Optimizing Disjunctive Queries in Object Bases*

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

†Universität Passau
Fakultät für Mathematik
und Informatik
94030 Passau, Germany
Tel. (++49-851) 509-321
FAX (++49-851) 509-771
⟨lastname⟩@*db.fmi.uni-passau.de*

‡Universität Karlsruhe
Fakultät für Informatik
76128 Karlsruhe, Germany
Tel. (++49-721) 608-2080
moer@ira.uka.de

Abstract

In this work we propose and assess a novel technique, called *bypass processing*, for optimizing the evaluation of disjunctive queries in object-oriented database systems. In the object-oriented data model, selection predicates often contain terms whose evaluation cost varies tremendously; e.g., the evaluation cost of a user-defined function 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 deduced 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 and—for comparison purpose—one algorithm for generating conventional query evaluation plans. The optimizer is assessed with respect to running time and quality of the obtained query evaluation plans on a (generated) set of queries.

1 Introduction

It is widely agreed upon that performance will be one of the key factors whether or not object-oriented database systems can grasp a substantial share of the information technology market. Therefore, during the past few years we have witnessed tremendous efforts in optimizing object-oriented database systems—see, e.g., [FMV93]. One particularly important aspect is the optimization and efficient processing of declarative

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

queries. Albeit this optimization area has hitherto been somewhat neglected in many of the commercial systems, there are numerous proposals—mostly from the academic world: [GD87, Bat86, BMG93, GM93] are proposals for optimizer generators; [Fre87] has made rule-based optimization popular in the relational context, which was adopted by the object-oriented community [OS90, KM90, CD92] as well as by researchers in extended relational systems [BG92, Loh88]. [MDZ93, KMP93] proposed architectural frameworks for query optimization.

While this (rather incomplete) list indicates the numerous efforts invested in query optimization, it is interesting to observe that all proposals are based on the assumption that selection predicates are conjunctively combined. If not, the prevailing treatment—adopted from the relational context [JK84]—consists of transforming the selection predicate into disjunctive normal form (DNF) and then to optimize each one of the disjuncts separately. Finally, the results obtained for all the disjuncts are merged by a union—including duplicate elimination. Another approach is to first transform the selection predicate into conjunctive normal form and then start the optimization from there.

In this paper we argue that both approaches fail to yield an efficient query evaluation plan in the object-oriented data model. This is mainly due to the observation that selection predicates can be arbitrarily complex (by, e.g., containing terms with computationally complex type-associated operations). Therefore, individual terms constituting the selection predicate may vary tremendously in processing costs—as was also observed by [HS93].

In this paper we propose a new technique for evaluating disjunctive queries. The basic idea consists of "*bypassing*" expensive terms whenever the outcome of the entire selection predicate can be determined without evaluating this particular term. As an example consider the predicate P which consists of three conditions P_1, P_2, and P_3:

$$P(o) \equiv P_1(o) \wedge (P_2(o) \vee P_3(o))$$

Now assume that P_3 is extremely expensive to evaluate, i.e., more expensive than P_1 and P_2. Then, the bypass technique will try to determine the outcome of the predicate P solely on the basis of P_1 and P_2. That is, those objects o for which $(P_1(o) \wedge P_2(o))$ is satisfied are to be included in the result and those objects o' for which $P_1(o')$ is not satisfied can be discarded from the result without even considering P_3. In other words, objects o and o' bypass the evaluation of the term P_3. Only for those objects $\hat{o}$, for which $P_1(\hat{o})$ holds but $P_2(\hat{o})$ does not hold, the (expensive) predicate $P_3(\hat{o})$ has to be evaluated.

In order to verify the viability of the bypass technique 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 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. 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

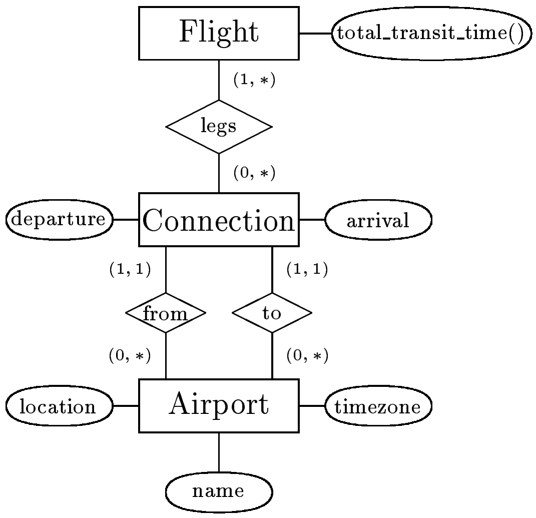


Figure 1: ER schema of the Airline Reservation System

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

For this work, we devised four alternative algorithms—knowledge sources, in our terminology—three of which generate QEPs for bypass processing and one for conventional processing. The latter one was developed for comparison purposes. 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. 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 of the example domain is shown in Figure 1; the object-oriented schema in the syntax of our prototype system GOM is given in Appendix A.

Consider the following example query retrieving those flights which are extremely tiring. These can be characterized as eastward journeys spanning more than three time

Condition	denoted as	Cost c	Selectivity s
f.legs.last().to.timezone – f.legs.first().from.timezone ≥ 3	P_{tz}	18	0.6
f.legs.length() ≥ 3	P_{length}	3	0.4
f.total_transit_time() ≥ 4	P_{time}	100	0.7

Table 1: Costs and selectivities for the conditions in "Tiring Flights"

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 this query can be expressed as

> **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)

where the total transit time is computed by the function *total_transit_time*() associated to the type *Flight*. 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 far more expensive to evaluate than any of 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}:

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

Besides 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_i, and the selectivities by s_i. For the example query, these figures are given in Table 1.

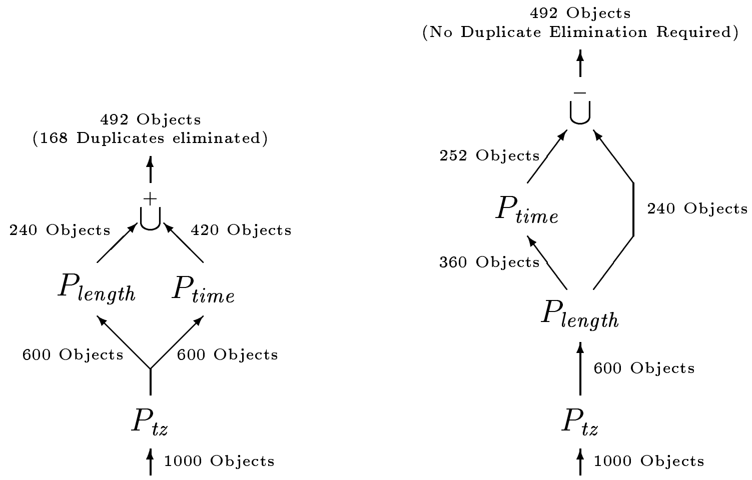
Let us now contrast classical query optimization with the bypass technique. In classical query optimization, the prevailing method for treating queries with disjuncts is to transform the query's predicate into its disjunctive normal form (DNF). For the example query this results in

$$(P_{tz} \wedge P_{length}) \vee (P_{tz} \wedge P_{time})$$

Then, common subexpressions are factored out as much as possible. This yields

$$P_{tz} \wedge (P_{length} \vee P_{time})$$

which accidentally is equivalent to the original predicate. Also, in this case the thus obtained starting point coincides with the conjunctive normal form—another proposed strategy for dealing with disjunctive predicates [JK84]. The corresponding evaluation plan is given in Figure 2a. Its cost can be determined as follows: In the first step, P_{tz} is



(a)
Conventional plan

(b)
Bypass technique

Figure 2: Alternative evaluation plans for "Tiring Flights"

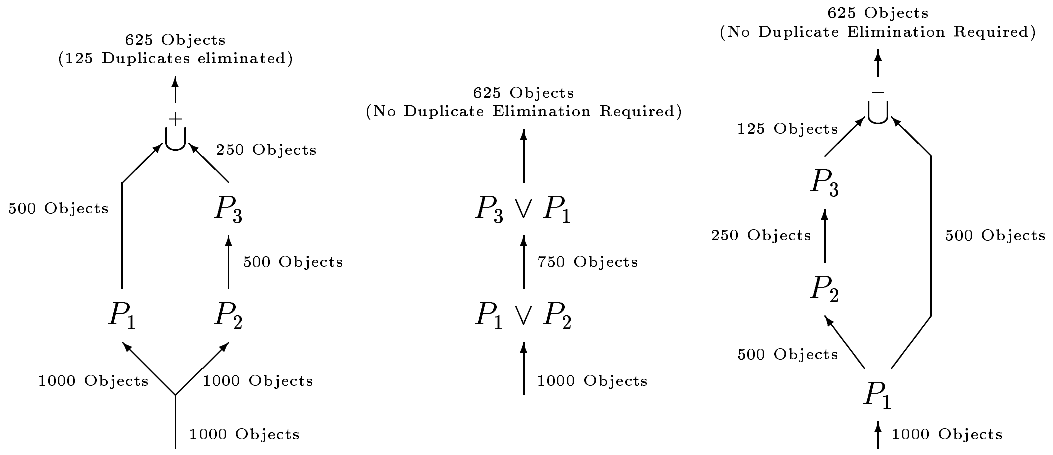
evaluated for each object of type *Flight*. For each object, the cost is $c(P_{tz}) = 18$. Only a fraction of $s(P_{tz})$ passes this test. For these, P_{length} and P_{time} are evaluated. Not counting for the cost of a duplicate eliminating union, the cost of evaluating the query's predicate for a single *Flight* object can be calculated as

$$c(P_{tz}) + s(P_{tz}) \cdot (c(P_{length}) + c(P_{time})) = 18 + 0.6 \cdot (3 + 100) = 79.8$$

The major drawback of the scheme is that *every* minterm of the DNF is evaluated for every single object—no attempt is made to seek for possible "shortcuts." For instance, if for a certain object P_{tz} and P_{length} both evaluate to *true*, it is not necessary to evaluate P_{time}, since it is known that it already qualifies. Furthermore, objects that pass, apart from P_{tz}, both P_{length} and P_{time} are inserted twice into the result, enforcing a potentially expensive elimination of duplicates, denoted by the operator symbol $\overset{+}{\cup}$.

In order to reduce the evaluation cost, we construct an evaluation plan that takes advantage of shortcuts by *bypassing*. In addition, the order of predicate evaluation must take into account the selectivities and costs. For example, expensive predicates should be evaluated as late as possible. Generally speaking, the goal of the paper is to derive optimized bypass plans.

For the example query, the optimal bypass plan is given in Figure 2b. The first evaluated condition, P_{tz}, is the one with the highest influence on the result of the predicate. Objects satisfying this condition subsequently move on to the next condition, P_{length} with cost 3. Objects that pass P_{tz} but do not satisfy P_{length} must further be processed in the left branch, where P_{time} is being evaluated; objects that pass P_{length}, too, are certain to be elements of the result set and bypass the evaluation of P_{time}. Thus, in this case the most expensive condition P_{time} is not evaluated at all. The average cost per object for



(a) DNF-based plan | (b) CNF-based plan | (c) Bypass technique

Figure 3: DNF- vs. CNF-based evaluation plans for $(P_1 \vee P_2) \wedge (P_1 \vee P_3)$

the bypass plan 2b is much lower than for the conventional plan 2a:

$$c(P_{tz}) + s(P_{tz}) \cdot (c(P_{length}) + (1 - s(P_{length})) \cdot c(P_{time})) = 18 + 0.6 \cdot (3 + 0.6 \cdot 100) = 55.8$$

This amounts to a saving of 30% even without considering the costs for eliminating duplicates necessary for plan 2a and avoided in plan 2b (the merge step without duplicate elimination is denoted by the symbol $\bar{\cup}$).

A third alternative, the transformation of the selection predicate into conjunctive normal form (CNF) and basing the evaluation thereupon does not yield better results than the DNF approach. This is illustrated by means of the following abstract example predicate: $(P_1 \vee P_2) \wedge (P_1 \vee P_3)$, where we assume that the selectivities and costs of the conditions are the same, i.e., $s(P_1) = s(P_2) = s(P_3) = 0.5$ and $c(P_1) = c(P_2) = c(P_3) = 100$. The optimal bypass plan (Figure 3c) evaluates P_1 first; P_2 and P_3 are tested only if $P_1 = \mathit{false}$. The average cost of this plan is

$$c(P_1) + (1 - s(P_1)) \cdot (c(P_2) + s(P_2)c(P_3)) = 100 + 0.5 \cdot (100 + 50) = 175$$

Transforming the predicate into DNF yields $P_1 \vee (P_2 \wedge P_3)$, which leads to the evaluation plan in Figure 3a. The average cost of this plan is

$$c(P_1) + c(P_2) + s(P_2) \cdot c(P_3) = 100 + 100 + 0.5 \cdot 100 = 250$$

(+ cost for duplicate elimination). The CNF-based plan is depicted in Figure 3b. An object moves on to the next stage as soon as it is certain that it qualifies. For instance, if an object passes the test P_1 in the first stage, it moves on to the second stage without testing P_2. Therefore, P_1 is the second condition in the disjunction $P_3 \vee P_1$, because objects arriving at this node are certain to be already tested with P_1 in the first node—dependencies of this kind must be detected and taken into account by an optimizer that generates these plans. The average cost for this CNF-based plan is

$$c(P_1) + (1 - s(P_1)) \cdot c(P_2) + (1 - s(P_1)s(P_2)) \cdot (c(P_3) + (1 - s(P_3))c(P_1)) = \\ 100 + 0.5 \cdot 100 + 0.75 \cdot (100 + 0.5 \cdot 100) = 262.5,$$

which is far from the optimal value of 175 although costs and selectivities are the same for all three conditions P_1, P_2 and P_3.

These examples motivate 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 easily. 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 an instantiation of the generic framework tailored for optimizing disjunctive queries is described.

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.[1] 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_0: $f.legs$ and T_1: $T_0.length()$ and one condition P_{length}: $T_1 \geq 3$.

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 the columns of which can be denoted by the term identifiers. For this presentation, the following algebraic operations will be sufficient:

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

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 output 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, one *true*-stream and the *false*-stream.

2. *Function invocations* $\chi_{T:f(args)}$ where T identifies the function term, f 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*.

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 to 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*[2] [KM93]—as it is 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 object types or index structures, is called *anchor set*. To simplify this discussion, only one alternative anchor set, namely $\{\varrho_{T_0=\#0}(oid(Flight))\}$, 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 T_0, scans the extension of *Flight*, and associates the object identifiers (*OIDs*) with the attribute T_0. 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

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

	Building Blocks	Single Costs	Selectivities
Anchor Sets	$\{\varrho_{T_0=\#0}(oid(Flight))\}$	1,000.0	–
Expansions	$\chi_{T_3:T_0.legs}$	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	–
	$\chi_{T_{15}:total\_transit\_time(T_0)}$	99.0	–
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: *MCNF* of the running example

obtained. The *Control Function* is derived from the selection predicate by substituting the conditions by 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 as, e.g., $false \wedge P = false$. 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 receive *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} and the *false*-stream by $P_{tz} \wedge (false \vee P_{time})$ which simplifies to $P_{tz} \wedge P_{time}$.

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 its control function equals *true*. This is the precondition for the final merge by the $\overset{-}{\cup}$

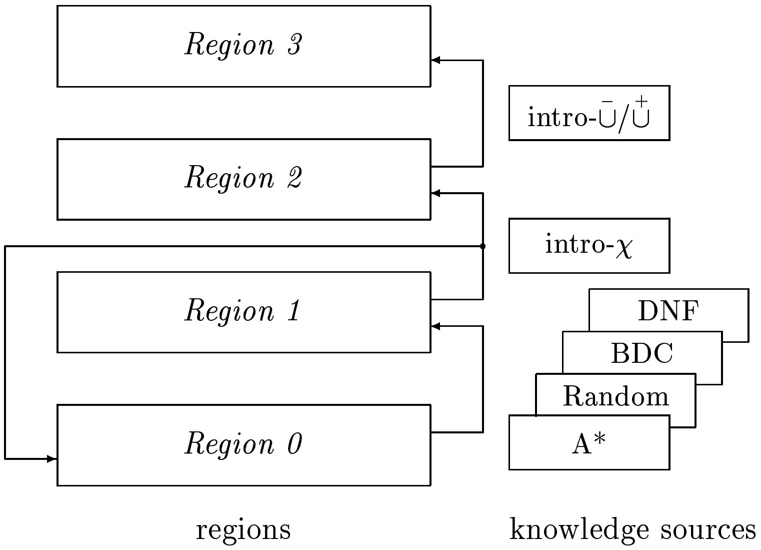


Figure 4: Blackboard architecture

operator.

3.3 Blackboard Architecture with A* Search

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 intermediately 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.

The A^* algorithm will always compute the optimal solution if the estimated future costs constitute lower bounds of the actually arising costs. Since our cost model fulfils this property the optimal query evaluation plan is obtained for each query.

4 Disjunctive Predicate Optimization

In this section, we introduce three algorithms for optimizing queries for bypass processing and one algorithm based on the conventional 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. 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.

Since the A^* implementation is represented by the global search strategy, it is expected not to be as efficient as a dedicated knowledge source customized for integrating selections. However, the A* knowledge source possesses the following advantages:

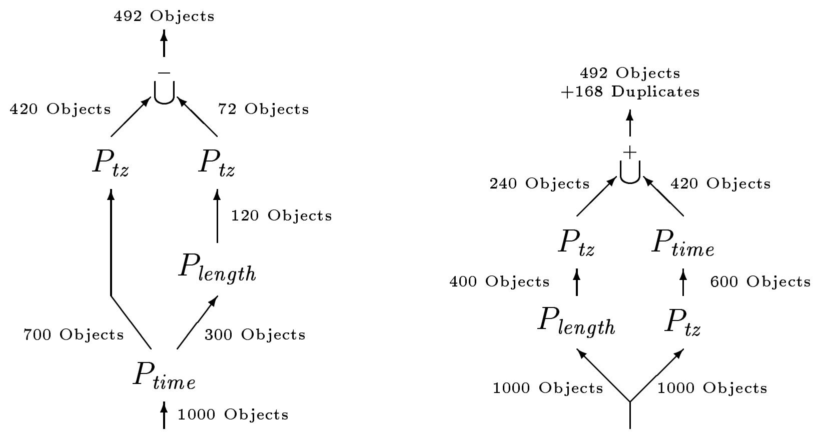
1. The integration of the selections can easily be combined with other heuristics applied by further knowledge sources.

2. A mismatch between the orders of alternatives arranged by the global search strategy and the knowledge source cannot occur since they use the same search strategy.

3. The query evaluation plan assessed as optimal by the cost model will always be found.

Weighing these advantages and the potential performance problem, the global A^* algorithm will be applied on queries which contain a lot of (diverse) characteristics, as e.g., many joins, many selections etc., such that a specialized knowledge source cannot cope with all combinations of operators.

For the running example, the A^* strategy computes the optimal evaluation plan as it was sketched in Figure 2b (Section 2).



(a)
Random search

(b)
"Once and for all" search

Figure 5: Query evaluation plans

4.2 Random Search

The algorithm called *Random* is a very simple knowledge source. 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.

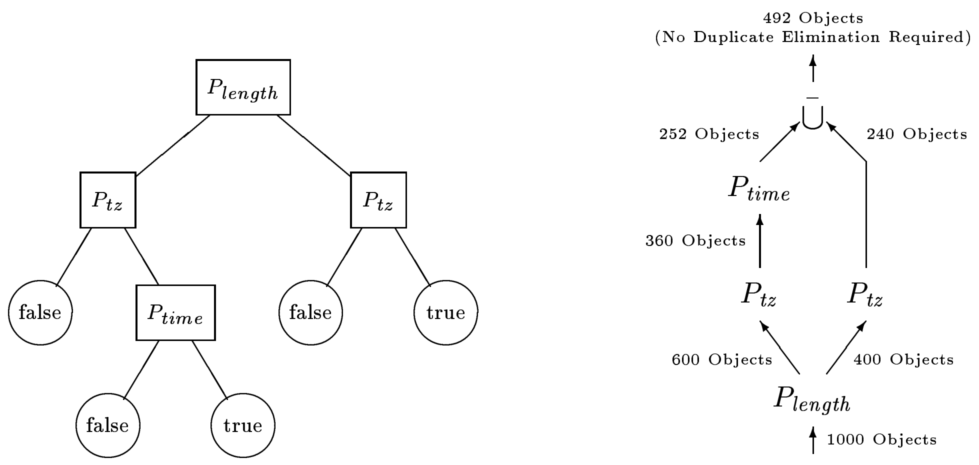
The query evaluation plan generated by *Random* is sketched in Figure 5a. The condition P_{time} is integrated first since it depends on the least number of expansions and, therefore, it can be added to the current expressions in an early iteration. The condition P_{tz} requires the most expansions such that it is integrated last. The generated plan induces an average cost of 115.66 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:

$$\Delta_{x_i} f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \stackrel{\text{def}}{=}$$



(a)
Decision tree

(b)
Evaluation plan

Figure 6: BDC-based evaluation plan for the query "Tiring Flights"

$$f(x_1, \ldots, x_{i-1}, x_i = 0, x_{i+1}, \ldots, x_n) \not\equiv f(x_1, \ldots, x_{i-1}, x_i = 1, x_{i+1}, \ldots, x_n)$$

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 a so-called decision tree for the Boolean function f, where the above mentioned weight of a condition is the quotient of its Boolean difference probability and its evaluation cost. This strategy has been described in [KMS92], where also the complete algorithm for the computation of the decision tree will be found.

With the data given for the example in Section 2, the decision tree in Figure 6a can be computed using the Boolean difference calculus heuristic as described above. If a condition turns out to be *false*, evaluation continues in the left subtree, otherwise in the right subtree; the leaf nodes denote the result of the evaluation. Note that the algorithm favors the inexpensive condition P_{length} ($c = 3$) that is to be evaluated first. On the other hand, the very expensive condition P_{time} is the last one to be evaluated, and in some cases it is not even necessary to do so. The derivation of the evaluation plan for the query itself (cf. Page 4) is straightforward:

- A processing node (=inner node) distributes the objects it receives at its input into two output streams (*false* and *true*), depending on the outcome of the node condition's test.

- Objects arriving at leaf nodes *false* are discarded, as they are not elements of the result set.

- All object streams arriving at leaf nodes *true* are united in an additional merge ($\bar{\cup}$) processing node. As all those streams are necessarily disjunct, there is no need for elimination of duplicate objects. The output of this "union" node is the query's result.

Turning the decision tree in Figure 6a into the query evaluation plan for the query "Tiring Flights" according to the rules stated above yields the evaluation plan in Figure 6b. However, in this case, the heuristic is being misled by the "cheap" condition P_{length}, as the generated plan (avg. cost/object=57) is slightly worse than the optimum (avg. cost/object=55.8, cf. Figure 2b).

4.4 Generation of Conventional Plan

For the purpose of comparison, we have also implemented a knowledge source—denoted DNF in Figure 4—which optimizes the queries according to the conventional selection processing technique. This knowledge source presupposes that the selection predicate is transformed into disjunctive normal form, first. Then, each *and*-connected (sub-)predicate of the normal form can be evaluated by the conventional type of selection processing. In order to obtain a good order of these selections, we have considered a "once and for all" and a "step by step" strategy:

Once and for all: The conditions of each subpredicate are ordered by their ranks which is $\textit{cost per tuple}/(1 - \textit{selectivity})$. Thereafter, a global bottom-up factorization reuniting common subexpressions is performed.

Step by step: The knowledge source integrates at most one selection per invocation into each subbranch, and tries a factorization forthwith. Since, in general, there are several selections that may be integrated, alternatives are generated which are assessed by the global search strategy.

The "step by step" algorithm delivers better optimization results since the factorization is better incorporated. For the running example, the "once and for all" approach generates a query evaluation plan as it is sketched in Figure 5b with an average cost per object of 88.2. Because of $rank(P_{length}) < rank(P_{tz}) < rank(P_{time})$ the common subexpression P_{tz} is ordered in such a way that the factorization fails. The plan generated by the "step by step" strategy is sketched in Figure 2a; it induces an average cost per object of 79.8. For this plan, the global strategy was able to recognize that a factorization results in lower costs than an "optimal" order of the selections. In the quantitative assessment we used the "step by step" approach since it delivers better quality.

5 Assessment

5.1 Generated Benchmark Queries

The benchmarks described in this section were carried out with "synthetical" queries from a query generator. This query generator produces queries that are based on a given

Figure	# Conditions	Percentage AND-Operations	# Queries/Plot Point
7a	3–10	66.66% (33.33% OR)	30
7b	5	0–100%	30
8a	3–10	66.66%	30
8b	5	0–100%	30

Table 3: Benchmark parameters

schema (in our case, the (augmented) airline schema as depicted in Appendix B), and on several other parameters. The basic structure of such a query is

range var: type
retrieve var
where $\mathrm{var.attr}_{1,0}.\cdots.\mathrm{attr}_{1,\mathrm{n}_1}\ op_1\ \mathrm{const}_1$ **and/or**
$\mathrm{var.attr}_{2,0}.\cdots.\mathrm{attr}_{2,\mathrm{n}_2}\ op_2\ \mathrm{const}_2$ **and/or**
...
$\mathrm{var.attr}_{\mathrm{m},0}.\cdots.\mathrm{attr}_{\mathrm{m},\mathrm{n}_\mathrm{m}}\ op_\mathrm{m}\ \mathrm{const}_\mathrm{m}$

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 ($op_i \in \{=, \neq, <, >, \ldots\}$). 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, i.e., $n_i \leq n$. Invocations of type-associated functions count as attribute accesses.

The third parameter, the maximum total number of attribute accesses, 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, or, in other words, 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.

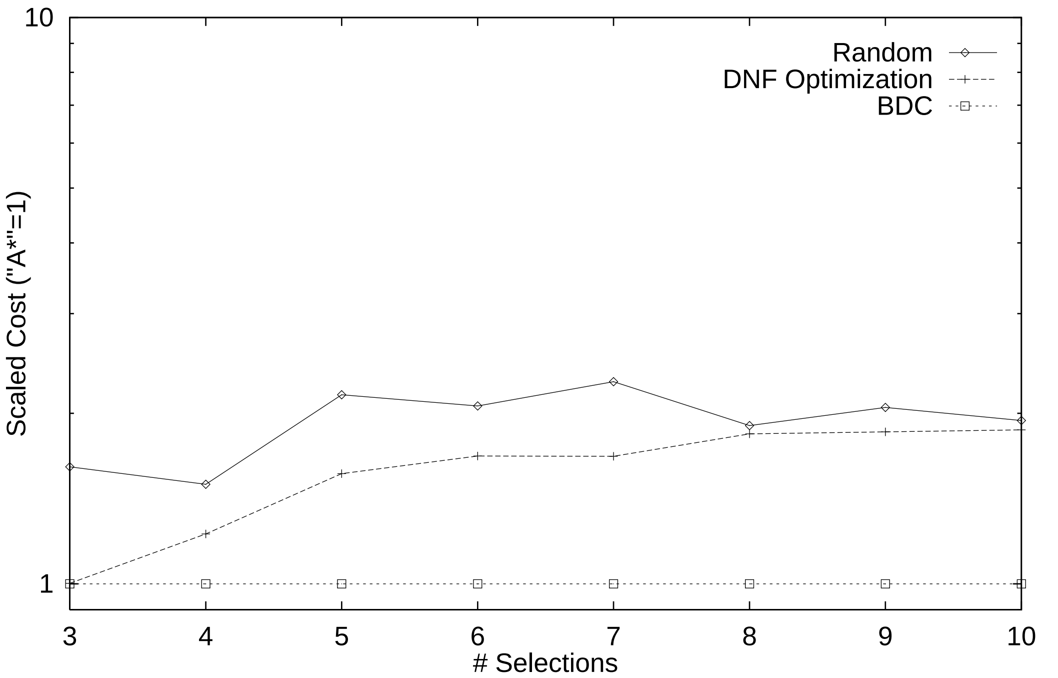
In order to determine these figures, we carried out two series of benchmarks, one for the optimization quality and one for the optimization cost. The parameters for all benchmarks are listed in Table 3. In addition, the plot points are computed as the median of all thirty queries with that particular 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 7a and 7b, the optimization quality of DNF optimization, BDC optimization and the random strategy (as described in Section 4) are compared with the optimal A* algorithm. Figure 7a shows the median of the scaled (with respect to the cost of the A* optimized query) costs for these three knowledge sources. We observe that the BDC optimization performs 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 7b, the number of selection predicate candidates 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 ANDs. 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. 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.

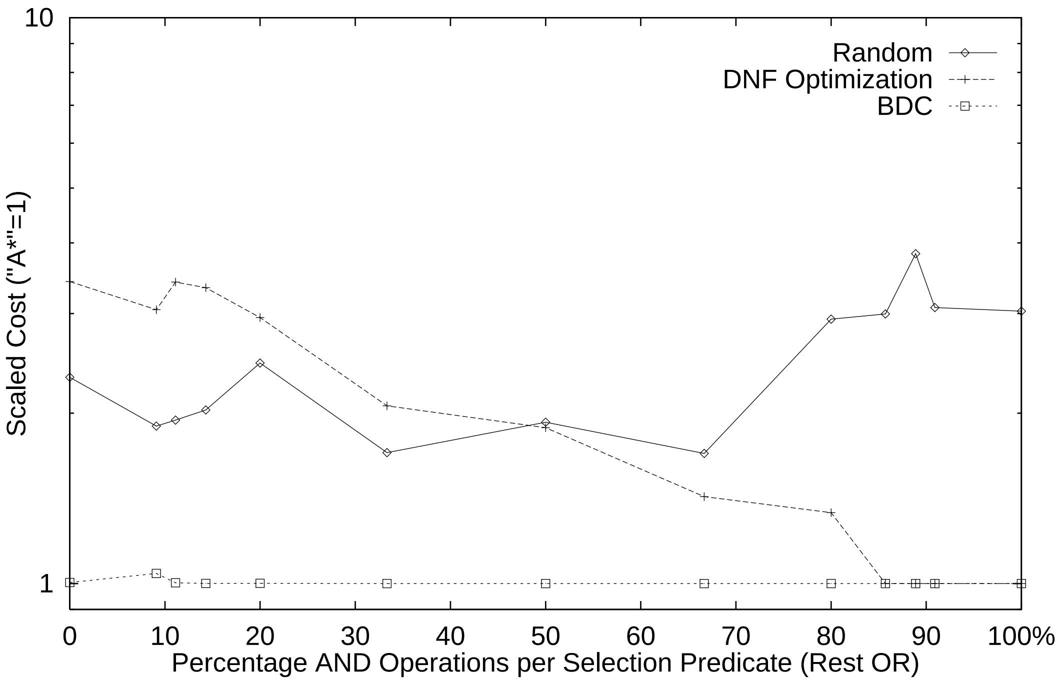
These results indicate clearly that disjunctive queries, especially if many conditions are involved, are not handled very well by the conventional DNF optimization strategy, i.e., the generated evaluation plan based on conventional selection processing is quite bad. A* as well as BDC outperform DNF optimization 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 itselves 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 8a and 8b correspond to Figures 7a and 7b, respectively.

In Figure 8a, the impact of an increasing number 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.



(a)

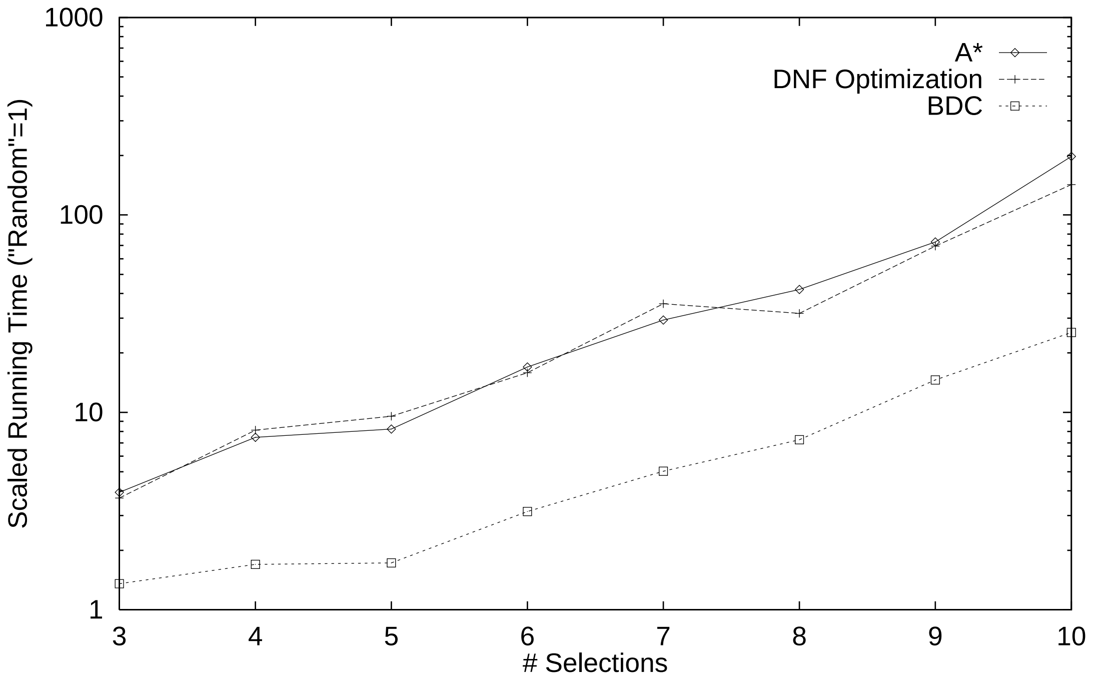
Increasing number of selection predicate conditions (66% AND/33% OR operations)



(b)

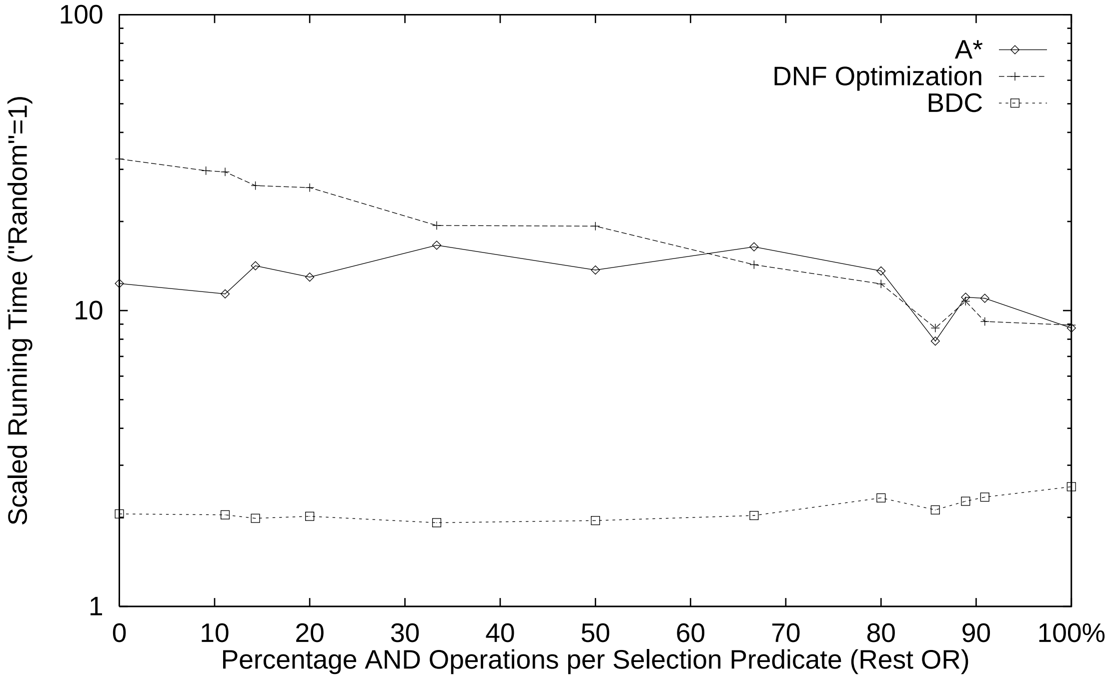
Varying ratio of AND/OR operations (five selection predicate conditions)

Figure 7: 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 selections)

Figure 8: Running time for optimization algorithms

Figure 8b 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.

These results strongly suggest the use of the BDC strategy for the optimization of disjunctive queries. 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 proposed and assessed the *bypass* technique for processing disjunctive selection predicates. Bypass processing is particularly advantageous in object-oriented database systems, because in this context the individual conditions constituting the selection predicate may vary substantially in evaluation cost. The bypass technique tries to derive the outcome of the selection predicate without evaluating such expensive condition whenever possible. To assess the viability of bypass processing, we incorporated 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 formerly developed for finding near-optimal decision trees for Boolean predicates [KMS92].
  3. Random: A random construction of bypass QEPs.
- One algorithm starting with the DNF of the selection predicate and producing a conventional QEP.

On a set of automatically generated queries based upon a "real" schema we quantified the performance gains. It turned out that bypass processing is far superior to the conventional processing—as long as the selection predicate contains a significant number of disjunctions. Among the three optimization algorithms the one denoted A* derives optimal bypass QEPs for an arbitrary query. Nevertheless, BDC could be identified as the "winner in almost all classes" since it produces near-optimal QEPs 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].

Acknowledgement We thank B. Hartmann for implementing the query generator and for carrying out the benchmarks.

References

[Bat86] D. S. Batory. Extensible cost models and query optimization in GENESIS. *IEEE Database Engineering*, 9(4), Dec 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, Jun 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, 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, Jun 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, Apr 1993.

[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, USA, May 1993.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, Jun 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, Aug 1990.

[KM93] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting the relational techniques. In J.-C. Freytag, D. Maier, and G. Vossen, editors, *Query Processing for Advanced Applications*, pages 63–94. Morgan-Kaufmann, 1993.

[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, Aug 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, Aug 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, Sep 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, Aug 1993.

[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, Oct 90.

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

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

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

A Example Type Definitions

```
type Flight is
  public
    legs, total_transit_time;

  body [
    legs: ⟨Connection⟩; /* list-structured attribute */
  ]

  operations
      /* Time the passenger spends waiting between successive flights */
    declare total_transit_time → integer;

  implementation
    define total_transit_time is
    var
      inter: Time;
      sum: integer;
      connection: Connection;
    begin
```

```
        inter := self.legs[0].departure;
        sum := 0;

        /* Add all times between arrivals
         * and departures of successive connections
         */
        foreach (connection in self.legs)
        begin
            sum := sum + connection.departure − inter;
            inter := connection.arrival;
        end;
        return (sum);
    end;
end type Flight;


type Connection is
  public
    departure, arrival, from, to;

  body [
    departure, arrival: Time;
    from, to: Airport;
  ]

end type Connection;


type Airport is
  public
    location, name, timezone;

  body [
    location, name: string;
    timezone: integer;  /* 0 is Greenwich Mean Time, +1 is MET etc. */
  ]

end type Airport;
```

B Schema for the Generated Queries