Optimization and Evaluation of Disjunctive Queries

Jens Claussen, Alfons Kemper, *Member*, *IEEE*, Guido Moerkotte, *Member*, *IEEE*, Klaus Peithner, and Michael Steinbrunn

Abstract—It is striking that the optimization of disjunctive queries—i.e., those which contain at least one **or**-connective in the query predicate—has been vastly neglected in the literature, as well as in commercial systems. In this paper, we propose a novel technique, called *bypass processing*, for evaluating such disjunctive queries. The bypass processing technique is based on new selection and join operators that produce two output streams: the *true*-stream with tuples satisfying the selection (join) predicate and the *false*-stream with tuples not satisfying the corresponding predicate. Splitting the tuple streams in this way enables us to "bypass" costly predicates whenever the "fate" of the corresponding tuple (stream) can be determined without evaluating this predicate. In the paper, we show how to systematically generate bypass evaluation plans utilizing a bottom-up building block approach. We show that our evaluation technique allows to incorporate the standard SQL semantics of null values. For this, we devise two different approaches: One is based on explicitly incorporating three-valued logic into the evaluation plans; the other one relies on two-valued logic by "moving" all negations to atomic conditions of the selection predicate. We describe how to extend an iterator-based query engine to support bypass evaluation with little extra overhead. This query engine was used to quantitatively evaluate the bypass evaluation plans against the traditional evaluation techniques utilizing a CNF- or DNF-based query predicate.

Index Terms—Query optimization, query processing, disjunctive queries, query evaluation plans, expensive query predicates, bypass processing.

1 Introduction

Since the early stages of relational database development, query optimization has received a lot of attention. Consequently, this attention has recently shifted to so-called "next-generation" database systems [3]. References [4], [5], [6] made rule-based query optimization popular, which was later adopted in the object-oriented context as, e.g., [7], [8], [9]. Many researchers have worked on optimizer architectures that facilitate flexibility: [5], [10], [11], [12] are proposals for optimizer generators; [13], [14] describe extensible optimizers in the extended relational context; [15], [16] propose architectural frameworks for query optimization in object bases.

Besides these works on optimizer architectures, optimization strategies for both traditional and "next-generation" database systems are being developed. Levy et al. [17] introduce a technique for moving predicates across query components, where a component constitutes, for instance, a view definition. Hellerstein and Stonebraker [18] optimize the placement of predicates within the query graph. The authors 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 and, in extended relational and object-oriented systems, additionally in the form of user-defined functions. Hellerstein and Stonebraker's [18] work is based on ordering the conditions in a sequence according to their relative selectivity and evaluation cost—adapting a technique developed in operations research [19]. Their [18] work was recently extended by [20].

It is striking that, in all these works, the optimization of disjunctive query predicates tends to be neglected. References [21] and [22] are the only works—to the authors' knowledge—that dealt with disjunctions in particular. We will comment on their work later in this section.

The traditional approaches transform a query predicate (consisting of selection and/or join predicates) into a normal form (namely, conjunctive or disjunctive normal form), thus reducing the problem to the common, purely conjunctive case: Either disjunctions are considered atomic within a single conjunction (conjunctive normal form, for instance in System R [23]) or the predicate is subdivided into several conjunctive streams that are optimized separately (disjunctive normal form, e.g., [7], [24], [25], [26]).

In this paper, we show that either approach fails to exploit a vast optimization potential because a sufficiently fine-tuned adaptation to a particular query's characteristics cannot be achieved that way. The *bypass technique* fills the gap between the achievements of traditional query optimization and the theoretical potential. In this technique, specialized operators are employed that yield the tuples

• *J. Claussen and A. Kemper are with the Universität Passau, D-94030 Passau, Germany. E-mail: {claussen, kemper}@db.fmi.uni-passau.de.*
• *G. Moerkotte is with the Universität Mannheim, Lehrstuhl für Praktische Informatik III, D7,27 D-68131 Mannheim, Germany. E-mail: moer@pi3.informatik.uni-mannheim.de.*
• *K. Peithner is with MicroStrategy Inc. Hanauer Str. 40, 68305 Mannheim, Germany. E-mail: kpeithner@strategy.com.*
• *M. Steinbrunn is with Sun Microsystems GmbH, Bretonischer Ring 3, D-85630 Grasbrunn, Germany. E-mail: Michael.Steinbrunn@germany.sun.com.*

Manuscript received 6 Sept. 1996; revised 25 June 1997; accepted 29 Jan. 1998.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104363.

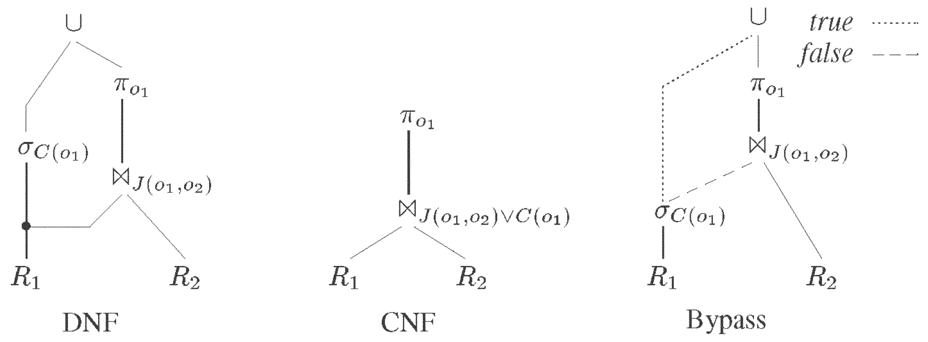


Fig. 1. Evaluation plans for $\pi_{o_1}(\sigma_{C(o_1)\vee J(o_1,o_2)}(R_1 \times R_2))$.

that fulfill the operator's predicate *and* the tuples that do not on two different, disjoint output streams: the *true*-stream and the *false*-stream. This gives the opportunity of performing an individual, "customized" optimization for both streams. Particularly costly and/or less selective predicates may thereby be *bypassed* by certain tuple streams because the fate of those tuples may be determined without evaluating the particular predicate. As an example, consider the following (normalized) relational algebra expression:

$$\pi_{o_1}(\sigma_{C(o_1)\vee J(o_1,o_2)}(R_1 \times R_2)).$$

A selection operation with the predicate $C(o_1) \vee J(o_1, o_2)$ (where $o_1 \in R_1$ and $o_2 \in R_2$) is performed on the Cartesian product $R_1 \times R_2$, followed by a projection on R_1's attributes. The DNF approach transforms the selection predicate's disjunction into a union operation, yielding

$$\sigma_{C(o_1)}(R_1) \cup \pi_{o_1}(R_1 \bowtie_{J(o_1,o_2)} R_2).$$

The CNF approach, on the other hand, has to join the two extensions with the Boolean factor $C(o_1) \vee J(o_1, o_2)$ as join condition:

$$\pi_{o_1}(R_1 \bowtie_{C(o_1)\vee J(o_1,o_2)} R_2).$$

It is not possible in either conventional approach to perform the restriction $C(o_1)$ first in order to minimize the input cardinalities of the join. The bypass technique, however, computes the result of this query as the union

$$\sigma_{C(o_1)}(R_1) \cup \pi_{o_1}\left(\sigma_{\neg C(o_1)}(R_1) \bowtie_{J(o_1,o_2)} R_2\right),$$

where the two selections $\sigma_{C(o_1)}(R_1)$ and $\sigma_{\neg C(o_1)}(R_1)$ are implemented as a single *bypass selection*, so we may expect cost reductions in comparison with both the DNF- and CNF-based evaluation plan. Fig. 1 shows the three evaluation plans. The bypass plan contains the bypass selection with two output streams, the *true*-stream for tuples matching $C(o_1)$ and *bypassing* the expensive join, and the *false*-stream for tuples that do not satisfy $C(o_1)$. (Of course, all three plans can be further optimized by transforming the join and the subsequent projection into a semijoin.)

This new class of evaluation plans requires the development of adequate construction algorithms. We present an algorithm generating the optimal bypass plan and another one producing near optimal plans exploring the search space only partially.

We also show that our evaluation technique allows us to incorporate the standard SQL semantics of null values. For this we devise two different approaches: One is based on explicitly incorporating three-valued logic into the evaluation plans. Thereby, sometimes three output streams (the *true*-, the *false*-, and the *unknown*-streams) are generated for a selection or join-node. However, very often, two of the three streams can be combined since they lead to the same final outcome of the entire selection predicate. The other technique remains within two-valued logic by "moving" all negations to atomic conditions of the selection predicate. For this technique, we need two different "polarizations" of the condition evaluation: The positive polarization maps an *unknown* outcome to *true* and the negative polarization maps it to *false*.

In this paper, we do not content ourselves with pointing out the theoretical merits of the bypass technique, but show its superiority by means of an actual implementation and experimental assessment. Since the efficient implementation of bypass operators, on one hand, poses several questions and is, on the other hand, crucial for competitive performance, we present a number of proven strategies. In principle, this puts the reader into the position of incorporating the bypass technique into existing optimizers and execution engines and emphasizes the technique's practical feasibility.

Let us compare our bypass processing with the two related works of [21] and [22], which offer special approaches to deal with disjunctions. Muralikrishna [21] tries to reduce the number of file scans and join operations for disjunctive predicates. Starting from disjunctive normal form, *merge graphs* are employed to combine multiple disjunctive selection conditions on the same relation and evaluate them in a single operation. For example,

$$\sigma_{(C_1(o_1)\wedge J(o_1,o_2))\vee(C_2(o_1)\wedge J(o_1,o_2))}(R_1 \times R_2)$$

is simplified to use a single join operation:

$$\sigma_{(C_1(o_1)\vee C_2(o_1))\wedge J(o_1,o_2))}(R_1 \times R_2)$$

Muralikrishna's [21] major concern is to identify common subexpressions in order to prevent repeated evaluation of the same predicate on the same tuples (objects). Our bypass evaluation technique guarantees that any predicate is evaluated at most once for any given object.

Bry [22] converts predicates with disjunctions and quantifiers into the so-called *Miniscope* form, which is

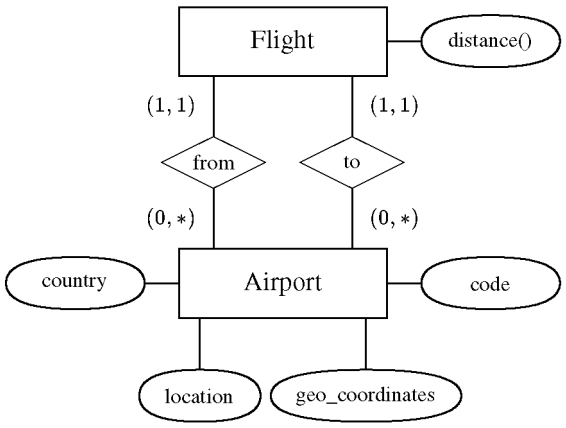


Fig. 2. ER-schema of the flight reservation system.

achieved by pushing quantifiers as far inside as possible (as opposed to the *prenex normal form* [26], where all quantifiers are moved outside). Disjunctions in predicates are then evaluated using a newly introduced *constrained outer-join* operator. The operator allows the evaluation of disjunctive predicates over multiple relations without the necessity of a setunion operator. Already qualified tuples are marked by a special attribute such that redundant computation of predicates is avoided. This solution keeps a single output stream (as does [21]'s) and is thus probably easier to implement than our bypassing, but it is not as powerful as our approach. In particular, both [21] and [22] only avoid the evaluation of predicates, but they do not bypass expensive operations in general. Furthermore, separate (i.e., individual) optimization of the two output streams is not possible.

The rest of the paper is organized as follows: Section 2 illustrates the bypass technique by means of a sample query and points out the differences to conventional evaluation plans. Section 3 introduces bypass operators in a more detailed way. This is used for the construction of both conventional evaluation plans and bypass plans. The construction strategies for all plans are described in Section 4. In Section 5, we extend the evaluation technique to deal with null values. In Section 6, efficient implementation techniques for bypass operators are discussed and Section 7 quantitatively compares sample bypass evaluation plans with conventional evaluation plans on an experimental basis. Section 8 concludes the paper. In the appendices, we give a description of the optimization algorithm in pseudocode and we provide a more detailed description of our benchmarking environment.

2 Bypass Technique

2.1 Running Example

In order to illustrate the optimization potential that is made available by the bypass technique, let us consider a sample query. It is based on a (partial) schema of a flight reservation system, the ER-schema of which is shown in Fig. 2. The query itself retrieves all airports that may act as "immigration airports" into the USA. Those airports may be characterized as:

- US airports; or
- non-US airports, from which a direct flight's distance to a US airport does not exceed 400 miles.

The OQL [29] rendering of this query is

select **distinct** a
from Airport **as** a, Flight **as** f
where a.country = "USA" **or**
(a = f.from **and** f.to.country = "USA" **and**
f.distance() $\leq$ 400)

A sample object base according to the schema in Fig. 2 is depicted in Fig. 3. It contains two flights, namely from Toronto to New York City and from Mexico City to New York City. Although the attribute *distance* is implemented as a type-associated function and will be computed "on demand," the result is listed in Fig. 3 for the purpose of clarity. Thus, the query stated above would yield id_3 (inside the USA) and id_4 (starting point of a flight into the USA of less than 400 miles) as "immigration airports." For convenient handling, we abbreviate the query's atomic conditions and assume probabilities as listed in Table 1. The (relative) cost figures given in the following subsections for the different evaluation plan alternatives denote total elapsed processing time as obtained from experiments in Section 7 and assume a database containing 1,000 objects of type *Airport* and 29,000 objects of type *Flight*.

2.2 Drawbacks of Common Techniques

The prevailing techniques use the conjunctive normal form (CNF) or the disjunctive normal form (DNF) as construction

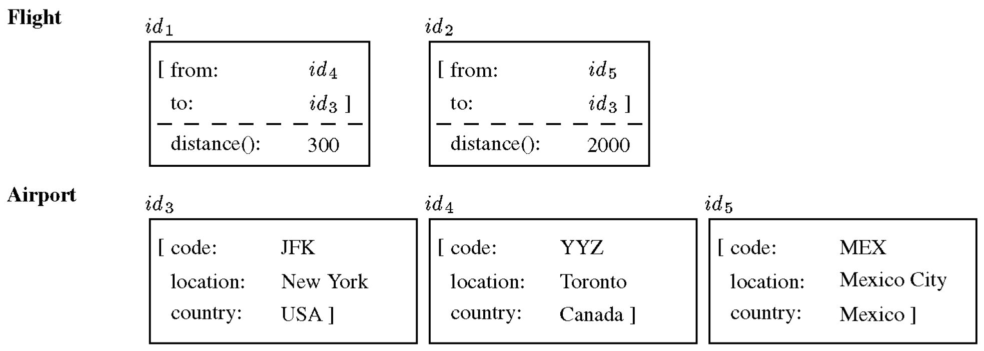


Fig. 3. Sample object base.

TABLE 1
Abbreviations and Probabilities for "Immigration Airports"

Condition	Abbreviation	Probability p
$a.country =$ "USA"	$C_{USA}(a)$	0.33
$f.to.country =$ "USA"	$C_{to}(f)$	0.40
$f.distance() \leq 400$	$C_{distance}(f)$	0.10
$a = f.from$	$J_{from}(a, f)$	0.001

base for query evaluation plans. However, both these approaches suffer from certain drawbacks, particularly when dealing with disjunctive queries. In Fig. 4, the optimal evaluation plan that can be derived from the conjunctive normal form is depicted. The conjunctive normal form of the query predicate is

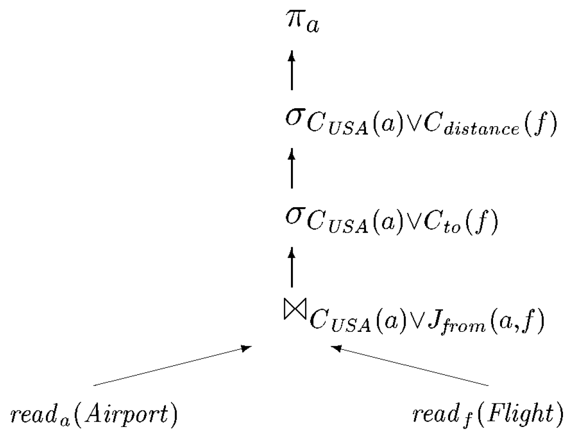
$$\big(C_{USA}(a) \vee J_{from}(a, f)\big) \wedge (C_{USA}(a) \vee C_{to}(f)) \\ \wedge (C_{USA}(a) \vee C_{distance}(f)).$$

The three selection and join processing nodes correspond to the so-called *Boolean factors* of the conjunctive normal form. In the first processing node, the join operation $\bowtie_{C_{USA} \vee J_{from}}$ is performed. Objects from the Cartesian product *Airport* $\times$ *Flight* that satisfy the condition C_{USA} immediately move on to the second stage; those that do not must be tested by the second condition, the join condition J_{from}.

The same procedure is repeated for the second and third stage, respectively: If an object satisfies the first condition of the Boolean factor, it is moved on; otherwise the second condition determines its fate. The final operation of this evaluation plan performs the projection on the desired attributes and yields the resulting set.

This evaluation plan is the optimal CNF-based plan under the assumption that a result cache for the condition C_{USA} is available, i.e., the test for any given object must be carried out at most once. This evaluation plan suffers from a number of drawbacks. First, partitioning the query's conditions into Boolean factors leads to an overly expensive first operation node, where the entire Cartesian product of *Airport* and *Flight* must be considered in order to carry out the join. It would be much better if the selection C_{USA} would be carried out in advance in order to reduce the number of *Airport* objects that participate in the join operation. Second, one of the four conditions, namely C_{USA}, appears in every operation node's predicate, making a result cache—as proposed by [30]—indispensable for a more expensive predicate if competitive performance is desired. A predicate may become expensive, e.g., because of the evaluation of a complex user-defined function involving possibly disk i/o. And third, we cannot do without the (duplicates eliminating) projection π_a because the query demanded duplicate elimination ("select distinct"). These drawbacks are also reflected in the high observed cost for this evaluation plan: It is 150 times higher than the observed cost of the optimal bypass plan.

The second "classical" alternative uses the disjunctive normal form (DNF) as the foundation for constructing evaluation plans. The DNF of the example query's predicate is



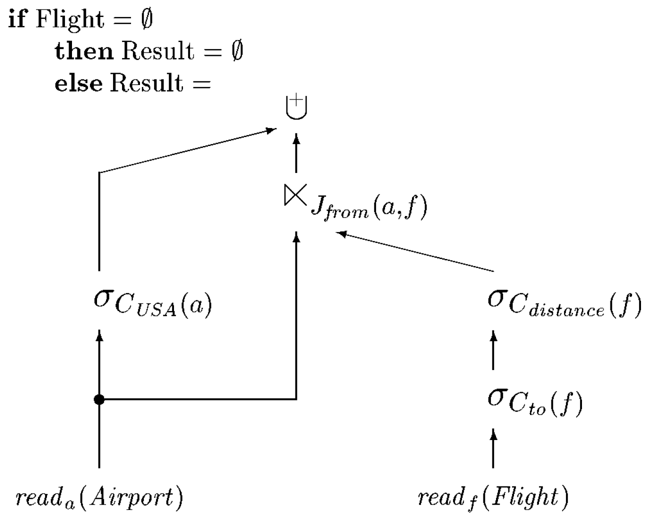
π_a
$\sigma_{C_{USA}(a) \vee C_{distance}(f)}$
$\sigma_{C_{USA}(a) \vee C_{to}(f)}$
$\bowtie_{C_{USA}(a) \vee J_{from}(a,f)}$
$read_a(Airport)$
$read_f(Flight)$

Fig. 4. CNF-based evaluation plan for "Immigration Airports." (Relative Cost: Factor 150).

$$C_{USA}(a) \vee \big(C_{to}(f) \wedge C_{distance}(f) \wedge J_{from}(a, f)\big).$$

In contrast to CNF-based plans, which always consist of a single object stream, DNF-based plans maintain as many object streams as the disjunctive normal form contains minterms. The individual results of these object streams are combined in a final union operation that represents the DNF's *or*-operations. Fig. 5 shows the implementation of this strategy for the sample query "Immigration Airports." In order to ascertain the compliance of the evaluation plan to the query's semantics as specified by the OQL standard, the test of whether the object extension *Flight* is empty must be carried out in advance. If it is, the result is definitely empty; if it is not, the actual evaluation plan is invoked. The stream on the lefthand side of Fig. 5 represents the DNF's minterm with the single condition C_{USA}, which is turned into the selection operation $\sigma_{C_{USA}}$. Objects from *Airport* that satisfy C_{USA} are certain to be elements of the result set and move on immediately to the above-mentioned final union operation node.

The second object stream starts on the righthand side with *Flight* objects, which is filtered by the two selection



if Flight $= \emptyset$
then Result $= \emptyset$
else Result $=$
$\uplus$
$\ltimes_{J_{from}(a,f)}$
$\sigma_{C_{USA}(a)}$
$\sigma_{C_{distance}(f)}$
$\sigma_{C_{to}(f)}$
$read_a(Airport)$
$read_f(Flight)$

Fig. 5. DNF-based evaluation plan for "Immigration Airports" (relative cost: 240 percent).

operations $\sigma_{C_{to}}$ and $\sigma_{C_{distance}}$ before serving as one input for the semijoin $\ltimes_{J_{from}}$. The semijoin's second input is the set of all *Airport* objects—this set is a "clone" of the input set for C_{USA}. The resulting objects satisfy the three conditions C_{to}, $C_{distance}$ and J_{from}, i.e., the DNF's second minterm $C_{to}(f) \wedge C_{distance}(f) \wedge J_{from}(a, f)$ and are therefore elements of the result set, too.

The union operation $\uplus$ that unites the two object streams must remove duplicate elements that are caused by objects that pass both streams, i.e., objects that satisfy both minterms. In general, DNF-based plans for queries where tuples may satisfy more than one minterm do not work without duplicate removal and thus cannot preserve duplicates according to the SQL semantics. Therefore, the DNF-based evaluation technique—as presented here—can only be used if duplicate preservation is not required. If duplicate preservation were required one would have to associate additional identifiers (e.g., TIDs) with the tuples in order to distinguish "wanted" duplicates from "unwanted" ones. However, we will not further elaborate on this issue since our bypass processing technique correctly retains duplicates without any additional control mechanisms.

Although this DNF-based evaluation plan does not yet constitute the optimal strategy, the observed cost is already much lower than for the CNF-based plan, namely "only" 240 percent of the optimal bypass plan's observed cost. There are mainly two places where the lever for further cost reductions can be positioned: First, one can observe that *all* objects, even those that satisfy C_{USA}, are subject to testing by condition J_{from}. Restricting this test to objects that do not satisfy C_{USA} would eliminate that redundancy. This redundancy also forces the duplicate eliminating union operation—here denoted $\uplus$—as a final step, an operation that also is quite costly. Second, depending on the query predicate, it may turn out that a certain condition appears in more than one minterm—the corresponding evaluation plan then has to evaluate that condition more than once per object. This consideration is particularly important when the evaluation costs for the conditions in question are high.

If we compare the two classic approaches, CNF- and DNF-based evaluation plans, we conclude that neither is capable of yielding the minimum cost plan. Both strategies suffer from various, albeit different, weaknesses. In the following sections of this paper, we present a technique, the so-called *bypass technique*, that provides remedies for these weaknesses.

2.3 Benefits of the Bypass Technique

In Section 2.2, we identified two main causes for the poor handling of disjunctive queries by traditional evaluation techniques. Evaluation plans based on the conjunctive normal form suffer from the (rather coarse) partitioning into Boolean factors. That is—apart from redundant computations of conditions—also the main reason for the suboptimal performance of plans that are based on the disjunctive normal form. Our new evaluation technique, the *bypass technique*, addresses both points: It allows fine-grained allocation of the query predicate's conditions to operation nodes (i.e., on the level of individual atomic conditions, not merely of Boolean factors), and it avoids redundant computations entirely. Both achievements are

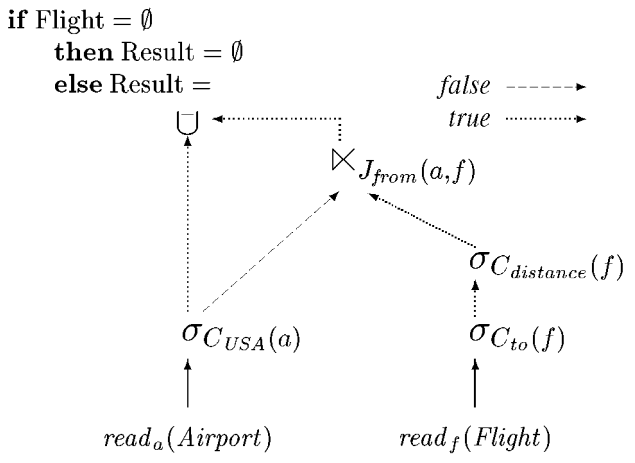


Fig. 6. Bypass evaluation plan for "Immigration Airports" (relative cost: 100 percent (Optimum)).

made possible by introducing a new class of operators, so-called *bypass operators*, namely *bypass selection* and *bypass joins*. Those operators are characterized by *two* result sets instead of a single one: They do not simply determine those input objects that satisfy the operation's predicate, but distinguish the input set into two disjoint output sets, consisting of objects that satisfy the predicate and those that do not. An example of the application of bypass operators is the bypass plan for our sample query "Immigration Airports" (Fig. 6). This evaluation plan does not merely constitute the lowest-cost bypass plan for the sample query, but the optimal evaluation plan with respect to the presented construction methods.

Let us now study how the evaluation plan works in detail. Again, the test for an empty *Flight* extension must be carried out in advance in order to guarantee the correct semantics. On the lefthand side of Fig. 6, *Airport* objects are tested as to whether they satisfy the condition C_{USA}, similar to the DNF-based evaluation plan (cf. Fig. 5). However, whereas, in the DNF-based evaluation plan *all* objects of type *Airport* must be submitted to the test J_{from}, the bypass plan tests only those objects that are not already certain to be elements of the result (i.e., those that do not satisfy C_{USA}). This task is performed by the bypass selection operator $\sigma_{C_{USA}}$ that separates its input into two disjoint output streams. In Fig. 6, these two different streams are marked with dotted lines (objects that satisfy the condition) and dashed lines (objects that do not satisfy the condition), respectively.

On the righthand side of Fig. 6, objects of type *Flight* must satisfy the two conditions C_{to} and $C_{distance}$, before the semijoin J_{from} is carried out (similar to Fig. 5). The last operation in the bypass evaluation plan is the union node $\cup$, where the two streams are united. In contrast to the DNF-based evaluation plan, this union operation is guaranteed to unite disjoint streams, so an elimination of duplicates is not required, with lower processing cost as a consequence. In order to emphasize this difference, the common set union (with elimination of duplicates) is denoted as $\uplus$, and the union operation that can rely on the disjointness of its operands and simply merges them is denoted as $\cup$.

Considering the differences between the CNF- and DNF-based evaluation plans and the bypass evaluation plan, we observe that the bypass evaluation plan does not only bypass operation nodes (such as the semijoin node in Fig. 6), but also bypasses the disadvantages of the two traditional techniques. The bypass technique allows fine-grained allocation of selection or join predicates to an extent not available in CNF-based plans. Additionally, it avoids redundant invocations of operation predicates—for any given object, a certain test is carried out at most once. These facts are reflected in the evaluation cost for our sample query's bypass evaluation plan, which is the lowest of the three alternatives. As a matter of fact, it is the optimal evaluation plan for our sample query. We recall that the best CNF-based plan (Fig. 4) is 150 times as expensive, and the best DNF-based plan (Fig. 5) is still 2.4 times more expensive to evaluate than the bypass plan. Apart from higher cost, the DNF plan has the disadvantage that it is only applicable if duplicate elimination is desired. The cost relationship between the CNF- and DNF-based plan is due to our example query. Other queries, in particular joinless ones (cf. [1]), may have the effect that the CNF-based plan is superior to the DNF-based plan. However, in the vast majority of cases, the optimal plan for evaluating a disjunctive query is a bypass plan. So, the fact that the bypass plan is the one with lowest cost is *not* a peculiarity of the chosen example.

Several more questions remain to be answered in this paper:

- What bypass operators are available for the construction of evaluation plans?
- What techniques should be used for constructing bypass plans?
- How can bypass operators be implemented efficiently?
- What is the actual performance of bypass plans compared to traditional plans?

Each of these questions will, in turn, be addressed in the subsequent sections.

3 Preliminaries

This section first describes the logical (object-oriented) algebra and, second, outlines the algebraic extensions for exploiting bypassing.

3.1 Basic (Logical) Algebra

For the purposes of this paper, a "slim" object-oriented algebra is sufficient. It consists of the well-known relational operators selection "σ," projection "π," Cartesian product "$\times$," join "$\bowtie$," semijoin "$\ltimes$," and "$\rtimes$," division "$\div$," set difference "$-$," and union "$\cup$." Furthermore, the algebra contains a so-called *expand operator* "χ" [8]—comparable with [11]'s materialize operator or the map function [31]—that is used for accessing object attributes, be they stored or computed (i.e., methods). For instance, the two path expressions

$$f.to.country \quad \text{and} \quad f.distance()$$

will be translated into

$$\chi_{to:f.to}, \chi_{cou:to.country} \quad \text{and} \quad \chi_{dis:f.distance()}$$

with system-generated new variables *to*, *cou*, and *dis*. The expand operator incorporates object-oriented concepts into the relational algebra context. It constitutes the major difference between the familiar relational algebra and the object-oriented algebra (within the scope of this paper). It does not play a decisive role with respect to the overall shape of the discussed evaluation plans; this implies that the presented concepts are in no way restricted to the "object-oriented world," but can be applied without major modifications for any "modern" data model, be it relational, object-relational, or object-oriented.

Subsequently, $\mathcal{A}(S)$ denotes the attribute set of a relation S. For a formal definition of the expand operator, let S be a relation with (at least) one attribute $a_i \in \mathcal{A}(S)$, g be an operation (attribute access or function invocation) defined on a_i, and $a \notin \mathcal{A}(S)$ be a further attribute name. Then, the expand operator is defined as follows ("$\circ$" is the tuple concatenation operator):

$$\chi_{a:a_i.g()}(S) := \{s \circ [a : s.a_i.g()] \mid s \in S\}$$

For reading the type extensions into memory, we use the following operation:

$$read_s(S) := \{[s : s'] \mid s' \in S\}$$

A relation with one column s is generated.

3.2 (Logical) Bypass Operators

The operators discussed so far constitute the "lowest common denominator" for all kinds of evaluation plans considered in this paper. However, for bypass evaluation plans, a new class of operators is required, namely so-called *bypass selections* and *bypass joins*, as well as *bypass semijoins*. What makes these operators stand out is the fact that they yield *two* result sets instead of just one. In order to facilitate the algebraic handling of the bypass operators, they are separated into two "halves," namely σ^+/σ^-, $\bowtie^+ / \bowtie^-$, and $\ltimes^+/\ltimes^-$, that provide the two *complementary* result sets —although this separation is *not* reflected in the actual implementation, where the two result streams are provided simultaneously (cf. Section 6).

The definitions for bypass operators with predicate C and operands S and T are as follows:

$$\begin{aligned}
\sigma_C^+(S) &:= \{s \mid s \in S \wedge C(s)\} \\
\sigma_C^-(S) &:= S - \sigma_C^+(S) \overset{*}{=} \{s \mid s \in S \wedge \neg C(s)\} \\
S \bowtie_C^+ T &:= \{s \circ t \mid s \in S \wedge t \in T \wedge C(s,t)\} \\
S \bowtie_C^- T &:= (S \times T) - (S \bowtie_C^+ T) \\
&\overset{*}{=} \{s \circ t \mid s \in S \wedge t \in T \wedge \neg C(s,t)\} \\
S \ltimes_C^+ T &:= \{s \mid s \in S \wedge \exists t \in T : C(s,t)\} \\
S \ltimes_C^- T &:= S - (S \ltimes_C^+ T) \overset{*}{=} \{s \mid s \in S \wedge \neg \exists t \in T : C(s,t)\}.
\end{aligned}$$

Thus, bypass operators always come in pairs, one with a "positive" and the other with a "negative" output, but the two matching parts of a bypass operator are still *one* operation. The input is split into two disjoint parts which can subsequently be reunited by a merge operator *without* duplicate eliminations. We denote the bypass operators by $\sigma^{\pm}$, $\bowtie^{\pm}$, $\rtimes^{\pm}$, and $\ltimes^{\pm}$.

Note that, in the above definitions, we have to carefully handle negations in order to obtain the equalities marked with *. Usually, a condition $C \equiv x\ \theta\ y$ can easily be negated by inverting the comparison operator θ. For instance, "=" becomes "$\neq$." However, this transformation will be counterintuitive, if *null* values are allowed. For example, both conditions $a.country = "\mathrm{USA}"$ and $a.country \neq "\mathrm{USA}"$ yield *false* whenever the attribute *country* of an airport object a is not defined, i.e., is *null*. We discuss null values in detail in Section 5. Until then we will assume that the database contains no null values and, therefore, for all conditions C and all objects (tuples) o, either $C(o)$ or $\neg C(o)$ yields *true*—which is required for the "starred" equations.

4 Construction Strategies

In this section, we shall discuss construction strategies for building CNF-based, DNF-based, and bypass plans.

4.1 Conventional Plans: CNF and DNF

In the CNF-based approach, the normalized Boolean function consists of the conjunction of so-called *Boolean factors* (disjunctive terms of the predicate's conditions):

$$\underbrace{(C_{1,1} \vee \cdots \vee C_{1,k_1})}_{1^{\mathrm{st}}\ \text{Boolean factor}} \wedge \cdots \wedge \underbrace{(C_{m,1} \vee \cdots \vee C_{m,k_m})}_{m^{\mathrm{th}}\ \text{Boolean factor}}.$$

Thus, the optimization can be subdivided into two steps:

1. arranging the Boolean factors; and
2. arranging the conditions within the Boolean factors.

Determining the least-cost ordering can be carried out efficiently at least for selection predicates [32] in the special case when no condition appears in more than one Boolean factor. In general, however, this is not the case. A duplicated condition's probability for being true and—provided a result cache is available—its invocation cost depends on the condition's position within the CNF. In consequence, both orderings cannot be carried out independently as soon as one of the conditions C_i appears in more than one Boolean factor. Hence, for disjunctive queries, an exhaustive search seems necessary for determining the optimal CNF-based query evaluation plan. We shall call this algorithm "CNF."

The DNF-based construction method starts from the predicate's disjunctive normal form, consisting of *Boolean summands* (conjunctive terms of the predicate's conditions):

$$\underbrace{(C_{1,1} \wedge \cdots \wedge C_{1,k_1})}_{1^{\mathrm{st}}\ \text{Boolean summand}} \vee \cdots \vee \underbrace{(C_{m,1} \wedge \cdots \wedge C_{m,k_m})}_{m^{\mathrm{th}}\ \text{Boolean summand}}.$$

Since the $\vee$-operations are converted into one m-way union operation with elimination of duplicates, the optimization "only" comprises the sorting of the conditions within the Boolean summands according to [32]. However, in order to minimize redundant computations, common subexpressions must be identified. Muralikrishna reduced this problem to a graph covering problem, which he proved to be NP-hard [21]. Hence, the search for the optimal DNF-based plan comprises two problems—determining the best join order and determining common subexpressions—which both have been shown to be NP-hard in general.

In the following, we will refrain from identifying common subexpressions, but content ourselves with determining the best ordering of conditions within each Boolean summand by exhaustive search. This algorithm we shall call "DNF."

4.2 Bypass Plans

In this section, we shall outline the procedure for the construction of bypass plans.

4.2.1 *The Construction Algorithm*

The basic idea is to start from the query's canonical representation, namely as a selection σ_g (with $g(C_1, \ldots, C_m)$ the selection predicate) on the Cartesian product $R_1 \times \ldots \times R_n$ of the involved relations (object extensions):

$$Q = \sigma_{g(C_1,\ldots,C_m)}(R_1 \times \ldots \times R_n).$$

Such a canonical representation can be decomposed into selection and join operations with atomic conditions as their predicates using the following two equations:

$$\begin{aligned} Q = {} & \sigma_{g|C_i=false}(R_1 \times \cdots \times \sigma^-_{C_i}(R_j) \times \cdots \times R_n) \,\dot\cup \\ & \sigma_{g|C_i=true}(R_1 \times \cdots \times \sigma^+_{C_i}(R_j) \times \cdots \times R_n) \end{aligned} \qquad (1)$$

$$\begin{aligned} Q = {} & \sigma_{g|C_i=false}(R_1 \times \cdots \times (R_j \bowtie^-_{C_i} R_k) \times \cdots \times R_n) \,\dot\cup \\ & \sigma_{g|C_i=true}(R_1 \times \cdots \times (R_j \bowtie^+_{C_i} R_k) \times \cdots \times R_n). \end{aligned} \qquad (2)$$

At each step of the construction process, a condition C_i is selected as a "splitting point," depending on the nature of C_i: If it is a selection condition, i.e., only a single relation R_j is involved, (1) is employed; otherwise, if C_i is a join condition on relations R_j and R_k, (2) is applied. For convenience reasons, we shall use the notation $\{R_1, \ldots, R_n\}_{g(C_1,\ldots,C_m)}$ for a canonical query $\sigma_{g(C_1,\ldots,C_m)}(R_1 \times \ldots \times R_n)$. The expression $\{R_1, \ldots, R_n\}_{g(C_1,\ldots,C_m)}$ is called a *bundle* with *control function* $g(C_1, \ldots, C_m)$. For instance, using this notation, (2) can be rewritten as

$$\begin{aligned} & \{R_1, \ldots, R_n\}_{g(C_1,\ldots,C_m)} = \\ & \{R_1, \ldots, (R_j \bowtie^-_{C_i} R_k), \ldots, R_n\}_{g|C_i=false(C_1,\ldots,C_{i-1},C_{i+1},\ldots,C_m)} \,\dot\cup \\ & \{R_1, \ldots, (R_j \bowtie^+_{C_i} R_k), \ldots, R_n\}_{g|C_i=true(C_1,\ldots,C_{i-1},C_{i+1},\ldots,C_m)}. \end{aligned}$$

For a query with a Boolean function g, the following algorithms start with the initial bundle $\{R_1, \ldots, R_n\}_g$ and apply (1) and (2) repeatedly until a set of bundles with control functions $g' = true$ or $g' = false$ is obtained. A more detailed description of the algorithm based on pseudocode is given in Appendix A.

4.2.2 *Application for the Example Query*

Let us restate the example query "Immigration Airports" in calculus representation:

$$\begin{aligned} & \{[a] \mid \exists f : a \in Airport \wedge f \in Flight \\ & \wedge (C_{USA}(a) \vee (J_{from}(a, f) \wedge C_{to}(f) \wedge C_{distance}(f)))\}. \end{aligned}$$

The starting bundle is derived from the range conditions, i.e., from $a \in Airport$ and $f \in Flight$. We obtain the following bundle:

$$B_1 = \\ read_a(Airport), read_f(Flight)\}_{C_{USA}(a) \vee (J_{from}(a,f) \wedge C_{to}(f) \wedge C_{distance}(f))}.$$

The control function of this bundle corresponds to the entire selection predicate. Now, one condition of the control function is introduced. Based on estimated selectivity and evaluation cost, the optimizer chooses $C_{USA}(a)$ and the following two bundles are generated:

$$B_{11} = \{\sigma^+_{C_{USA}(a)}(read_a(Airport)), read_f(Flight)\}_{true}$$
$$B_{12} = \{\sigma^-_{C_{USA}(a)}(read_a(Airport)), \\ read_f(Flight)\}_{J_{from}(a,f) \wedge C_{to}(f) \wedge C_{distance}(f)}.$$

B_{11}'s control function ($g' = true$) indicates that the evaluation of this bundle already yields solution tuples. B_{11}'s tuples "bypass" the expensive condition $C_{distance}(f)$ and the join $J_{from}(a,f)$. Next, we introduce $C_{to}(f)$ into B_{12}:

$$B_{11} = \{\sigma^+_{C_{USA}(a)}(read_a(Airport)), read_f(Flight)\}_{true}$$
$$B_{121} = \{\sigma^-_{C_{USA}(a)}(read_a(Airport)), \\ \sigma^+_{C_{to}(f)}(read_f(Flight))\}_{J_{from}(a,f) \wedge C_{distance}(f)}$$
$$B_{122} = \{\sigma^-_{C_{USA}(a)}(read_a(Airport)), \sigma^-_{C_{to}(f)}(read_f(Flight))\}_{false}.$$

The bundle B_{122} bears the control function *false* and can thus be discarded (its tuples are certain not to be elements of the result set). The removal of B_{122} implies that only the *true*-output $\sigma^+_{C_{to}}$ yields tuples that are solution candidates. Hence, this operator is nothing else but the familiar one-output selection $\sigma_{C_{to}}$. After the next step, the introduction of $C_{distance}(f)$ into B_{121}, the following bundles are retained:

$$B_{11} = \{\sigma^+_{C_{USA}(a)}(read_a(Airport)), read_f(Flight)\}_{true}$$
$$B_{1211} = \{\sigma^-_{C_{USA}(a)}(read_a(Airport)), \\ \sigma^+_{C_{distance}(f)}(\sigma^+_{C_{to}(f)}(read_f(Flight)))\}_{J_{from}(a,f)}$$
$$B_{1212} = \{\sigma^-_{C_{USA}(a)}(read_a(Airport)), \\ \sigma^-_{C_{distance}(f)}(\sigma^+_{C_{to}(f)}(read_f(Flight)))\}_{false}.$$

Again, the *false*-stream constituting the bundle B_{1212} cannot lead to result tuples because its control function is *false*.

The next condition to be incorporated is $J_{from}(a,f)$ which joins the elements of B_{1211}. Applying (2) leaves us with two bundles with control functions $g' = true$:

$$B_{11} = \{\sigma^+_{C_{USA}(a)}(read_a(Airport)), read_f(Flight)\}_{true}$$
$$B_{12111} = \{\sigma^-_{C_{USA}(a)}(read_a(Airport)) \bowtie^+_{J_{from}(a,f)} \\ \sigma^+_{C_{distance}(f)}(\sigma^+_{C_{to}(f)}(read_f(Flight)))\}_{true}$$
$$B_{12112} = \{\sigma^-_{C_{USA}(a)}(read_a(Airport)) \bowtie^-_{J_{from}(a,f)} \\ \sigma^+_{C_{distance}(f)}(\sigma^+_{C_{to}(f)}(read_f(Flight)))\}_{false}.$$

Once again, the *false*-stream, i.e., bundle B_{12112}, cannot contribute result tuples. Since both remaining control functions—i.e., bundles B_{11} and B_{12111}—are reduced to *true*, the decomposition process terminates. The just exercised construction process is visualized in Fig. 7. Fig. 8 shows the resulting evaluation plan.

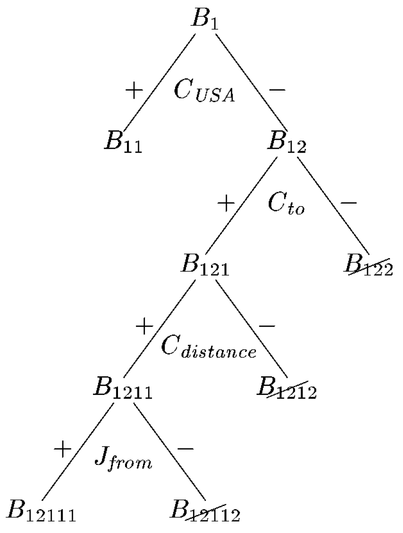


Fig. 7. Visualization of the construction process.

As a final optimization step, we can push projections that eliminate one relation down in the evaluation plan. In our example query, the *select distinct* clause indicates that set semantics is desired. Therefore, the projection operator can be pushed down in the evaluation plan. The resulting final query evaluation plan is:

$$\sigma^+_{C_{USA}(a)}(read_a(Airport)) \uplus \\ \sigma^-_{C_{USA}(a)}(read_a(Airport)) \ltimes_{J_{from}(a,f)} \\ \sigma_{C_{distance}(f)}(\sigma_{C_{to}(f)}(read_f(Flight))).$$

This plan has already been shown in Fig. 6. The case of an empty *Flight* extension is treated by the separate *if*-statement in the evaluation plan.

4.3 Generating Alternative Bypass Plans

The order for introducing conditions is chosen by the optimizer based on selectivity and evaluation cost estimation. It may sometimes be advantageous to construct

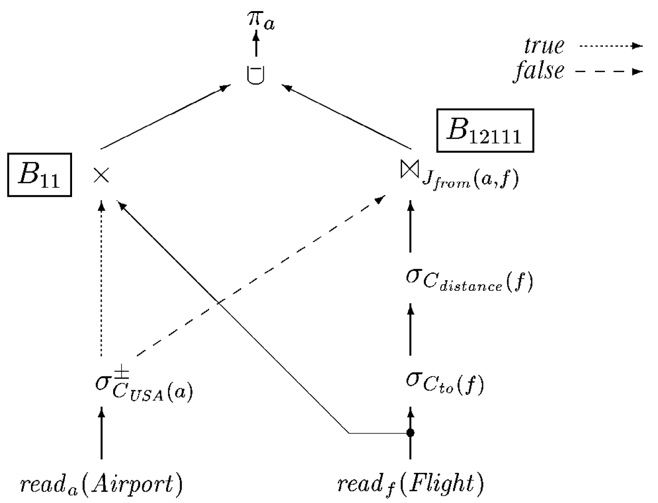


Fig. 8. Preliminary query evaluation plan for "Immigration Airports."

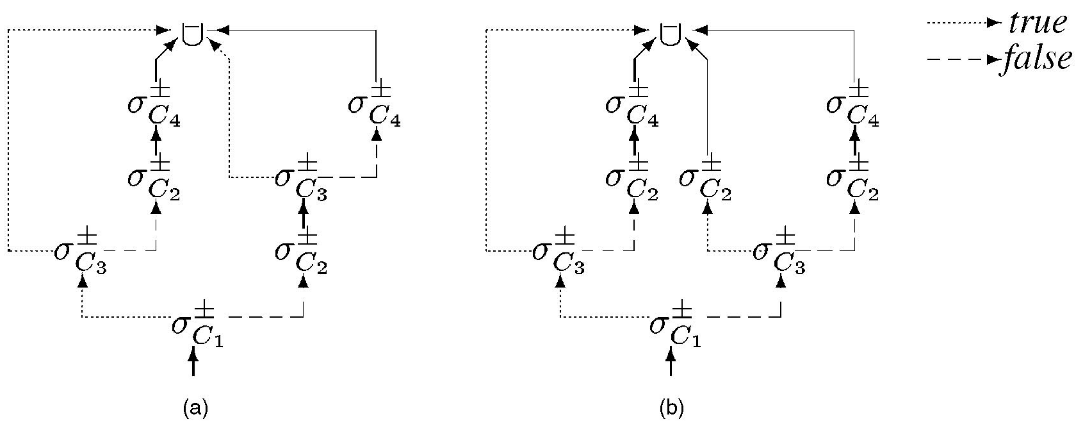


Fig. 9. Alternative plans generated by the (a) OPT and (b) FIX strategy.

evaluation plans where the evaluation orders are not determined globally, but independently for each possible path that a tuple might take from the first to the last stage. For instance, it may be the best solution to pursue the evaluation order C_1, C_2, C_3 if $C_1 = true$ for a particular tuple, but C_1, C_3, C_2 in case $C_1 = false$. This is the way the strategy "OPT" works: The order in which atomic conditions are introduced into bundles is chosen for each bundle independently. Thus, OPT is an exhaustive search algorithm which always finds the optimal plan.

In contrast to OPT, the "FIX" heuristics constructs evaluation plans where the conditions' evaluation order is the same for all possible paths from the first stage (*read*-operations) to the final stage (union of all disjoint streams). In other words, the evaluation order is always determined globally for the *entire* evaluation plan. Consider an example from [28], using the control function:

$$g = (C_1(x) \vee C_2(x)) \wedge (C_2(x) \vee C_3(x)) \wedge (C_3(x) \vee C_4(x)).$$

Given appropriate selectivities and cost functions for the conditions, the OPT strategy might yield the optimal bypass plan as shown in Fig. 9a. On the *true*-stream of C_1, the remaining conditions are evaluated in the order C_3, C_2, C_4, as opposed to the order C_2, C_3, C_4 on the *false*-stream. The FIX strategy globally prescribes the order of conditions for all streams such that the plan in Fig. 9a cannot be found by FIX. Instead, both output streams of C_1 must obey the same order, possibly resulting in the plan in Fig. 9b. A more thorough comparison of FIX and OPT plans is given in [28].

5 Treatment of Null Values

The treatment of null values requires a modification of the evaluation process. Let us consider the example

$$\sigma_{C \vee \neg C}(R).$$

There, the elements of R for which the condition C is *unknown* because of null values are not part of the result—at least not according to the standard SQL semantics.

There have been various proposals for treating null values in the literature. However, let us restrict the discussion to the SQL semantics which is based on a three-valued logic with the additional value *unknown*. The truth tables for the three-valued logic are as follows:

and	true	false	unknown
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

or	true	false	unknown
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

$\neg$	
true	false
false	true
unknown	unknown

In order to capture the SQL semantics, we need to refine our construction algorithm to reflect the presence of null values within the three-valued logic. Depending on whether the query predicate contains negations, we devise two cases. In the first case, where the query predicate does not contain any negation, we proceed as before and remain within two-valued logic by "mapping" any *unknown* value to *false*. As can be derived from the truth tables for *and* and *or* above it does not cause harm to handle *false* and *unknown* equivalently because tuples or objects qualify only if the entire selection predicate evaluates to *true*.

As soon as any negation occurs in the query predicate, however, this does not work any more. Just consider the above example predicate $C \vee \neg C$. Application of rule (1) would yield two bundles each with control function *true*, thus tuples with $C = unknown$ would pass through into the result via the *false*-stream. This would be a violation of SQL semantics.

To overcome the problem, we offer two alternative extensions to our construction algorithm.

5.1 (Potentially) Maintaining Three Streams

One approach is to convert (1) and (2) to three-valued logic such that a bypass operator possibly yields three streams:

1. the *true*-stream
2. the *false*-stream, and
3. the *unknown*-stream with tuples for which the corresponding selection or join predicate is *unknown*.

Thus, the two generating equations have the following form:

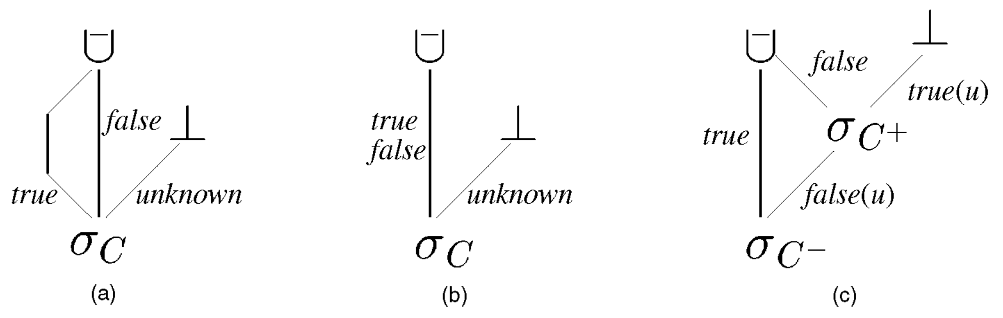


Fig. 10. Alternative evaluation plans for $\sigma_{C \vee \neg C}(R)$: (a) three-valued, (b) merged streams, (c) two-valued variant.

$$\begin{aligned} Q = {} & \sigma_{g|C_i=false}(R_1 \times \cdots \times \sigma_{C_i=false}(R_j) \times \cdots \times R_n) \,\dot\cup \\ & \sigma_{g|C_i=true}(R_1 \times \cdots \times \sigma_{C_i=true}(R_j) \times \cdots \times R_n) \,\dot\cup \\ & \sigma_{g|C_i=unknown}(R_1 \times \cdots \times \sigma_{C_i=unknown}(R_j) \times \cdots \times R_n) \end{aligned} \qquad (1')$$

$$\begin{aligned} Q = {} & \sigma_{g|C_i=false}(R_1 \times \cdots \times (R_j \bowtie_{C_i=false} R_k) \times \cdots \times R_n) \,\dot\cup \\ & \sigma_{g|C_i=true}(R_1 \times \cdots \times (R_j \bowtie_{C_i=true} R_k) \times \cdots \times R_n) \,\dot\cup \\ & \sigma_{g|C_i=unknown}(R_1 \times \cdots \times (R_j \bowtie_{C_i=unknown} R_k) \times \cdots \times R_n). \end{aligned} \qquad (2')$$

Applying the extended rule (1′) to our simple example $\sigma_{C \vee \neg C}(R)$ results in three bundles.[1]

$$\begin{aligned} \rightarrow B_1 &= \{\sigma_{C(r)=true}(read_r(R))\}_{true \vee \neg true \rightarrow true} \\ \rightarrow B_2 &= \{\sigma_{C(r)=false}(read_r(R))\}_{false \vee \neg false \rightarrow true} \\ B_3 &= \{\sigma_{C(r)=unknown}(read_r(R))\}_{unknown \vee \neg unknown \rightarrow unknown}. \end{aligned}$$

Simplification of the first two control functions yields *true*, while the last one remains *unknown*. In the final step, we "switch back" to two-valued logic, converting *unknown* to *false*. Thus, we obtain the plan

$$\sigma_{C(r)=true}(read_r(R)) \quad \dot\cup \quad \sigma_{C(r)=false}(read_r(R))$$

for evaluating the query $\sigma_{C \vee \neg C}(R)$ (see Fig. 10a). For this example, the two result streams may be collapsed to form a single output stream, as shown in Fig. 10b.

5.2 Different Polarizations of Condition Evaluation

The second approach is based on an idea proposed by von Bültzingsloewen [33]. A "trick" makes it possible to retain two-valued logic: We keep track of whether an unknown result should be mapped to *true* or *false*. A superscript $^+$ or $^-$ indicates that *unknown* yields *true* or *false*, respectively. Let us call this the *positive* or *negative polarization* of a condition. Thus, a condition C^+ yields *true* if the three-valued logic result were *unknown*, whereas C^- yields *false* if the three-valued logic result were *unknown*. These polarizations are not to be confused with the two output streams of the bypass selection or bypass join operators. For example, the operator $\sigma^+_{C^-}$ yields the *true*-stream of the selection condition C with negative polarization and $\sigma^-_{C^-}$ yields the corresponding *false*-stream (which also contains the *unknown* tuples because of the negative polarization of the condition).

The key idea of this approach can be derived from Table 2a, b, c. Let us first concentrate on Table 2b which shows the three-valued truth-table for a selection predicate consisting of a conjunction of an atomic condition C (negated and nonnegated) and the remaining predicate P, i.e., a predicate of the forms "$\neg C \wedge P$" or "$C \wedge P$." Keep in mind that—in the end—the selection predicate has to yield *true* in order for the corresponding object (tuple) to qualify. From the truth tables, we observe that the final outcome of the predicate "$\neg C \wedge P$" is equivalent for $C = true$ and $C = unknown$. This is indicated by the two groups of bold-faced truth-values. On the other hand, for the predicate "$CwedgeP$," the final result is equivalent for $C = false$ and $C = unknown$. The analogy holds for a selection predicate with a disjunction, which is of the forms "$\neg C \vee P$" or "$CveeP$." This is shown in Table 2c. In addition, Table 2a shows that, for a selection predicate consisting of merely one negated atomic condition (i.e., "$\neg C$") the result *true* is obtained only if $C = false$. Again, $C = true$ and $C = unknown$ are equivalent as far as the final result is concerned.

Let us summarize these equivalences in the following table, where we use (the above introduced) polarizations for the atomic condition C:

three-valued logic		two-valued logic
$C = true$	$\Leftrightarrow$	$C^- = true$
$\neg C = true$	$\Leftrightarrow$	$\neg(C^+) = true$
$C \wedge P = true$	$\Leftrightarrow$	$C^- \wedge P = true$
$\neg C \wedge P = true$	$\Leftrightarrow$	$\neg(C^+) \wedge P = true$
$C \vee P = true$	$\Leftrightarrow$	$C^- \vee P = true$
$\neg C \vee P = true$	$\Leftrightarrow$	$\neg(C^+) \vee P = true$

Therefore, a selection predicate without any "global" negations can be converted to an equivalent predicate with polarized atomic conditions that are evaluated in two-valued logic.

In order to make use of this solution, we proceed as follows:

1. Move all negations to atomic conditions applying DeMorgan's laws.
2. Eliminate multiple negations by applying the rule of double negation $\neg\neg C = C$.

1. In the following, a bundle whose control function definitely evaluates to *true* is marked with a leading arrow and simplifications of control functions are indicated by a small subsequent arrow.

TABLE 2
Mapping from Three-Valued Logic to Two-Value Logic

C	$\neg C$	$\neg\neg C$
t	**f**	t
f	t	**f**
u	u→**f**	u→**f**

(a)

C	P	$\neg C \wedge P$	$C \wedge P$
t	t	**f**	t
t	f	**f**	f
t	u	**f**	u→f
f	t	t	**f**
f	f	f	**f**
f	u	u	**f**
u	t	u→**f**	u→**f**
u	f	**f**	**f**
u	u	u→**f**	u→**f**

(b)

C	P	$\neg C \vee P$	$C \vee P$
t	t	**t**	t
t	f	**f**	t
t	u	u→**f**	t
f	t	t	**t**
f	f	t	**f**
f	u	t	u→**f**
u	t	**t**	**t**
u	f	u→**f**	u→**f**
u	u	u→**f**	u→**f**

(c)

3.

   a. Polarize each condition C without a preceding negation as C^-.
   b. Polarize each condition C with a preceding negation to C^+.

4. Now, apply the "normal" construction process for generating bypass evaluation plans as described in Section 4.2; however, differently polarized occurrences of a condition, i.e., C^+ and C^-, have to be treated as different conditions.

Note that the first two steps are carried out within three-valued logic and, in Step 3, we switch to two-valued logic by polarizing all the atomic conditions appropriately. In the following, we will assume that the polarizations have higher precedence than the negation, i.e., $\neg(C^+)$ will simply be denoted as $\neg C^+$.

5.3 Some Examples

Reconsider our example query $\sigma_{C \vee \neg C}(R)$. Using the second approach, we first check for negations that have to be moved "inside." Since the only negation is already placed correctly, we proceed with the polarization. The predicate becomes $C^- \wedge \neg C^+$. Then, we employ rule (1) and substitute the conditions C^+ and C^-, resulting in the evaluation plan depicted in Fig. 10c. The additional (u) adornment indicates the route that *unknown* tuples take.

Let us now examine our two approaches by means of another example. We will look at the predicate

$$g = neg(C_1 \wedge C_2) \vee (C_1 \wedge C_3):$$

1. Using the first approach based on three-valued logic, we obtain the following bundles (*unknown* is abbreviated to *unk*, simplifications of the control functions are indicated by a small subsequent arrow):

$$B_1 = \{\sigma_{C_1(r)=true}(read_r(R))\}_{\neg(true \wedge C_2)\vee(true \wedge C_3)}$$

$$\rightarrow B_2 = \{\sigma_{C_1(r)=false}(read_r(R))\}_{\neg(false \wedge C_2)\vee(false \wedge C_3)\rightarrow true}$$

$$B_3 = \{\sigma_{C_1(r)=unk}(read_r(R))\}_{\neg(unk \wedge C_2)\vee(unk \wedge C_3)}$$

$$B_{11} = \{\sigma_{C_2(r)=true}(\sigma_{C_1(r)=true}(read_r(R)))\}_{\neg(true \wedge true)\vee(true \wedge C_3)}$$

$$\rightarrow B_{12} = \{\sigma_{C_2(r)=false}(\sigma_{C_1(r)=true}(read_r(R)))\}_{\neg(true \wedge false)\vee(true \wedge C_3)\rightarrow true}$$

$$B_{13} = \{\sigma_{C_2(r)=unk}(\sigma_{C_1(r)=true}(read_r(R)))\}_{\neg(true \wedge unk)\vee(true \wedge C_3)}$$

$$B_{31} = \{\sigma_{C_2(r)=true}(\sigma_{C_1(r)=unk}(read_r(R)))\}_{\neg(unk \wedge true)\vee(unk \wedge C_3)\rightarrow unk}$$

$$\rightarrow B_{32} = \{\sigma_{C_2(r)=false}(\sigma_{C_1(r)=unk}(read_r(R)))\}_{\neg(unk \wedge false)\vee(unk \wedge C_3)\rightarrow true}$$

$$B_{33} = \{\sigma_{C_2(r)=unk}(\sigma_{C_1(r)=unk}(read_r(R)))\}_{\neg(unk \wedge unk)\vee(unk \wedge C_3)\rightarrow unk}$$

$$\rightarrow B_{111} = \{\sigma_{C_3(r)=true}(\sigma_{C_2(r)=true}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge true)\vee(true \wedge true)\rightarrow true}$$

$$B_{112} = \{\sigma_{C_3(r)=false}(\sigma_{C_2(r)=true}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge true)\vee(true \wedge false)\rightarrow false}$$

$$B_{113} = \{\sigma_{C_3(r)=unk}(\sigma_{C_2(r)=true}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge true)\vee(true \wedge unk)\rightarrow unk}$$

$$\rightarrow B_{131} = \{\sigma_{C_3(r)=true}(\sigma_{C_2(r)=unk}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge unk)\vee(true \wedge true)\rightarrow true}$$

$$B_{132} = \{\sigma_{C_3(r)=false}(\sigma_{C_2(r)=unk}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge unk)\vee(true \wedge false)\rightarrow unk}$$

$$B_{133} = \{\sigma_{C_3(r)=false}(\sigma_{C_2(r)=unk}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge unk)\vee(true \wedge unk)\rightarrow unk}.$$

The lefthand side of Fig. 11 depicts the corresponding evaluation plan.

2. To make use of the second variant (i.e., the one with different polarizations of the conditions), we first

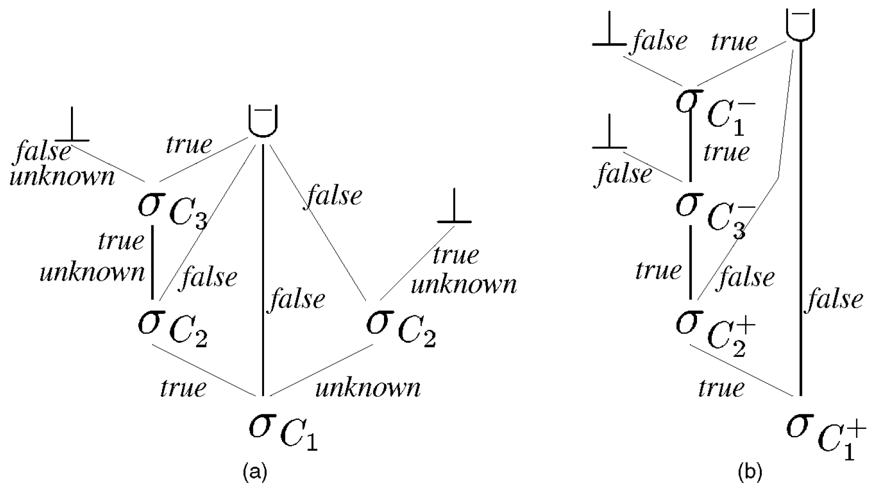


Fig. 11. Second example: $\sigma_{\neg(C_1 \wedge C_2) \vee (C_1 \wedge C_3)}(R)$.

transform our control function into an equivalent one that can be evaluated with two-valued logic, i.e., we apply DeMorgan's law to move the negation and then we polarize the conditions:

$$\neg(C_1 \wedge C_2) \vee (C_1 \wedge C_3) = \neg C_1^+ \vee \neg C_2^+ \vee (C_1^- \wedge C_3^-).$$

Now, (1) and (2) for two-valued logic can be used (remember that C_1^+ and C_1^- have to be treated as different conditions, i.e., the substitution of a constant *true* or *false* for C_1^+ does not affect C_1^-).

$$\begin{aligned}
B_1 &= \{\sigma_{C_1^+(r)=true}(read_r(R))\}_{\neg true \vee \neg C_2^+ \vee (C_1^- \wedge C_3^-)} \\
\rightarrow B_2 &= \{\sigma_{C_1^+(r)=false} \\
&\quad (read_r(R))\}_{\neg false \vee \neg C_2^+ \vee (C_1^- \wedge C_3^-) \rightarrow true} \\
B_{11} &= \{\sigma_{C_2^+(r)=true}(\sigma_{C_1^+(r)=true} \\
&\quad (read_r(R)))\}_{\neg true \vee (C_1^- \wedge C_3^-)} \\
\rightarrow B_{12} &= \{\sigma_{C_2^+(r)=false}(\sigma_{C_1^+(r)=true} \\
&\quad (read_r(R)))\}_{\neg false \vee (C_1^- \wedge C_3^-) \rightarrow true} \\
B_{111} &= \{\sigma_{C_3^-(r)=true}(\sigma_{C_2^+(r)=true}(\sigma_{C_1^+(r)=true} \\
&\quad (read_r(R))))\}_{(true \wedge C_3^-)} \\
B_{112} &= \{\sigma_{C_3^-(r)=false}(\sigma_{C_2^+(r)=true}(\sigma_{C_1^+(r)=true} \\
&\quad (read_r(R))))\}_{(false \wedge C_3^-) \rightarrow false} \\
\rightarrow B_{1111} &= \{\sigma_{C_1^-(r)=true}(\sigma_{C_3^-(r)=true}(\sigma_{C_2^+(r)=true} \\
&\quad (\sigma_{C_1^+(r)=true}(read_r(R)))))\}_{true} \\
B_{1112} &= \{\sigma_{C_1^-(r)=false}(\sigma_{C_3^-(r)=true}(\sigma_{C_2^+(r)=true} \\
&\quad (\sigma_{C_1^+(r)=true}(read_r(R)))))\}_{false}.
\end{aligned}$$

Again, the corresponding evaluation plan is shown in Fig. 11b.

5.4 Discussion of the Two Approaches

Both presented variants have pros and cons. The three-valued approach sometimes introduces three output streams for bypass operators, possibly causing higher implementation and evaluation effort. The overhead is alleviated, however, by the possibility to combine two of the three output streams. This may be done (at optimization time) in our first example. On the other hand, the second solution guarantees the evaluation using always two streams, at the additional cost of evaluating some conditions twice, like C_1^+ and C_1^- in the second example. Multiple evaluation of a condition C is only necessary, however, if there are already multiple occurrences of C in the original predicate and if these multiple occurrences end with different polarizations. Summarizing, either variant causes a limited overhead in some cases. The polarization approach offers the additional advantage that these special cases are user controllable, i.e., they occur only if the user explicitly uses a condition C multiple times in a predicate. Furthermore, the only required change to the query engine is a slight modification of predicate evaluation as opposed to introducing a third output stream in order to implement the first approach. Keep in mind, however, that any special handling of null values is only necessary if the query predicate contains any negations at all. If it does not contain negations, *unknown* can simply be handled like *false*.

6 Evaluation Techniques for Bypass Operations

In this section, we will discuss some aspects of implementing bypass operations in a query execution engine. First, we outline some general architectural alternatives for query engine implementation that are relevant to bypass evaluation, afterward we sketch some details of our system, especially with respect to the implementation of bypass operators. The query engine served as the platform for the experiments presented in the subsequent section.

6.1 Query Evaluation Strategies

Apart from the algorithms for implementing algebra operators, choosing techniques for operator communication and synchronization are critical tasks. Of course, these design decisions are correlated. In the following, we restrict ourselves to a client/server system where each query constitutes a client to a page server. Other architectures,

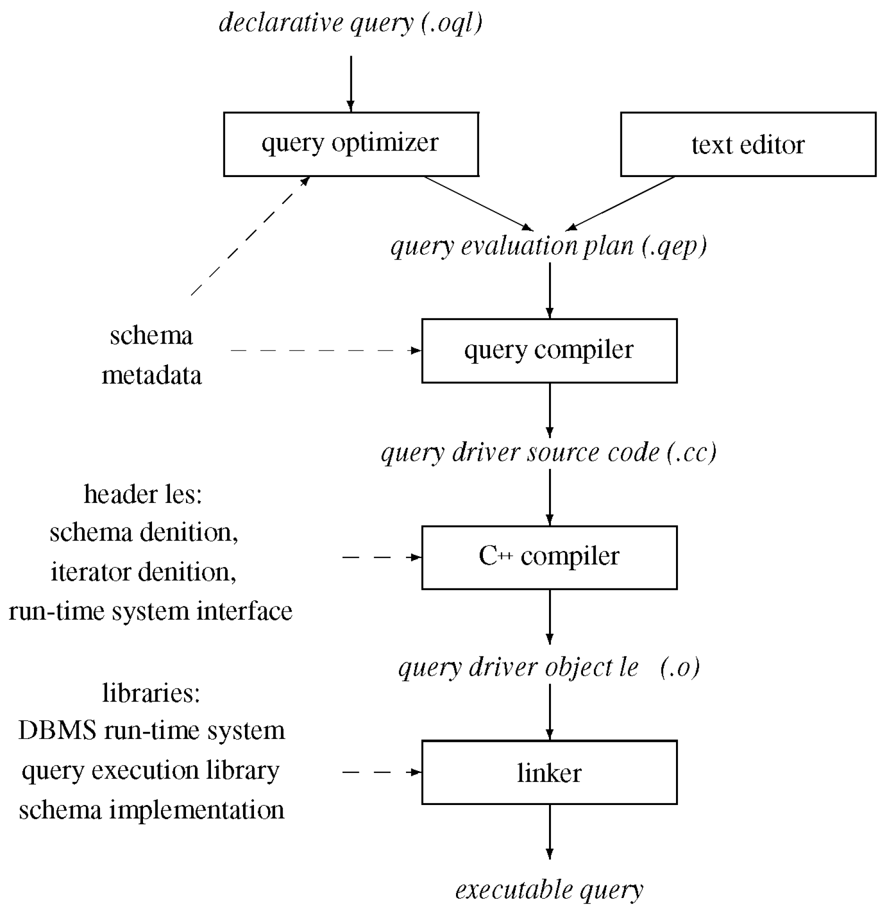

declarative query (.oql)
query optimizer
text editor
query evaluation plan (.qep)
schema metadata
query compiler
query driver source code (.cc)
header les: schema denition, iterator denition, run-time system interface
C++ compiler
query driver object le (.o)
libraries: DBMS run-time system query execution library schema implementation
linker
executable query

Fig. 12. Query generation from query evaluation plans.

especially parallel and distributed execution, are out of the scope of this paper.

6.1.1 Operator Scheduling

If operators do not pass complete tables but single records or groups of records, one has to consider how control flow is managed between operators. One approach that is very easy to implement, especially when using per-record data passing, is *demand driven control flow*. Starting at the top of the QEP, each operator asks its input operator(s) to produce records whenever it is ready to process new data. When implementing operators as functions, this constitutes a simple function call. Demand driven data flow has the advantage that data is only computed if really needed. The simplicity not only facilitates the implementation, but also minimizes scheduling overhead (that is, no scheduler is necessary).

The alternative to demand driven data flow is *data-driven control flow*. In this case, processing starts at the leaves of a query plan and proceeds bottom-up within the tree. Since here we usually have several starting points (as opposed to a single top), a simple implementation via function calls does not suffice. An operator-independent manager module, like a simple dispatcher, has to control the interaction of operators. This would be advantageous, anyway, as soon as parallel execution is taken into account.

6.1.2 Implementation

In our query engine, data is passed on a per-record basis by default, but can also be transferred in units of pages. As long as possible, only references to data are passed between operators. Algebra operators are implemented as *iterators* as proposed, e.g., by Graefe [34]. An iterator is an abstract data type offering at least the functions *Open*, *Next*, and *Close*, for initializing the iterator, producing a record, and final clean-up, respectively. Considering the above classification, the iterators implement demand-driven control flow.

The query engine is implemented in C++ [35]. Each physical operator is implemented as a C++ class derived from the abstract class *QE_iterator*, providing the above mentioned virtual functions *Open*, *Next*, and *Close*. The library of operators includes two kinds of file scans, e.g., $read_a(Airport)$, one for scanning an index and returning OIDs, another one for scanning a file and returning objects. Furthermore, there is an expand operator for accessing objects and computing method results (χ, cf. Section 3), nested loops join ($\bowtie^{\mathrm{NL}}$), hybrid hash and merge join ($\bowtie^{\mathrm{hash}}$, $\bowtie^{\mathrm{merge}}$), selection (σ), merge union ($\Cup$), set operations ($\uplus, -, \ldots$), and a buffering operator (buf).

Fig. 12 depicts the generation of an executable query. A query evaluation plan is a plain text file, thus allowing manual editing for experimentation purpose. Usually, the plan is generated by the query optimizer, in our case by our blackboard optimizer [16]. The query compiler processes the query execution plan as input, amends it with some

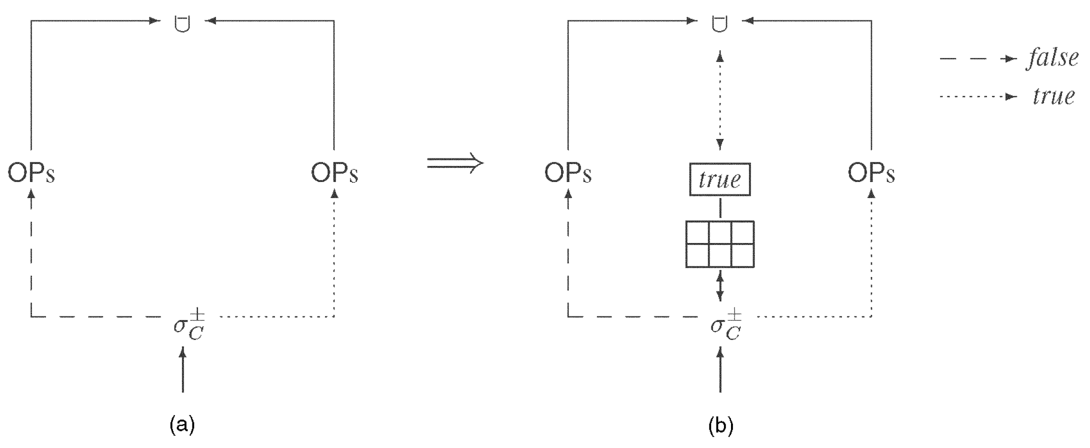


Fig. 13. Structure of bypass cycles in evaluation plans. (a) General structure. (b) Implementation concept.

information about resource management and DBMS access, and finally generates a C++ program that serves as a "driver" for the query evaluation. The driver instantiates operators from the library and supplies them with additional parameters and support functions, e.g., predicate-, hash-, and copying functions. The driver program is compiled with a regular C++ compiler. The generated object file is linked with the DBMS run-time system, the operator library, and the implementation of methods of the underlying database schema to form an executable program. Alternatively, the query compiler can serve as an interpreter instantiating the operators and driving the query itself. This saves the compilation and linking phase at the cost of some more run-time overhead.

6.2 Implementing Bypass Selection

Bypass operators have (at least) two output streams.[2] Since we assume a single result table, however, every plan has only one topmost operator. Consequently, query evaluation plans using bypass operators do not form a tree, as conventional plans do, but, instead, they are DAGs. Fig. 13a shows the general structure of a bypass DAG with a fork at a bypass selection and a subsequent merge at the corresponding merge-union ($\uplus$) operator. These fork/merge-cycles do not cause any problems if the operators are evaluated strictly sequentially and each operator stores its intermediate result anyway. For the evaluation with a demand-driven query engine, the top-down evaluation strategy forces a merge union operator to choose one input stream each time it is called. One (naive) approach to choosing the correct stream is to abandon strict top-down control flow and to evaluate the part of the query beneath the bypass operator first in order to find out along which branch the record will go. This approach, however, contains a pitfall: Due to the nature of the bypass evaluation plans, at least one branch of the bypass cycle contains several other operators (as indicated by "OPs" in Fig. 13). When trying to follow the "previewed" select result and choosing one input stream, the expected record may possibly be filtered out on the path to the merge union. Since this naive lookup mechanism fails, we have decided to use an additional buffer in our prototype to collect records that belong to the "wrong" output stream once we have selected a particular stream.

The conceptual implementation of the buffering technique is shown in Fig. 13b. The grid symbolizes the bypass buffer. There is only one buffer for both output streams, and a switch, depicted in Fig. 13 by the box currently containing the label "*true*," is used to mark what records are cached in the buffer. For three output streams, two buffers are required, each equipped with a label indicating which output streams are buffered (*true*, *false*, or *unknown*) and with a counter for the number of stored records. Depending on the buffer state, the bypass operator either serves a *Next*-call from the buffer, or it retrieves new records from its input operator and saves "misdirected" records (i.e., designated for the other output stream) in the buffer until one matching record can be returned. Whenever the merge union operator has to choose the input operator, it looks at the label of the buffer and selects an input that will fetch a record from this buffer. This keeps the buffer quite small in practice, but since buffer growth cannot be avoided in all cases, overflow pages of the buffer may be written to temporary segments, using the same mechanisms that are used, e.g., for run files of an external sort operator or for partition files of hash operators. In the worst case, assuming the merge union operator chooses one input and all records go to the other stream, the fraction of the bypass input that does not fit into the buffer is written to the temporary segment. In a way, this resembles the predicate caching solved by [30]. However, our problem is much simpler since we only retrieve objects sequentially from the cache. Therefore, we need not maintain any data structures for random access (e.g., hash tables) as [30] does. Other merge union policies, apart from *lookup*, are feasible; especially if very high or very low selectivity is expected, simply exhausting one input while buffering nonmatching records may be useful. If multiple bypass operators occur within one plan, the corresponding $\uplus$ operators may either be nested (the topmost $\uplus$ communicating with the

2. Actually, when implementing the three-valued treatment of null-values according to Section 5.1, bypass operators could have three output streams.

bottom-most bypass operator), or an n-ary $\uplus$ combines all streams in one step.

A totally different way to avoid large bypass buffers is to cause a rollback of execution: If too many nonmatching records are found, control is given back to the calling merge union operator, associated with a hint to call the other input. Then, writing overflow pages to a temporary segment is not necessary at all. The rollback mechanism requires, however, even if implemented via C++ exceptions, that all operators can cope with interruption and are able to resume their work at the point of interruption.

6.3 Implementing Bypass Joins

Essentially, everything mentioned in the previous section also applies to bypass joins. In addition, some features of efficient join evaluation algorithms can be exploited.

In contrast to simple selection operators, some join algorithms, like hash join and sort merge, join allocate larger amounts of memory to make efficient evaluation possible. This implies that a larger number of records is already present in memory. If the join operator has bypassing semantics, it can profit from the in-memory structure. Instead of copying records for the "wrong" output stream to the bypass buffer, they can often be kept in memory, e.g., in a hash table or in a buffer used by a merge join. Generally, however, one cannot omit the bypass buffer totally. Suppose, for example, that a hash partition has been processed completely and memory is to be freed to process the next partition. In this case, it is possible that there remain records for one output stream. In order to be able to continue processing, the remaining records must be flushed to the bypass buffer before a new hash partition can be processed.

For antijoins (i.e., $\bowtie^-$), processing is usually quite expensive. It does not suffice to find matching records, but, additionally, nonmatching records must be combined with all records from the other input, i.e., the bypass join simply performs a partitioning of the cross product. It incurs, therefore, the high cost of a cross product (therefore, the optimizer will rarely generate plans containing these operators). Because of the indispensable cost of the cross product, we offer only two implementation methods: One uses a nested loops algorithm in connection with a bypass buffer. It does not impose any restrictions at all, but requires the predicate evaluation for each combination of input tuples. The other implementation assumes that one table will fit into main memory. Then, all conventional join algorithms are applicable. For example, a hash join could first mark those tuples with a match found and, afterward, output the unmarked tuples.

As with conventional semijoins, bypass semijoins can be processed very efficiently. Since the total cardinality is bounded by the size of one input stream, the additional *false* output stream does not increase processing cost very much. This is especially true for using hash-based methods since the hash table may be used as bypass buffer if the build input forms the returned output stream. Processing one partition is then performed in two phases. First, the normal algorithm is used, building the hash table from the build input and using the probe input to mark matching records in the hash table. In a second phase, the hash operator starts returning records, choosing the output stream for each record from the mark that was set in the first phase. Of course, the control mechanisms for the merge-union ($\uplus$) work as before, with the additional advantage that the bypass join operator knows the number of *true*/*false* records present in the hash table.

This section has shown how bypass operators can be integrated into conventional query engines with reasonable effort. The following section will prove that the proposed implementation offers good performance, too.

7 Quantitative Assessment

In this section, we present a quantitative assessment of our novel bypass technique. In order to increase the intuitive understanding of the results we derive the initial benchmark setup from the introductory example, i.e., we have an object schema with meaningful queries.

Another (second) way for specifying benchmarks would be first to generate all scenarios consisting of k object extensions, l restrictions, and m joins, second to optimize them by means of bypass, CNF-based, and DNF-based techniques for numerous settings (we have to vary the cardinality of each extension and the selectivity, as well as the cost value of each condition), and, third, to compare the resulting evaluation plans with respect to estimated costs and evaluation times. Since this would result in an enormous count of (possibly unrealistic) experiments, we have chosen a compromise: We first benchmark the "Immigration Airports" query using the same parameters as in the introductory section. Then, we continue benchmarking this query, but we examine the variation of cardinality and selectivity parameters—sometimes leading to unrealistic cases, as, e.g., databases with more airports than flights. In a second part, we examine abstract queries on a schema with three extensions.

7.1 Benchmark Environment

Before we discuss individual benchmarks, we first describe the benchmark environment common to all queries. The experiments were run on a two-processor Sun SPARCstation 20 Model 502MP, running under the Solaris 2.5 operating system. The system is equipped with 64 MB of main memory and a single 2 GB disk (Seagate ST12400N, average access time 9ms read, 10.5ms write). Since, for all queries, we measured elapsed real time, the system was taken off/line while running the experiments in order to get reproduceable results. As persistent storage system, we used the locally developed *Merlin* client-server storage system [36]. The system provides, e.g., storage manager, buffering, and index structures. On top of the run-time system, we built the query engine described in the previous section. Both page server and querying client were run on the same machine and the database was held on the local disk, thus avoiding any network traffic and interference. On the client side, a segmented page buffer was used to cache data pages. Temporary files were handled in the same way as persistent data files, i.e., they were transported to the (local) server.

condition	selectivity
C_{to}	0.4
$C_{distance}$	0.1
C_{USA}	*var*
J_{from}	1/#Airports
#Airport	1000
#Flight	29000

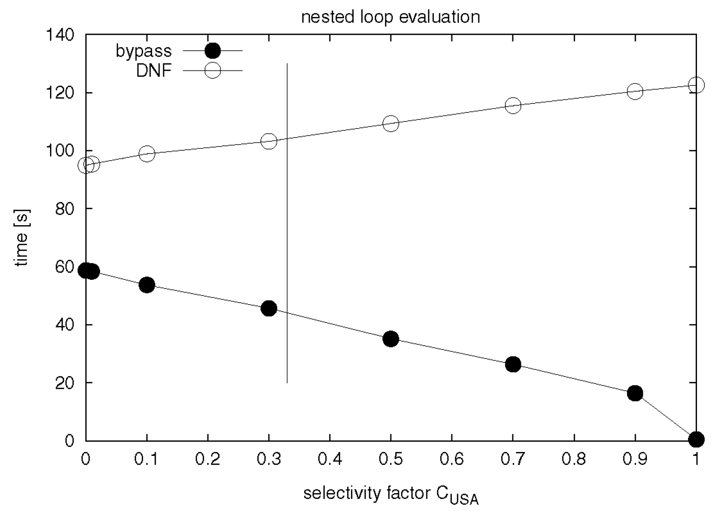


Fig. 14. Nested loops join evaluation of "Immigration Airports" query.

condition	selectivity
C_{to}	0.4
$C_{distance}$	0.1
C_{USA}	*var*
J_{from}	1/#Airports
#Airport	1000
#Flight	29000

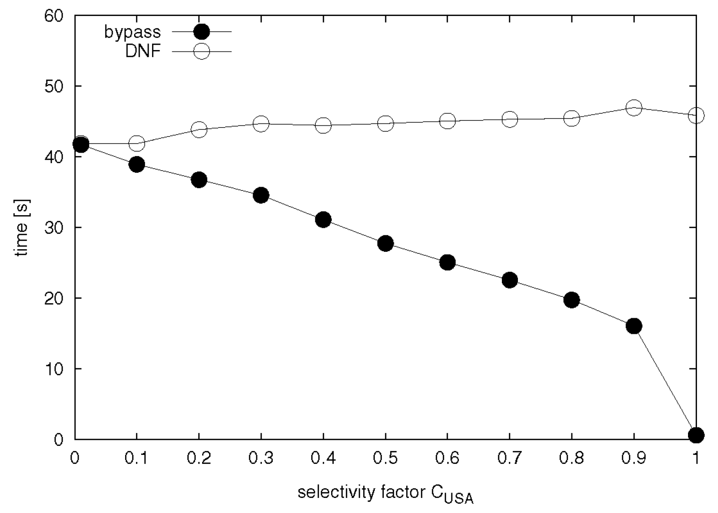


Fig. 15. Hash join evaluation of "Immigration Airports" query.

7.2 Running the "Immigration Airport" Query

We will first show experiments with the "Immigration Airports" query already mentioned in Section 2.1. We have created a database of 1,000 Airport objects and 29,000 Flight objects. Each object had a constant size of 400 bytes, consisting of the attributes mentioned before (coded as integers) and an additional string attribute filling the remaining bytes. This resulted in an on-disk database of about 13MB. The attribute values were distributed uniformly in the range 0 to 9999. The selectivities of conditions C_x were modified by comparing the attribute value with a varying constant. We have run all plans mentioned before: CNF (Fig. 4), DNF (Fig. 5), and bypass (Fig. 6). The client page buffer was always configured to a total of 600KB. Since the CNF-based plan can only be evaluated with a nested loops join, we have first tried to provide the same conditions for all plans, using nested loops join in all three plans. Fig. 14 shows the result of running the DNF and bypass query using a nested loops join. The selectivities of C_{to} and $C_{distance}$ have been chosen as in the example, the selectivity C_{USA} of the bypass operator has been varied through the full range. The value of $C_{USA} = 0.33$ from Table 1 has been marked with a vertical bar. The actually consumed time for the bypass query at this point is about 43 seconds. The DNF plan with the same parameters took about 105 seconds, which is about 240 percent of the bypass plan's cost. The CNF plan, not shown in the figure, took 6,450 seconds (factor 150 against the bypass plan). Since the CNF-based plans could not compete in any way with bypass and DNF, we have omitted any further figures regarding CNF.

The higher cost of DNF is partly due to the additional duplicate elimination. Duplicate elimination suffered in this case from the fact that a large amount of buffer space has been allocated to the nested loops (semi)join. With increasing C_{USA}, the runtime of the bypass query continuously decreases (since the number of tuples bypassing the costly join operation increases), while the runtime of the DNF query slightly increases due to growing cost for duplicate elimination. This first chart already shows the tendency that will be observed in most following plots: With the bypass selectivity factor (in this case, C_{USA}) close to 0, execution time of DNF and bypass plans do not differ too much. With

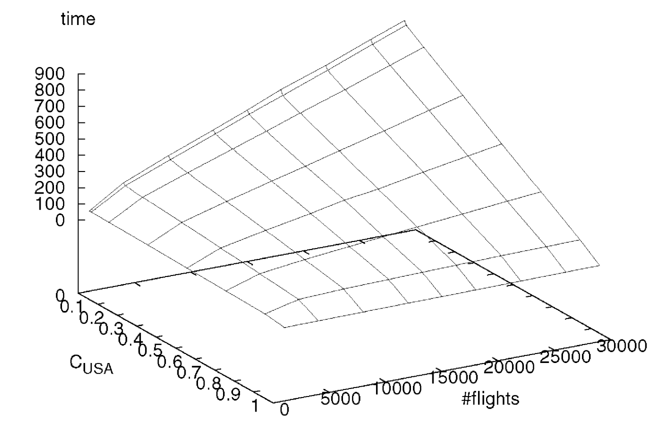


Fig. 16. Evaluating "Immigration Airports" on different databases with bypass.

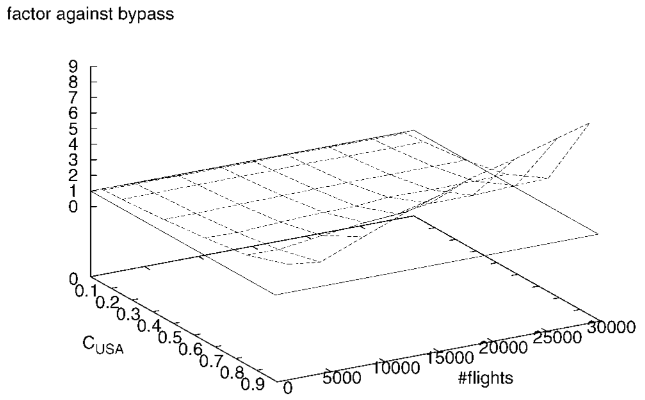


Fig. 17. Comparing DNF to bypass.

increasing selectivity factor, the bypassing advantage increases, too.

The same experiment has been performed using a hybrid hash join algorithm for join evaluation. The results are depicted in Fig. 15. The tendencies are the same as in the previous experiment, with the only difference that the fixed cost of duplicate elimination does not cause a visible offset. Since the select operations for C_{to} and $C_{distance}$ enormously reduce the join input, the join operation is cheap in this case anyway such that even nested loops evaluation seems feasible in this example query.

7.3 Varying Other Parameters

So far, we have only changed the selectivity of the bypass operator. Now, we want to vary other parameters influencing the performance. Therefore, we do not restrict the database to close-to-reality data. Instead, we have built a variety of databases with different cardinalities. The total amount of objects is kept constant at a number of 30,000, with an object size of 400 bytes, as before. Altering the ratio of cardinalities of *Airport*/*Flight* has the same effect as changing the selectivities of C_{to} and $C_{distance}$ (i.e., changing the join input size), thus we did not investigate these selectivities. The result of a benchmark varying C_{USA} on the one hand and the ratio of object cardinalities on the other hand for the bypass query is shown in Fig. 16. The number of airports has been varied from 1,000 to 29,000, thus providing a ratio of #Airports/#Flights ranging from 1:29 to 29:1. The figure shows that minimal cost occurs if many airports can bypass the join, i.e., if the number of flights is quite small and the selectivity factor C_{USA} is close to 1. If only one factor, either selectivity or cardinality ratio, changes disadvantageously, execution time still stays quite low. Only if both factors fall into a disadvantageous range, execution time increases up to several hundred seconds.

Fig. 17 shows the relative cost of the DNF query in comparison to the bypass query of Fig. 16 with identical parameters. As one can see, the DNF query nearly always causes higher cost. Only in the case with very few airports and a selectivity factor for C_{USA} close to 0, i.e., hardly any records can take advantage of bypassing, is DNF better than bypassing. This is due to the buffering overhead of our current bypass implementation, as mentioned in the previous section. The bypass gain increases enormously with growing selectivity factor. In order to retain readability, we have restricted C_{USA} to a maximum of 0.9. Otherwise, the maximum factor of 48 would have scaled down everything else.

7.4 A More Complex Scenario

For generating more complex benchmark queries, we extend our schema to three object extensions E_1, E_2, and E_3, as e.g., $E_1 = Airport$, $E_2 = Flight$, and $E_3 = Airline$. The objects are denoted by e_1, e_2, and e_3. We assume one restriction C_i for each extension E_i, as, e.g., $e_1.location = "USA"$, and join predicates J_{ij} between E_i and E_j, as e.g., $e_1.country = e_3.nationality$. The queries always project on E_1. Then, a generic query is specified as follows:

$$\{e_1 \mid e_1 \in E_1 \wedge \exists e_2 \in E_2 \wedge \exists e_3 \in E_3 : \\ bool(C_1, C_2, C_3, J_{12}, J_{23}, J_{13})\},$$

where $bool$ determines a particular Boolean function consisting of the conditions given by the arguments. Evaluation of the predicate involves the invocation of member functions which are implemented alternatively as main memory operations or operations causing I/O by, e.g., evaluating a path expression. We have analyzed five representative Boolean functions $bool_1, \ldots, bool_5$. We only sketch the examined queries here. For further details about the queries refer to Appendix B.

The extensions of all object types E_i and T_i had a cardinality of 100,000 objects each, with objects of the constant size of 100 bytes. This resulted in a database size of about 42MB. For the second benchmark variant ("dereferencing"), additional type extensions $T_4 \ldots T_9$ have been created, again each with a cardinality of 100,000 objects and each object sized to 100 bytes. Thus, the total size of the database amounted to 130MB. The client page buffer was allocated on a per-operator basis. A memory chunk of 2MB was assigned to each hash operator (set union, hash join, and duplicate elimination), an amount of 1MB was asssigned to each bypass operator and to each extent (for scans and dereferencing). This is certainly not the optimal allocation policy; however, it works reasonably for the evaluated plans. When comparing bypass and DNF plans,

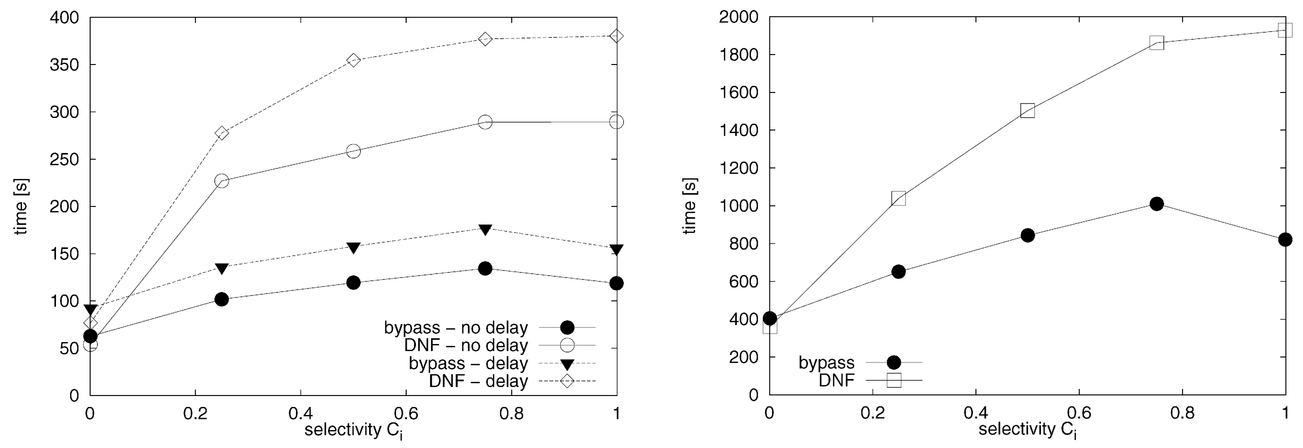


Fig. 18. $bool_1 = C_1 \vee (C_2 \wedge C_3 \wedge J_{12} \wedge J_{23})$.

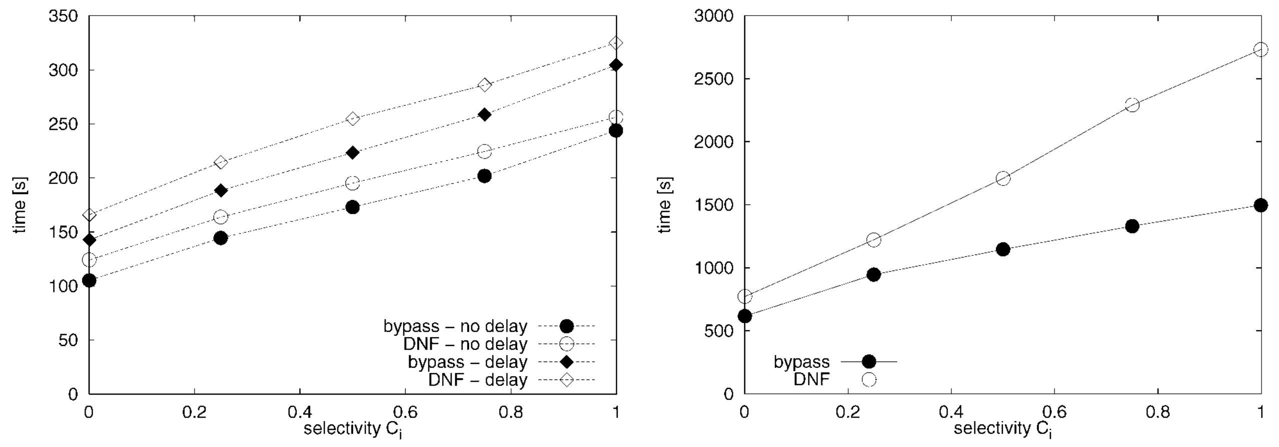


Fig. 19. $bool_2 = (C_1 \wedge J_{13} \wedge C_3) \vee (J_{12} \wedge C_2 \wedge J_{23})$.

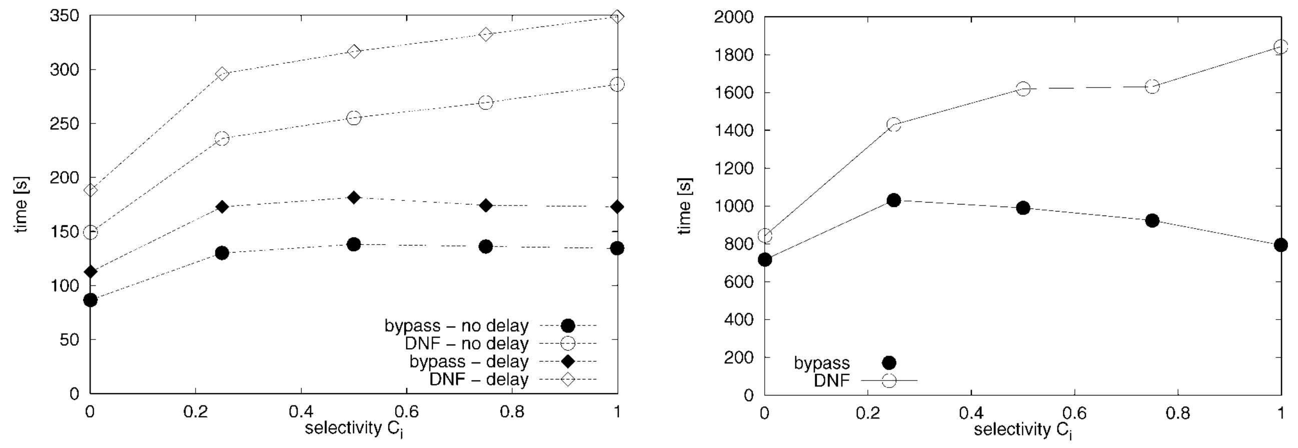


Fig. 20. $bool_3 = C_1 \vee (J_{12} \wedge (C_2 \vee (J_{23} \wedge C_3)))$.

the total amount of memory allocated by a DNF plan is larger than the amount allocated by the corresponding bypass plan since the set union operator obtains 2MB, while the merge union operator does not need any buffer space and the bypass buffer obtains only 1 MB. With respect to our benchmark results, this allocation is unfair against the bypass plans, but a fair strategy would only increase the performance advantage of bypass evaluation.

Figs. 18, 19, 20, 21, and 22 summarize the results of benchmarking the queries for $bool_1, \ldots, bool_5$. Each pair of diagrams contains variants with main memory operations for invoked type-based functions on the left-hand side and object accesses, i.e., I/O operations, on the righthand side. As before, all *bypass* queries are depicted by *filled* plot symbols, *DNF* queries by *hollow* symbols. We have generated matching CNF plans, too, but their restriction to nested loops evaluation always caused run-times orders

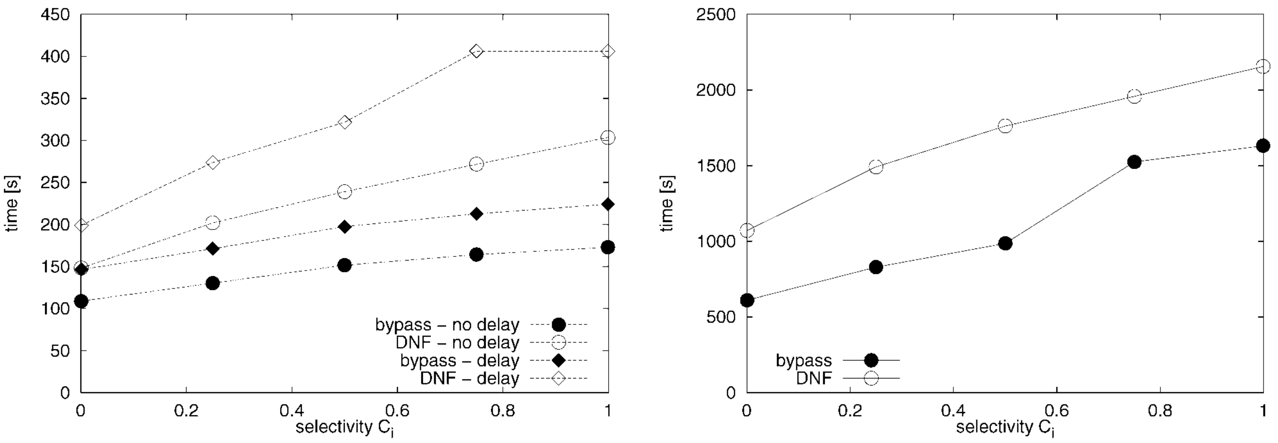


Fig. 21. $bool_4 = C_1 \vee (J_{13} \wedge C_3) \vee (J_{12} \wedge C_2 \wedge J_{23})$.

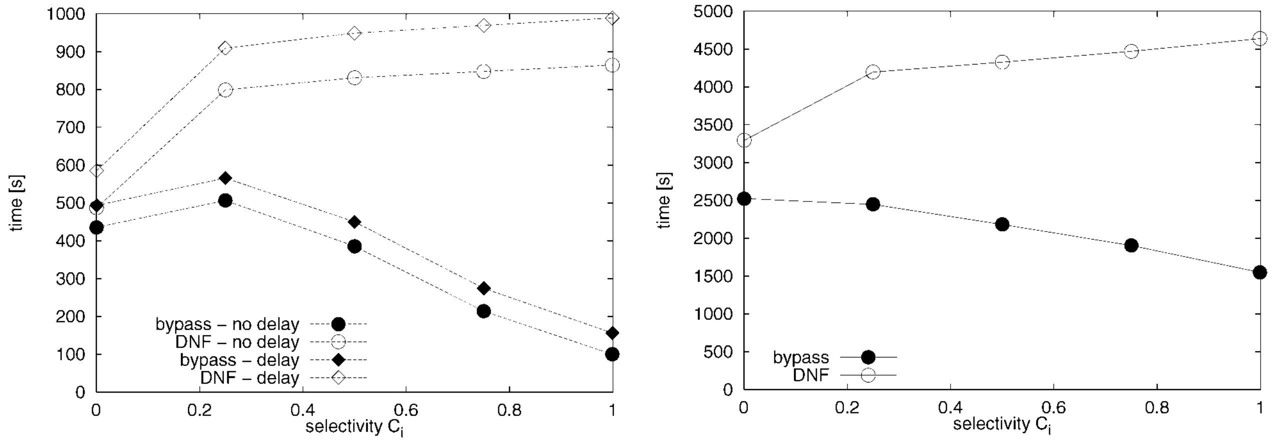


Fig. 22. $bool_5 = C_1 \vee (J_{12} \wedge (C_2 \vee (J_{23} \wedge (C_3 \vee J_{13}))))$.

of magnitudes larger than DNF and bypass and, therefore, they have been discarded in the figures. Since the construction of the evaluation plans has been explained before in Section 4, we refrain from showing the generated plans here. Sometimes the optimizer has generated more than one reasonable plan. In these cases, all feasible execution plans (i.e., those with close-to-optimal cost estimation) have been run and, for each data point, the best result for DNF and bypass has been plotted.

In all charts, the selectivities of all conditions C_i $(1 \leq i \leq 3)$ were equal. This has the consequence that effects caused by changing the selectivity of one condition may supersede contrary effects of changed selectivity of a different condition. For example, increasing the selectivity factor of a bypass operator causes a performance gain, but another selection operation feeds more records to a join as a consequence, thus reducing the visible bypass gain. Nevertheless, most charts still show an advantage of bypass evaluation. If, in all queries, we had only changed the selectivities of bypass operators, the advantage of bypassing would be even greater.

For the first benchmark (main memory operations for functions), we have separated two cases: First, all queries were run with all functions f_i as no-ops (marked as *no delay*). Second, these functions performed a random delay (implemented as a simple *for*-loop).

Let us browse through the single experiments of Figs. 18, 19, 20, 21, and 22. Depending on the structure of the Boolean function, there can be observed some tendencies. Both function $bool_1$ and $bool_2$ contain only one disjunction. They differ only in the position of this disjunction and the presence of a third join condition. The diagrams show that the bypass gain is quite small for $bool_2$, while much greater in the case of $bool_1$. From this difference, we can derive that bypassing is especially useful if the Boolean function is divided asymmetrically by disjunctions, giving one cheaper part (used as bypass predicate) and one more expensive part (that is bypassed), just as for $bool_1$.

The remaining functions embody two, respectively, three disjunctions. Fig. 22 shows the maximum bypass gain since the Boolean factors are nested and the innermost Boolean operator is a disjunction. Both properties enlarge the potential gain of bypass processing.

7.5 Summary of the Quantitative Analysis

In this section, we have first benchmarked the query from the introductory example; later the example has been extended to an abstract three extension schema, where a number of query patterns have been examined. Throughout all experiments, it was evident that bypass plans were either totally superior to DNF plans (in most cases) or there was a small selectivity range (close to zero) where DNF plans performed minimally better. These cases were due to overhead of managing a bypass buffer. This is a restriction of our current query engine and might well be eliminated in a planned more powerful implementation. The CNF plans were generally orders of magnitude inferior and were therefore omitted in the curves.

8 Conclusion

In this paper, we have devised a novel evaluation technique for disjunctive queries, i.e., queries whose selection predicate contains at least one **or** ($\vee$)-connective.

The evaluation technique is based on new selection and join operators that generate two output streams: the *true*-stream with tuples satisfying the corresponding selection- or join-predicate and the *false*-stream with tuples not satisfying the predicate. Each of these streams can then be further processed individually. It thereby becomes possible to "bypass" certain costly or less selective predicates if the "fate" of the corresponding stream can be determined without considering the particular predicate.

The ideas of bypass evaluation have been presented in two prior conference papers: [1] introduced bypass selection and [2] contained the bypass join processing.

In this paper, we have covered the necessary issues to incorporate bypass evaluation in a "real" system:

- We defined the underlying algebra,
- The equations for generating bypass evaluation plans by the query optimizer were given and the OPT and the FIX approaches for searching optimal and suboptimal bypass plans were outlined,
- Two alternative ways to deal with null values within our query evaluation technique were devised,
- The incorporation of the bypass operators into an iterator-based query execution engine was described, and
- A set of benchmark results comparing bypass plans against conventional query execution plans based on a CNF or DNF-query predicate was presented.

The quantitative evaluation proved that bypass plans are superior to conventional plans. In particular, the CNF-based evaluation plans are often orders of magnitude costlier than bypass plans. The superiority of bypass plans over DNF-based plans was less drastic; however, the reader should keep in mind that the DNF-based processing—as used in this comparison—is not possible for queries that have to preserve duplicates according to the SQL semantics. On the other hand, the bypass plans do preserve SQL semantics.

Appendix A

Optimization Algorithm: A More Technical Look

The following algorithm is a simplification of the real implementation in that it does not consider operations besides join and selection. In the real implementation care has to be taken in order to introduce χs and other operators.

```
1.  while there exists a bundle with a control function
    not equal to true
2.  select such a bundle with control function $g$
3.  choose conditions $C$ occurring in $g$ that are to be
    considered next. For each such condition $C$ do
    a.  let $g^+ = g[C/true], g^- = g[C/false]$
    b.  if $C$ is a selection condition apply (1):
        i.   if $g^- = false$, then add a regular selection
             $\sigma_C$ to the bundle and let $g^+$ be its new
             control function
        ii.  else create two new bundles by adding the
             bypass selections $\sigma_C^+$ and $\sigma_C^-$ and replace the
             old bundle by the two new bundles with
             control functions $g^+$ and $g^-$, respectively
    c.  if $C$ is a join condition apply (2):
        i.   search for the partner streams of $C$ in the
             bundle
        ii.  if $g^- = false$, then add a regular join $\bowtie_C$
             connecting the partner streams to the
             bundle and let $g^+$ be its new control
             function
        iii. else create two new bundles by adding the
             bypass joins $\bowtie_C^+$ and $\bowtie_C^-$ to the original
             bundle and replace the old bundle by the
             two new bundles with control functions $g^+$
             and $g^-$, respectively
        iv.  if there is a bundle with only one argument,
             eliminate it
```

Step 3 is subject to several strategies concerning the selection of the next predicate. These strategies are discussed in Section 4.3.

Within the implementation, we represent each algebraic operator by an instance of a class. For bypassing, the following classes are used:

- *Bundle* is an operator used in intermediate optimization steps and has an attribute g for the control function
- *BYPUnion*, *BYPSelect*, and *BYPJoin* are operator nodes for merge union, bypass selection and bypass join, respectively

A.1 Example

Using this algebra, we illustrate the algorithm with a small example. Two relations or extents x and y have to be joined with a join predicate j which is conjunctively connected with a disjunct of two simple selections $x.a$ and $y.b$. The first step translates the query into a simple plan featuring the following characteristics (neglecting projections):

1. The top algebraic operator is always a *BYPUnion*,
2. The second level operator is always a *Bundle*,
3. The arguments of the *Bundle* are the relations or extents to be joined and the control function.

Note that this translation is—except for the top-level *BYPUnion* operator—not much different from the standard translation of SQL queries. The outcome of the translation is shown in Fig. 23a. The control function of the bundle is $j \wedge (x.a \vee y.b)$. According to the algorithm, we have to select a subset of the applicable predicates. Let us choose j and $x.a$. Since $j \wedge (x.a \vee y.b)[j/false]$ results in *false*, a regular

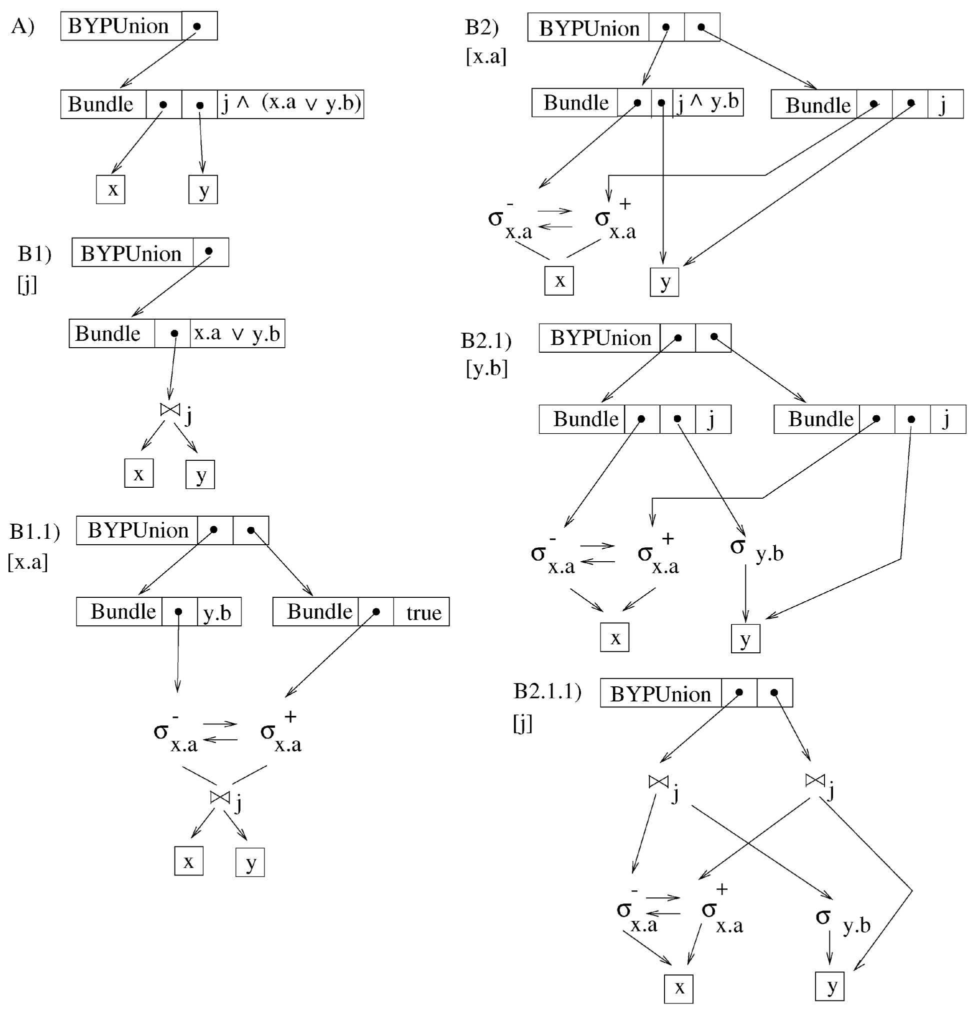


Fig. 23. Example application of the algorithm.

join is added to the bundle (Plan B1). For $x.a$, we have to add a bypass selection. This results in replacing the original bundle by two new bundles, one with control function j and one with control function $j \wedge y.b$ (Plan B2). The same thing happens to Plan B1 when chosing the predicate $x.a$. A bypass selection has to be added resulting in Plan B1.1. To Plan B2, we next choose to add condition $y.b$, yielding Plan B2.1. Then, we have to add join operators. Since both bundles have j as their control function, a regular join suffices for both bundles. Introducing the join operator leaves the bundles with one argument only. Hence, the plan can be simplified to the one shown in B2.1.1.

Appendix B
Details of Our Benchmark

Here are some more details of our benchmarks in Section 7.4. Recall that we have three abstract object extensions E_1, E_2, and E_3, one restriction C_i for each extension, and one join J_{ij} for each pair of extensions. All conditions depend on χ-operations—one for a restriction and two for a join. Thus, we obtain the following dependencies between scanning the objects of the extensions, performing χ-operations, and evaluating conditions:

$$\begin{array}{ll} read(E_i) \rightarrow \chi_i \rightarrow \sigma_{C_i} & (1 \leq i \leq 3) \\ read(E_1) \rightarrow \chi_4 \rightarrow \bowtie_{J_{12}} \leftarrow \chi_5 \leftarrow read(E_2) & \\ read(E_2) \rightarrow \chi_6 \rightarrow \bowtie_{J_{23}} \leftarrow \chi_7 \leftarrow read(E_3) & \\ read(E_1) \rightarrow \chi_8 \rightarrow \bowtie_{J_{13}} \leftarrow \chi_9 \leftarrow read(E_3) & \end{array}$$

Nine χ-operations are necessary. For defining these operations independently of each other, we need at least the following schema:

$$\begin{aligned} type(E_1) &= [a_1 : T_1, a_4 : T_4, a_8 : T_8] \\ type(E_2) &= [a_2 : T_2, a_5 : T_5, a_6 : T_6] \\ type(E_3) &= [a_3 : T_3, a_7 : T_7, a_9 : T_9], \end{aligned}$$

where T_i is a sort or object type ($i = 1, \ldots, 9$). In order to be able to define the conditions, we need at least one function f_i defined on each T_i. The signature of f_i is as follows:

$$f_i : T_i \rightarrow int$$

If T_i is a sort, the function f_i may only operate in main memory and, otherwise (T_i is an object type), an OID may be dereferenced by f_i, i.e., it is a usual dot-operation.

Summarizing, the following operations are defined:

$$
\begin{aligned}
read_{e_1,a_1,a_4,a_8}(E_1) &= obj(E_1)\\
read_{e_2,a_2,a_5,a_6}(E_2) &= obj(E_2)\\
read_{e_3,a_3,a_7,a_9}(E_3) &= obj(E_3)\\
\chi_i &= \chi_{t_i:f_i(a_i)} \quad (i=1,\ldots,9)\\
\sigma_{C_i} &= \sigma_{t_i\phi_i c_i} \quad (i=1,\ldots,3)
\end{aligned}
$$

$$\bowtie_{J_{12}}=\bowtie_{t_4\phi_4 t_5} \qquad \bowtie_{J_{23}}=\bowtie_{t_6\phi_6 t_7} \qquad \bowtie_{J_{13}}=\bowtie_{t_8\phi_8 t_9}$$

with $\phi_i \in \{=, \neq, <, <=, \ldots\}$ and c_i a constant of type *int*.

Now, we can start generating Boolean functions. We discuss queries derived from the generic forms:

$$
\begin{aligned}
&bool_1 = C_1 \vee (C_2 \wedge C_3 \wedge J_{12} \wedge J_{23})\\
&bool_2 = (C_1 \wedge J_{13} \wedge C_3) \vee (J_{12} \wedge C_2 \wedge J_{23})\\
&bool_3 = C_1 \vee (J_{12} \wedge (C_2 \vee (J_{23} \wedge C_3)))\\
&bool_4 = C_1 \vee (J_{13} \wedge C_3) \vee (J_{12} \wedge C_2 \wedge J_{23})\\
&bool_5 = C_1 \vee (J_{12} \wedge (C_2 \vee (J_{23} \wedge (C_3 \vee C_{13})))).
\end{aligned}
$$

All functions (queries) can be evaluated without Cartesian products, which is an indication for realistic queries. For the functions $bool_1$ and $bool_3$, we omit the join condition J_{13}.

We carry out two kinds of benchmarks: main memory operations and I/O operations.

1. Main memory: The function f_i is a main memory operation. The specification is as follows:

$$
\begin{aligned}
type(T_i) &= int\\
f_i(a_i) &= \text{for } (j=1; j<a_i; j++);\\
&\quad // \; return\ value\ is\ a_i\\
\phi_i &= \begin{cases} '<' & \text{for } 1 \le i \le 3\\ '=' & \text{for } 4 \le i \le 9 \end{cases}\\
c_i &= variable.
\end{aligned}
$$

   The *for* loop in f_i implements a random delay by means of a_i iterations in the loop. As mentioned before, a_i is distributed uniformly in the range $0 \ldots 9999$.

2. Dereferencing: The function f_i causes additional page faults. The functions f_i operate on different types:

$$
\begin{aligned}
type(T_i) &= \begin{cases} int & for\ 1 \le i \le 3\\ [a_i' : int] & for\ 4 \le i < j \le 9 \end{cases}\\
&\quad and\ type(T_i) \neq type(T_j)\ for\ 4 \le i < j \le 9\\
f_i(a_i) &= \begin{cases} a_i & for\ 1 \le i \le 3\\ a_i.a_i' & for\ 4 \le i < j \le 9 \end{cases}\\
\phi_i &= \begin{cases} '<' & for\ 1 \le i \le 3\\ '=' & for\ 4 \le i \le 9 \end{cases}\\
c_i &= variable.
\end{aligned}
$$

Both variants are printed side-by-side in Figs. 18, 19, 20, 21, and 22.

Acknowledgments

We thank the anonymous referees for several suggestions to improve the paper. This work was supported by the German Research Council under contracts DFG Ke 401/6-2 and Ke 401/7-1. Excerpts of this work appeared in [1] and [2].

References

[1] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn, "Optimizing Disjunctive Queries with Expensive Predicates," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 336-347, May 1994.

[2] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper, "Bypassing Joins in Disjunctive Queries," *Proc. Conf. Very Large Data Bases (VLDB)*, pp. 228-238, Sept. 1995.

[3] *Query Processing for Advanced Database Systems*, J.C. Freytag, D. Maier, G. Vossen, eds. San Mateo, Calif.: Morgan Kaufmann, 1993.

[4] J.C. Freytag, "A Rule-Based View of Query Optimization," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 173-180, May 1987.

[5] G. Graefe and D. DeWitt, "The EXODUS Optimizer Generator," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 160-172, May 1987.

[6] G. Lohman, "Grammar-Like Functional Rules for Representing Query Optimization Alternatives," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 18-27, May 1988.

[7] M.T. Özsu and D.D. Straube, "Queries and Query Processing in Object-Oriented Database Systems," *ACM Trans. Office Information Systems*, vol. 8, pp. 387-430, Oct. 1990.

[8] A. Kemper and G. Moerkotte, "Advanced Query Processing in Object Bases Using Access Support Relations," *Proc. Conf. Very Large Data Bases (VLDB)*, pp. 290-301, Aug. 1990.

[9] S. Cluet and C. Delobel, "A General Framework for the Optimization of Object-Oriented Queries," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 383-392. June 1992.

[10] D.S. Batory, "Extensible Cost Models and Query Optimization in GENESIS," *IEEE Database Eng.*, vol. 9, Dec. 1986.

[11] J.A. Blakeley, W.J. McKenna, and G. Graefe, "Experiences Building the Open OODB Query Optimizer," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 287-295, May 1993.

[12] G. Graefe and W. McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search," *Proc. IEEE Conf. Data Eng.*, pp. 209-218, Apr. 1993.

[13] L. Haas, J.C. Freytag, G. Lohman, and H. Pirahesh, "Extensible Query Processing in Starburst," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 377-388, May 1989.

[14] L. Becker and R.H. Güting, "Rule-Based Optimization and Query Processing in an Extensible Geometric Database System," *ACM Trans. Database Systems*, vol. 17, pp. 247-303, June 1992.

[15] G. Mitchell, U. Dayal, and S.B. Zdonik, "Control of an Extensible Query Optimizer: A Planning-Based Approach," *Proc. Conf. Very Large Databases (VLDB)*, pp. 517-528, Aug. 1993.

[16] A. Kemper, G. Moerkotte, and K. Peithner, "A Blackboard Architecture for Query Optimization in Object Bases," *Proc. Conf. Very Large Databases (VLDB)*, pp. 543-554, Aug. 1993.

[17] A.Y. Levy, I.S. Mumick, and Y. Sagiv, "Query Optimization by Predicate Move-Around," *Proc. Conf. Very Large Data Bases (VLDB)*, pp. 96-107, Sept. 1994.

[18] J.M. Hellerstein and M. Stonebraker, "Predicate Migration: Optimizing Queries with Expensive Predicates," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 267-276, May 1993.

[19] C. Monma and J. Sidney, "Sequencing with Series-Parallel Precedence Constraints," *Math. Operations Research*, vol. 4, pp. 215-224, 1979.

[20] S. Chaudhuri and K. Shim, "Optimization of Queries with User-Defined Predicates," *Proc. Conf. Very Large Data Bases (VLDB)*, pp. 87-98, Sept. 1996.

[21] M. Muralikrishna, "Optimization of Multiple-Disjunct Queries in a Relational Database System," Technical Report no. 750, Univ. of Wisconsin-Madison, Feb. 1988.

[22] F. Bry, "Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 193-204, May 1989.

[23] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 23-34, May 1979.

[24] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. Database Systems,* vol. 6, Dec. 1981.

[25] L. Kerschberg, P.D. Ting, and S.B. Yao, "Query Optimization in a Star Computer Network," *ACM Trans. Database Systems,* vol. 7, pp. 678-711, Dec. 1982.

[26] M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys,* vol. 16, pp. 111-152, June 1984.

[27] M. Steinbrunn, "Heuristic and Randomised Optimisation Techniques in Object-Oriented Database Systems," Ringstraße 32, 53757 St. Augustin, Germany: infix-Verlag, dissertation, Universität Passau, 1996.

[28] K. Peithner, "Optimierung deklarativer Anfragen in Objektbanken," PhD thesis, Ringstr. 32, 53757 Sankt Augustin, Germany, dissertation, Universität Passau, 1996.

[29] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, *The Object Database Standard: ODMG 2. 0.* San Mateo, Calif.: Morgan Kaufmann, 1997.

[30] J.M. Hellerstein and J.F. Naughton, "Query Execution Strategies for Caching Expensive Methods," *Proc. ACM SIGMOD Conf. Management of Data,* pp. 423-434, June 1996.

[31] D. Stemple and T. Sheard, "A Recursive Base for Database Programming Primitives," *Proc. Kiev East/West Workshop Next Generation Database Technology,* Apr. 1991.

[32] M.Z. Hanani, "An Optimal Evaluation of Boolean Expressions in an Online Query System," *Comm. ACM,* vol. 20, pp. 344-347, May 1977.

[33] G. von Bültzingsloewen, *SQL-Anfragen-Optimierung für parallele Bearbeitung.* New York, Berlin, etc.: FZI-Berichte Informatik, Springer-Verlag, 1991.

[34] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys,* vol. 25, pp. 73-170, June 1993.

[35] B. Stroustrup, *The C++ Programming Language,* second ed. Reading, Mass.: Addison-Wesley, 1992.

[36] C.A. Gerlhof, "Optimierung von Speicherzugriffskosten in Objektbanken: Clustering und Prefetching," PhD thesis, Universität Passau, Fakultät für Mathematik und Informatik, D-94030 Passau, dissertation, Universität Passau, 1996.

Jens Claussen received his diploma degree in computer science in 1995 from the University of Passau, Germany, and received his PhD from the the University of Passau in 2000. He now works for SAP.

Alfons Kemper received his MSc and PhD degrees in computer science in 1981 and 1984, respectively, from the University of Southern California (USC), Los Angeles. He is a full professor at the University of Passau, Germany. His research interests include query optimization and evaluation techniques, object-oriented database systems, in particular distributed systems, and data warehousing.

Guido Moerkotte received his doctorate in computer science in 1989 from the University of Karlsruhe. Since 1996, he has been a full professor at the University of Mannheim, Germany. His research interests include query optimization and evaluation techniques, object-oriented database systems, and data warehousing, in particular benchmarks.

Klaus Peithner received his diploma degree in computer science in 1991 from the University of Karlsruhe and received his PhD degree in 1995 from the University of Passau. After 2 1/2 years working for Deutsche Bank AG, he recently joined MicroStrategy Inc. (www.strategy.com)—a leading worldwide provider of enterprise decision support system software and related services.

Michael Steinbrunn received his diploma degree in computer science in 1991 from the University of Karlsruhe and received his PhD degree in 1995 from the University of Passau. He currently works for Sun Microsystems GmbH, Germany.