} moves on to condition P_{tz}. If it satisfies P_{tz}, too, it is certain to be an element of the result set. Otherwise, P_{time} has to be evaluated, whose outcome finally determines the object's fate. In the last step of the evaluation plan, the three streams are united in order to compute the result set. In contrast to the union operation in Figure 3a, this one does not need to eliminate duplicates, because its operand sets are always disjunct. To distinguish these two types of unions, the duplicate-eliminating set union is denoted as $\overset{+}{\cup}$ and the merge union as $\bar{\cup}$. The average cost per object for the bypass evaluation plan is

$$c(P_{length}) + (1 - s(P_{length})) \cdot c(P_{time}) +$$
$$s(P_{length} \cdot (c(P_{tz}) + (1 - s(P_{tz})) \cdot c(P_{time})) = 40.6,$$

that is the lowest of the alternative plans. Furthermore, no result cache is required because at each node of the evaluation plan, the results of all prior evaluations are implicitly known. The set of these specialized bypass plans is a superset of the set of CNF-based plans: every CNF-based plan implicitly specifies a bypass plan, but not every bypass plan can be expressed as a CNF plan. Restricting the evaluation to CNF-based plans one can order Boolean factors and, in addition, the disjuncts within each Boolean factor. However, it is not possible to rearrange the atomic conditions globally "across" Boolean factors. On the other hand, an optimal bypass plan treats Boolean conditions as atomic units that are ordered *globally* for minimum average cost. Consequently, well-designed bypass plans are at least as good as conventional CNF- or DNF-based evaluation plans.

This example motivates the enhancement of the query optimizer such that it can generate optimized bypass plans. During the rest of the paper, we will consider several strategies to find optimized bypass plans and compare their performance in terms of the costs of their generated evaluation plans and their (optimization) running times.

3 Optimization Framework

In [KMP93], a (generic) architectural framework for query optimization was introduced. This framework is based on a *blackboard* architecture which achieves desirable characteristics of a query optimizer as extensibility, adaptability, and evolutionary improvement. Especially, the integration of new optimization techniques can be carried out with acceptable effort. This allows us to test several algorithms for optimizing selection predicates for the bypass processing technique.

The blackboard query optimizer is based on a *building block approach* [Loh88]. The entire query is decomposed into building blocks, and alternative query evaluation plans are obtained by composing these blocks. The former—usually called simplification—normalizes and reduces the query to a set of atomic building blocks. The latter step—(re-)assembling by composing the blocks to different query evaluation plans—is actually the optimization process which is well supported by our blackboard architecture.

We will first sketch the building block approach, and then describe an instantiation of the generic framework tailored for optimizing disjunctive queries.

3.1 Building Block Approach

In order to obtain an unambiguous notation, each term factor and each condition is assigned an identifier—denoted by T, T_0, T_1, ..., P, P_0, P_1, ..., etc.[2] This facilitates a decomposition into building blocks, and a factorization of common subexpressions. For example, the condition $f.legs.length() \geq 3$ will be decomposed into two term factors T_3: *f.legs* and T_2: *length*(T_3) and one condition $P_{length}: T_2 \geq 3$. Thereby, the term *f.legs* appearing three times in the query of Figure 2 is implicitly factored into only one algebraic operation T_3: *f.legs*.

Thus, the selection predicate of the entire query will be decomposed into *atomic operations*. The operations are applied on intermediate results that are maintained in relations whose columns can be denoted by the term identifiers. For this presentation, the following algebraic operations will be sufficient:

1. *Selections* $\sigma_{P:l\Phi r}$ where P identifies the condition, l and r are constants or term identifiers, and Φ is a comparison operator ($=$, $\neq$, $<$, $\geq$, $>$, $\leq$, $\in$, $\notin$). The outputs of a selection are two (complementary) streams of tuples of the input relation corresponding to the evaluation of the condition $l\Phi r$; that is, a *true*-stream and a *false*-stream.

2. *Function invocations* $\chi_{T:g(args)}$ where T identifies the function term, g is the function name, and *args* is a list of term identifiers and constants. The semantics of a function invocation is a mapping from the input relations to the result relation containing all previous columns and, in addition, one newly generated column identified by T. For type-associated functions, the receiver object is specified by the first parameter of *args*.

[2]For the running example, we will use the (mnemonic) subscripts introduced in Section 2.

	Building Blocks	Single Costs	Selectivities
Anchor Sets	$\{\varrho_{f=\#0}(oid(Flight))\}$	1,000.0	–
Expansions	$\chi_{T_3:f.legs}$	1.0	–
	$\chi_{T_{12}:first(T_3)}$	1.0	–
	$\chi_{T_7:last(T_3)}$	10.0	–
	$\chi_{T_2:length(T_3)}$	1.0	–
	$\chi_{T_{13}:T_{12}.from}$	1.0	–
	$\chi_{T_{10}:T_{13}.timezone}$	1.0	–
	$\chi_{T_8:T_7.to}$	1.0	–
	$\chi_{T_5:T_8.timezone}$	1.0	–
	$\chi_{T_1:-(T_5\ T_{10})}$	1.0	–
	$\chi_{T_{15}:total\_transit\_time(f)}$	39.0	–
Selections	$\sigma_{P_{length}:T_2 \geq 3}$	1.0	0.4
	$\sigma_{P_{tz}:T_1 \geq 3}$	1.0	0.6
	$\sigma_{P_{time}:T_{15} \geq 4}$	1.0	0.7
Control Function	$(P_{tz} \wedge P_{length}) \vee P_{time}$	–	–

Table 2: The running example's *MCNF*

3. *Attribute accesses* $\chi_{T_2:T_1.a}$ where T_1 and T_2 are term identifiers, and a is the name of the accessed attribute. The input relation will be augmented by the column T_2 containing the values obtained by the attribute access $T_1.a$.

4. *Union* operators which will be distinguished into unions $\overset{+}{\cup}$ with duplicate elimination and unions $\overset{-}{\cup}$ only merging the disjoint input streams (without the need for duplicate elimination).

The algorithms presented in the next section will only consider queries decomposed into operations using the algebraic operators listed above. The complete set of operators in our algebra also contains a *join*, an *unnest*, a *projection*, and a *set difference* operator. The function invocations and the attribute accesses are generalized to *expansions* χ—an operator similar to the *materialize* operator as proposed in [BMG93].

The simplification step decomposes the query into its building blocks, and enriches them with statistical data, called *basic values*. For the purpose of the paper, we are mainly interested in selectivities and evaluation costs for function invocations. Among other things (explained below) building blocks and the basic values are stored in a (normalized) tabular format—called *MCNF*[3] [KM93]—as presented for the running example in Table 2.

A minimal (i.e., non-redundant) set of building blocks used for scanning the base relations, e.g., extensions of

[3]*MCNF* stands for *Most Costly Normal Form* indicating that this form of a query represents the initial—most costly—state of the optimization.

object types or index structures, is called an *anchor set.* To simplify this discussion, only one alternative anchor set, namely $\{\varrho_{f=\#0}(oid(Flight))\}$, is considered—the "real" optimizer has to consider alternative anchor sets to exploit index structures [KMP93]. This anchor set creates a new temporary relation with one column, called f, scans the extension of *Flight*, and associates the object identifiers (*OIDs*) with the attribute f. In total, the decomposition of the example query contains ten expansions and three selections. The evaluation costs for processing one single tuple and the selectivities of the selections are reported in the columns *Single Costs* and *Selectivities* of Table 2. If the costs of the expansions necessary for evaluating one selection are summed up and added to the costs of the selection itself, the costs we assumed in Section 2 will be obtained. For example, the selection marked by P_{length} presupposes the evaluation of the expansions $\chi_{T_2:length(T_3)}$ and $\chi_{T_3:f.legs}$ such that its costs amount to $1+1+1=3$. The *Control Function* is derived from the selection predicate by substituting the conditions with their corresponding identifiers.

3.2 Composition

Query evaluation plans are assembled bottom-up where the anchor sets form the basis. We call a not necessarily complete query evaluation plan a *current expression.* Hence, anchor sets are the smallest current expressions. The assembly process continues by adding selections and expansions to a current expression. Different applicable operations give rise to different alternatives. For the optimization problem considered here, adding a union, either $\overset{+}{\cup}$ or $\overset{-}{\cup}$, typically ends the assembly.

The composition process is "directed" by *control functions* assigned to the current expressions. At the beginning of the optimization, the control function of *oid*(*Type*) is taken from the *MCNF*. Every time a selection is appended to an expression the control function is updated, i.e., for the *false*-branch, the concerned identifier P is substituted by *false*, and for the *true*-branch by *true*. Then, the control function is simplified by the common algebraic simplification rules such as $false \wedge P = false$. For example, if we add $\sigma_{P_{length}:T_2\geq 3}$ to an expression with control function $(P_{tz} \wedge P_{length}) \vee P_{time}$, we will obtain *two* current expressions corresponding to the two evaluation streams. The *true*-stream is controlled by $(P_{tz} \wedge true) \vee P_{time}$, which is simplified to $P_{tz} \vee P_{time}$ and the *false*-stream by $(P_{tz} \wedge false) \vee P_{time}$, which simplifies to P_{time}.

Looking at the control function, the optimizer is able to decide which selections and, furthermore, which expansions are necessary for completing each expression such that, finally, its control function equals *true*. This is the precondition for the final merge by the $\overset{-}{\cup}$ operator.

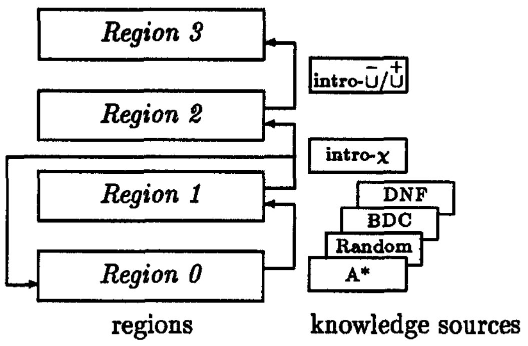


Figure 4: Blackboard architecture

3.3 Blackboard Architecture

The composition process is reflected by a blackboard architecture which is an instantiation of our generic blackboard framework [KMP93]. As sketched in Figure 4, the blackboard for this presentation consists of four regions between which the query evaluation plans are composed step by step. Each region symbolizes one state of the optimization process where the alternatives—called *items*—are temporarily maintained. Each item may contain several current expressions which will be enhanced and composed to a query evaluation plan. The propagation of the items from one region to the next region is performed by *knowledge sources* which might be atomic rules or sophisticated algorithms. Each knowledge source is allowed to generate alternative items. It is possible to associate more than one knowledge source between two regions. Then, the characteristics of an item determines which knowledge source will be applied.

The first region contains the *MCNF* of the entire query. Adding some selections to the current expressions is tried first—thereby transferring the item from *Region 0* to *Region 1*. For that, four alternative knowledge sources called *A**, *Random*, *BDC*, and *DNF* specified in the following section were developed. If a selection depends on an expansion, the item will be propagated to *Region 1* such that the knowledge source *intro-*χ can enhance this item by adding the necessary expansions. The generated items are put back to *Region 0* as long as any of the control functions indicates—by not being reduced to *true* or *false*—that some conditions still have to be integrated. Finally, the item leaves the iteration and, after incorporating the final union operator by the knowledge source *intro-*$\overset{-}{\cup}/\overset{+}{\cup}$, an (alternative) query evaluation plan is generated and stored into *Region 3*. This instantiation of the generic blackboard framework can easily be extended by further optimization heuristics, as e.g., the determination of a good join order (cf. [SMK93]), if some more regions with the appropriate knowledge sources are integrated into the iteration.

The optimization process is controlled by the global *search strategy* A^* [Pea84]—also used for global query optimization [Sel86]. For that, ***history*** and ***future*** costs derived from the state of the composition are assigned to each item. The current expressions determine the history, and the remaining operations—called *future work*—the future costs. At all times, the item for which the sum of history and future costs is minimal over all items is further processed.

4 Disjunctive Predicate Optimization

In this section, we introduce three algorithms for optimizing queries for bypass processing—A*, Random, and BDC—and one algorithm based on the DNF selection processing. All these algorithms determine an order of how the selections should be evaluated. Their placement within the blackboard architecture is shown in Figure 4. Random and BDC integrate all conditions within one Blackboard iteration, whereas A* and DNF require some more iterations in general. The results of these four search algorithms are demonstrated on the running example.

4.1 A* Search

Since the global search strategy is based on the A* search algorithm, a knowledge source using A* is easy to integrate. It is sufficient to establish one knowledge source which generates all possibilities of adding selections to the current expressions. The global A* search then controls which alternatives are further processed.

This simple knowledge source can easily be combined with further knowledge sources that optimize the orders of other algebraic operators. Furthermore, since the A^* algorithm will always find the optimal solution in a given search space if the estimated future costs constitute lower bounds of the actually arising costs—a constraint which is fulfilled by our cost model—the optimal bypass evaluation plan is obtained for each query. Consequently, for the running example, the A* strategy computes the optimal evaluation plan as it was sketched in Figure 3d (Section 2).

4.2 Random Search

The knowledge source called Random uses a very simple procedure to order the conditions into a bypass plan. It iterates over the future work of the entire item, and integrates an operation whenever possible. Since at each iteration at least one operation of the future work is integrated into the current expressions, the worst case complexity is $O(n^2)$ with n being the number of building blocks.

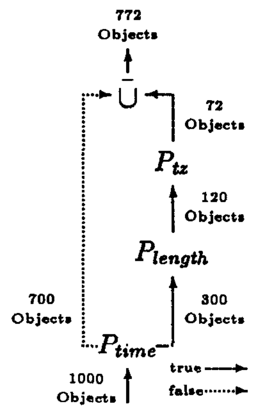


Figure 5: Random search

The query evaluation plan generated by Random is sketched in Figure 5. The condition P_{time} is integrated first since it depends on the least number of expansions and, therefore, it can be added to the current expressions in an early iteration. The condition P_{tz} requires the most expansions, so it is integrated last. The generated plan induces an average cost of 43.06 per object.

4.3 Boolean Difference Calculus Heuristic

In contrast to the algorithms described so far, the Boolean Difference Calculus heuristic (BDC heuristic for short) does not employ any search. Instead, the heuristic determines a good evaluation order of the selection predicate's conditions in one step. The basic idea is to assign "weights" to the conditions, depending on the respective influence on the result (e.g., in $x_1 \vee (x_2 \wedge x_3 \wedge x_4 \wedge x_5)$, x_1 is more influential than any one of $x_2, \ldots, x_5$) and on the evaluation cost per object.

The tool for computing the influence of a condition is the so-called *Boolean Difference Calculus.* The Boolean difference of the Boolean function f for the variable x_i is defined analogously to the differential in "ordinary" calculus, namely:

$$\begin{aligned}\Delta_{x_i} f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) &\overset{\text{def}}{=} \\ f(x_1, \ldots, x_{i-1}, x_i = \mathit{false}, x_{i+1}, \ldots, x_n) &\not\equiv \\ f(x_1, \ldots, x_{i-1}, x_i = \mathit{true}, x_{i+1}, \ldots, x_n)&\end{aligned}$$

The probability for the Boolean difference being *true* is a measure of the influence of the variable x_i on the result of the function f, similarly to the gradient of a continuous function: the higher the gradient, the higher the "leverage" of the variable x_i with respect to the function's value. The same principle applies to the Boolean difference: the higher the probability (=selectivity) $s(\Delta_{x_i}f)$ of $\Delta_{x_i}f$ being *true*, the higher the influence of the variable x_i on the value of the Boolean function f. This idea is employed to construct an evaluation plan for the Boolean function f, where the above-mentioned weight w_{x_i} of a condition x_i is the quotient of its Boolean difference probability and its evaluation cost $c(x_i)$: $w_{x_i} = s(\Delta_{x_i}f)/c(x_i)$. The algorithm for constructing the evaluation plan reads as follows (cf. [KMS92]):

1. Compute the weights w_{x_i} for all conditions x_i in the predicate f.
2. Choose x_i with the highest weight w_{x_i} as the first (next) processing node.
3. • If $f(x_i := \mathit{false}) = \mathit{false}$, omit the emanating "false" arc of the processing node x_i (objects are certain not to be elements of the result and can be discarded);
   Stop
   • If $f(x_i := \mathit{false}) = \mathit{true}$, draw the "false" arc from processing node x_i to the final union operation $\bar{\cup}$ (objects are certain to be elements of the result);
   Stop
   • Otherwise, draw the "false" arc from processing node x_i to the first processing node for the predicate $f(x_i := \mathit{false})$ that is determined recursively as in Step 1 and Step 2.
4. Exactly as in Step 3, but for $f(x_i := \mathit{true})$ (i.e., the "true" arc).

Applying this algorithm to the selection predicate "Tiring Flights" (cf. Figure 2) yields (in this case) the optimal evaluation plan as depicted in Figure 3d with an average cost per object of 40.6.

4.4 Generation of DNF-based Plans

For the purpose of comparison, we have also implemented a knowledge source called DNF which optimizes the queries according to the DNF-based selection processing technique. This knowledge source presupposes that the selection predicate is transformed into disjunctive normal form, first. Then, each *and*-connected subpredicate of the normal form can be evaluated by the conventional type of selection processing. Because of transforming the *or*'s into unions ($\overset{+}{\cup}$) we obtain evaluation plans with at least two subbranches. Usually, the subbranches are very similar to each other such that factorization of common subexpressions is expected to gain performance. Since, in general, there are several selections that may be integrated, alternatives are generated which are assessed by the global search strategy.

Figure	# Conditions	Percentage *and*-Operations	# Queries/ Plot Point
6a	3–10	66.67%	30
6b	5	0–100%	30
7a	3–10	66.67%	30
7b	5	0–100%	30
8	5	66.67%	30

Table 3: Benchmark parameters

The query evaluation plan for the running example generated by DNF is sketched in Figure 3a; it induces an average cost per object of 50.02. For this plan, factorization fails since the initial selection predicate is already in disjunctive normal form without any common subexpressions.

5 Assessment

5.1 Generated Benchmark Queries

The benchmarks described in this section were carried out with synthetic queries from a query generator. We want to point out that the generated queries are based on a realistic schema and object base. The schema models an airline reservation schema a part of which is sketched in Figure 1 of Section 2. It has 13 types with average numbers of 3.69 attributes and 0.46 type-associated functions per type and 7 free (not type-associated) functions. In order to be able to add costs to the query evaluation plans some characteristics, as e.g., the cardinalities of the types and the range sizes of the attributes and functions, of the object base are necessary. In addition, some system-dependent values have to be provided. For example, the average costs for an attribute access is 1.0 and the invocation of a type-associated function is 110.0. For our benchmarks, the selectivities of the conditions were derived from the cardinalities of the types and the range sizes of the attributes and functions by simple calculations. The derived selectivities of the conditions of the generated queries varied from 0 to 1. The basic structure of a generated query is

range var: type
retrieve var
where var.attr$_{1,0}$. $\cdots$ .attr$_{1,n_1}$ Φ_1 const$_1$ **and/or**
var.attr$_{2,0}$. $\cdots$ .attr$_{2,n_2}$ Φ_2 const$_2$ **and/or**
...
var.attr$_{m,0}$. $\cdots$ .attr$_{m,n_m}$ Φ_m const$_m$

That means, a set of objects of a single *type* is retrieved. None of the attributes in the query is set- or list-typed, i.e., all attributes are references to single objects.

Furthermore, as the structure of the queries indicates, there are no join operations, because all conditions of the selection predicate are simple comparisons with constants ($\Phi_i \in \{=, \neq, <, >, \ldots\}$). However, the selection predicate itself may be an arbitrary Boolean expression. The following parameters can be adjusted for the query generation:

- The number m of conditions in the selection predicate
- The ratio of the number of *and*-operations to the number of *or*-operations in the selection predicate
- The maximum total number n of attribute accesses and function calls, i.e., $n_i \leq n$.

The third parameter, the maximum total number of attribute accesses and function calls, is the same for all benchmarks we carried out, namely four times the number of selection predicate conditions. This ensures a sufficient variety of queries and, thus, of costs and selectivities. The first two parameters, the number of selection predicate conditions and the *and/or* ratio, are subject to variation in the benchmarks.

5.2 Quality Assessment

Basically, two characteristic features of optimization strategies are of paramount interest: first, the quality of the optimization, i.e., the execution cost of the optimized query compared to the optimal solution, and second, the cost incurred by the optimization process itself compared to the straightforward translation of the query into an evaluation plan without any optimization. The execution costs are estimated by the cost model of the blackboard framework that only assesses the costs of scan operations, the expansions, and the selections, and ignores the costs of the unions. The optimization costs are the running times needed for the optimizations.

In order to determine these figures, we carried out two series of benchmarks, one for the optimization quality and one for the optimization cost. The parameters for all benchmarks are listed in Table 3. In addition, the plot points are computed as the median of all thirty queries with that particular parameter set. The reason for using the median and not the average for combining is the comparatively small number of samples that have been taken, where the expressive power of the median is higher than that of the average.

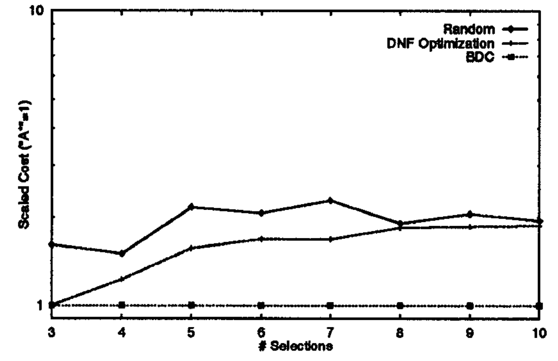
In Figure 6a and 6b, the optimization quality of DNF optimization, BDC optimization and the Random strategy are compared with the A* algorithm. Firstly, we observe that the bypass plans generated by A* are far better than the "conventional" plans generated by DNF. In some cases, the optimization to a conventional plan accomplishes evaluation plans even worse than a bypass plan generated by the Random search. Secondly, we notice that the BDC heuristics performs almost as well as the A* optimization.

Figure 6a shows the median of the scaled (with respect to the cost of the A* optimized query) costs for the knowledge sources Random, BDC, and DNF. The BDC optimization produces plans almost as well as the A* optimization regardless of the number of selection predicate conditions (at least within the interval 3–10), whereas the quality of DNF optimization decreases with increasing number of conditions. For ten conditions, DNF optimization is only slightly better than the Random strategy. However, both are no worse than twice the cost of the optimal solution.

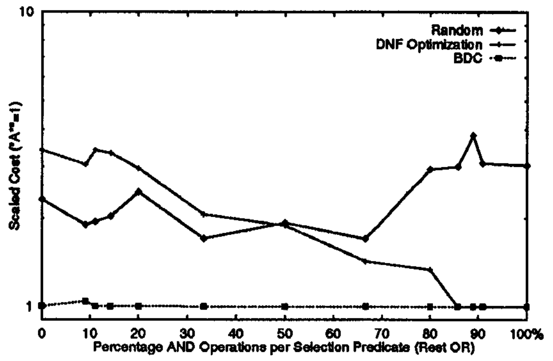
In Figure 6b, the number of selection predicate conditions is always five, but the *and/or* ratio varies. On the left-hand side, the predicates consist entirely of *or* operations, but their number is reduced in favour of *and* operations from left to right until the predicates for the rightmost plot points consist entirely of *and*s. Again, BDC optimization performs for the entire range of *and/or* ratios equally well (and as well as A*), whereas the DNF optimization yields rather bad results for pure disjunctive predicates—many disjunctions lead to many different data streams in DNF-based plans, and, thus, to multiple evaluations of the same conditions in different streams. The quality increases with increasing number of *and* operations, until for mainly conjunctive predicates the DNF optimization works as well as BDC and A* optimization. For the Random strategy, there is no clear tendency perceivable, although the graph indicates a slight favouring of predicates with roughly equal numbers of *and/or* operations.

These results indicate clearly that disjunctive queries, especially if many conditions are involved, are not handled very well by the conventional DNF optimization strategy, i.e., the generated evaluation plan based on conventional selection processing is quite bad. A* as well as BDC outperform DNF optimization by a cost factor of about two to three. Thus, paying special attention to disjunctive queries is certainly well worth the effort. However, as BDC and A* show a similar performance in optimizing the kind of query, the cost for performing the optimization itself plays a decisive rôle. If the cost for the well-performing optimization strategies itself proves to be prohibitively high, even the worse DNF strategy may be preferable. This question is addressed in the second benchmark series, where the running times of the A*, DNF and BDC optimizations are compared with the running time of the Random strategy. The graphs in Figures 7a and 7b correspond to Figures 6a and 6b, respectively.

In Figure 7a, the impact of an increasing number

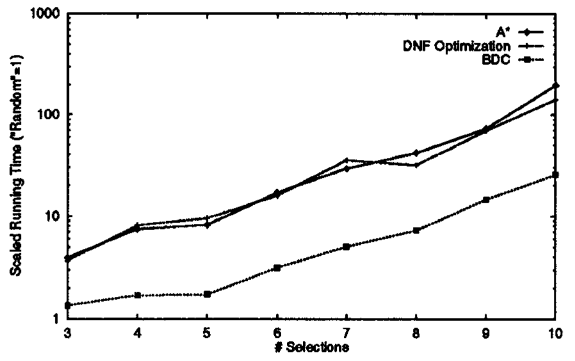


(a)
Increasing number of selection predicate conditions
(66% *and*/33% *or* operations)

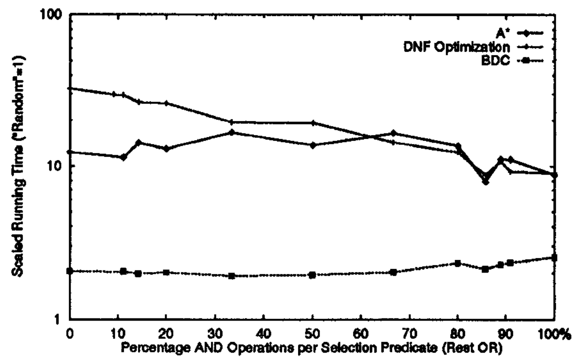


(b)
Varying ratio of *and*/*or* operations
(five selection predicate conditions)

Figure 6: Execution costs of optimized queries



(a)
Increasing number of selection predicate conditions
(66% *and*/33% *or* operations)



(b)
Varying ratio of *and*/*or* operations
(five selection predicate conditions)

Figure 7: Running time for optimization algorithms

of selection predicate conditions is shown. All the algorithms plotted reveal the same exponential time complexity with respect to the number of selection predicate conditions. However, the graph shows that the BDC optimization has a running time that is throughout this range about five times lower, whereas the running times of the A* and DNF optimization are roughly equal.

Figure 7b shows the algorithms' behaviour with respect to varying ratios of *and*/*or* operations in the selection predicate. BDC runs almost as fast as the Random strategy favouring disjunctive queries also in running time, whereas A* and the DNF optimization take a factor of about four longer to complete. For disjunctive queries, A* is faster than the DNF optimization, but as soon as conjunctions take a share of more than about 65%, the running times of A* and DNF optimization are about the same.

In Figure 8, the impact of varying selectivities on the QEPs' quality is shown. 80% of the generated queries' conditions have a selectivity factor as indicated by the x-axis, and the remaining 20% have a random selectivity factor (uniform distribution in the interval $[0, 1]$). As in Figure 6, the costs of the Random strategy, DNF and BDC optimization scaled with respect to the cost of A* are depicted in the graphs. While the BDC strategy is not affected by the variation, the performances of the Random as well as the DNF strategy depend on the prevailing selectivity: the Random strategy works better for high selectivity factors, whereas the DNF strategy yields better results for low selectivity factors. If 80% of all conditions have a selectivity factor that is close to zero, the early evaluation of one of the other 20% is particularly "nasty" for a bypass QEP's evaluation

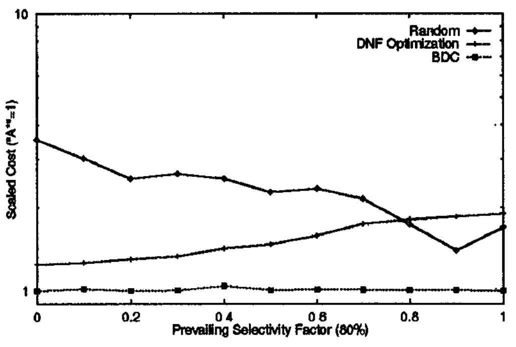


Figure 8: Variation of Sēlectivities

cost, which explains why the Random strategy performs poorly in this case. On the other hand, if most conditions have selectivity factors close to one, this consideration is much less an issue, but the performance of a DNF QEP with many parallel data streams suffers because only few objects can be eliminated early. This results in multiple evaluations of the same conditions in different streams, which, in turn, leads to higher average costs.

These results strongly suggest that optimizers should generate bypass plans for disjunctive queries. The use of the BDC strategy is recommended since—although BDC cannot outperform A* in terms of quality—the running time for BDC is much lower, and, in turn, both are superior to the conventional DNF optimization and the Random strategy. However, the BDC strategy is a specialized knowledge source that cannot be applied, for instance, to queries containing join operations—this requires an appropriate generalization, such as described in [SMK93].

6 Conclusion

In this paper we addressed the problem of optimizing the evaluation of disjunctive queries. The proposed technique—called *bypass processing*—is particularly advantageous when the evaluation costs of the conditions dominate the query processing costs. As pointed out in [HS93], such expensive conditions may occur in object-oriented as well as in (extended and pure) relational systems.

The bypass technique tries to derive the outcome of the selection predicate without evaluating such expensive conditions whenever possible. A specialized form of bypass evaluation can be identified in existing systems where the selection predicate is transformed into disjunctive (DNF) or conjunctive normal form (CNF). Within the CNF the evaluation of a sequence of disjunctions can be stopped as soon as one element evaluates to *true* and the evaluation of the sequence of conjuncts (i.e., the Boolean Factors) can be stopped as soon as one conjunct evaluates to *false*. Within the DNF the evaluation of a sequence of conjuncts can be stopped as soon as one element evaluates to *false* and the evaluation of the sequence of disjuncts can be stopped as soon as one disjunct evaluates to *true*.

As illustrated in the running example of the paper, this kind of bypassing limited to either CNF or DNF does not suffice, because restrictions on the search space are imposed such that the optimal plan is likely to be missed. Therefore, we remove these restrictions by *globally* ordering atomic conditions, which leads to generalized bypass plans.

Then, the viability of our more general approach was assessed by incorporating four algorithms classified as follows into our formerly developed query optimization framework:

- Three algorithms generating query evaluation plans for bypass processing, namely
  1. A*: This algorithm basically covers the entire solution space and directs its search by A*.
  2. BDC: A heuristic that was originally developed for finding near-optimal decision trees for Boolean predicates [KMS92].
  3. Random: A random construction of bypass evaluation plans.
- DNF: An algorithm starting with the disjunctive normal form of the selection predicate and producing a DNF-based query evaluation plan.

On a set of automatically-generated queries based upon a "real" schema we quantified the performance gains. Among the three optimization algorithms the one denoted A* derives optimal bypass query evaluation plans for an arbitrary query. Nevertheless, BDC could be identified as the "winner in almost all classes" since it produces near-optimal query evaluation plans under very low (optimization) running time for a quite large number of queries. In our future research we want to extend this heuristic to queries also containing joins, which cannot be handled by the BDC heuristic as employed in this work and, therefore, up to now have to be optimized by the time-consuming A* search. We are currently working on augmenting the BDC heuristic in order to cover arbitrary relational algebra operators, such as the join [SMK93].

Acknowledgements We thank Barbara Hartmann for implementing the query generator and for carrying out the benchmarks. We gratefully acknowledge Guy Lohman's constructive help in revising the paper.

References

[Bat86] D. S. Batory. Extensible cost models and query optimization in GENESIS. *IEEE Database Engineering*, 9(4), December 1986.

[BG92] 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, June 1992.

[BMG93] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the Open OODB Query Optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 287–295, Washington, DC, USA, May 1993.

[CD92] 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, June 1992.

[FMV93] J.-C. Freytag, D. Maier, and G. Vossen, editors. *Query Processing for Advanced Applications*. Morgan Kaufmann, San Mateo, USA, 1993.

[Fre87] 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.

[GD87] 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.

[GM93] 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, April 1993.

[Han77] M. Z. Hanani. An optimal evaluation of boolean expressions in an online query system. *Communications of the ACM*, 20:344–347, May 1977.

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 377–388, Portland, Or, June 1989.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–276, Washington, DC, USA, May 1993.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[KM90] 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, August 1990.

[KM93] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting the relational techniques. In [FMV93].

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[KMP93] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 543–554, Dublin, Ireland, August 1993.

[KMS92] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing Boolean expressions in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 79–90, Vancouver, B.C., Canada, August 1992.

[Loh88] 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.

[LV91] 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, September 1991.

[MDZ93] G. Mitchell, U. Dayal, and S. B. Zdonik. Control of an extensible query optimizer: A planning-based approach. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 517–528, Dublin, Ireland, August 1993.

[MS79] C. Monma and J. Sidney. Sequencing with series-parallel precedence constraints. *Math. Oper. Res.*, 4:215–224, 1979.

[OS90] M. T. Ozsu and D. D. Straube. Queries and query processing in object-oriented database systems. *ACM Trans. Office Inf. Syst.*, 8(4):387–430, October 90.

[Pea84] J. Pearl. *Heuristics*. Addison-Wesley, Reading, Massachusetts, 1984.

[SAC+79] 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, June 1979.

[Sel86] T. K. Sellis. Global query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 191–205, Washington, USA, June 1986.

[SMK93] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimizing join orders. Technical report MIP-9307, Universität Passau, 94030 Passau, Germany, 1993.