# Optimization and Evaluation of Disjunctive Queries* **

August 14, 1996

*Jens Claussen*† *Alfons Kemper*† *Guido Moerkotte*‡
*Klaus Peithner*† *Michael Steinbrunn*†

†Universität Passau
Lehrstuhl für Informatik
94030 Passau, Germany
⟨lastname⟩@*db.fmi.uni-passau.de*

‡Universität Mannheim
Lehrstuhl für Praktische Informatik III
68131 Mannheim, Germany
*moer@pi3.informatik.uni-mannheim.de*

**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 papers 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.

*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 [KMPS94] and [SPMK95].

# Contents

# 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 [FMV93]. [Fre87, GD87, Loh88] made rule-based query optimization popular, which was later adopted in the object-oriented context, as e.g., [ÖS90, KM90, CD92]. Many researchers have worked on optimizer architectures that facilitate flexibility: [Bat86, GD87, BMG93, GM93] are proposals for optimizer generators; [HFLP89, BG92] describe extensible optimizers in the extended relational context; [MDZ93, KMP93] 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. [LMS94] introduces a technique for moving predicates across query components, where a component constitutes, for instance, a view definition. [HS93] deals with the optimal 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. [HS93]'s 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 [MS79]. This approach yields the optimal evaluation sequence for *conjunctive* selection predicates.

It is striking that in all these works the optimization of disjunctive query predicates tends to be neglected. [Bry89] and [Mur88] are the only works—to the authors' knowledge—who dealt with disjunctions in particular. 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 [SAC+79]) or the predicate is subdivided into several conjunctive streams that are optimized separately (disjunctive normal form, e.g., [BGW+81, KTY82, ÖS90, JK84]).

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 that fulfil 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. Thereby, particularly costly and/or less selective predicates may 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:

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

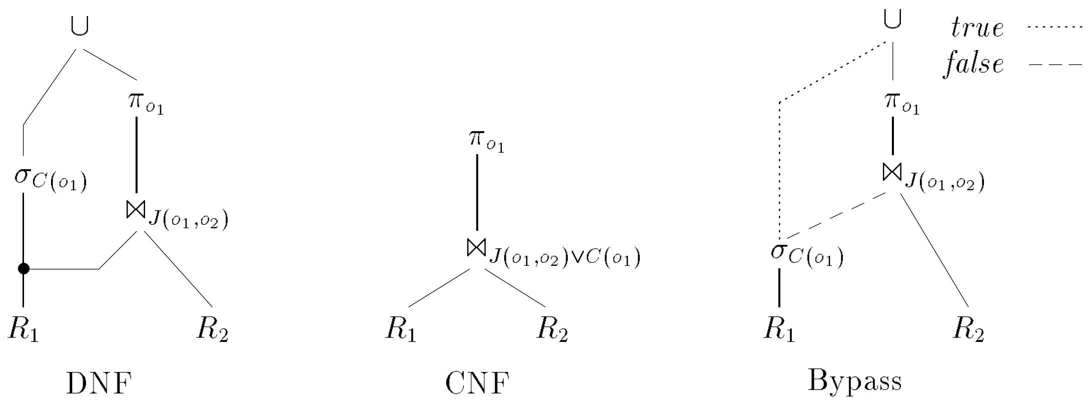



Figure 1: Evaluation Plans for $\pi_{o_1}(\sigma_{C(o_1)\vee J(o_1,o_2)}(R_1 \times R_2))$

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 to both the DNF- and CNF-based evaluation plan. Figure 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. Since Hellerstein speculates in [Hel94] that a well-working heuristic solution for placing selections in the presence of join operations might be hard to obtain (or even impossible), we propose two "building-block algorithms" which are comparable to algorithms based on dynamic programming [SAC+79]. 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 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 relies on 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*.

However, 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. This puts, in principle, the reader into the position to incorporate the bypass technique into existing optimizers and execution engines and emphasizes the technique's practical feasibility.

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 and sketches the building block approach for generating and optimizing query evaluation plans employing bypass processing. 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 techniuque 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 outline a correctness proof for the generation of bypass plans and 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 Figure 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 [Cat94] rendering of this query is

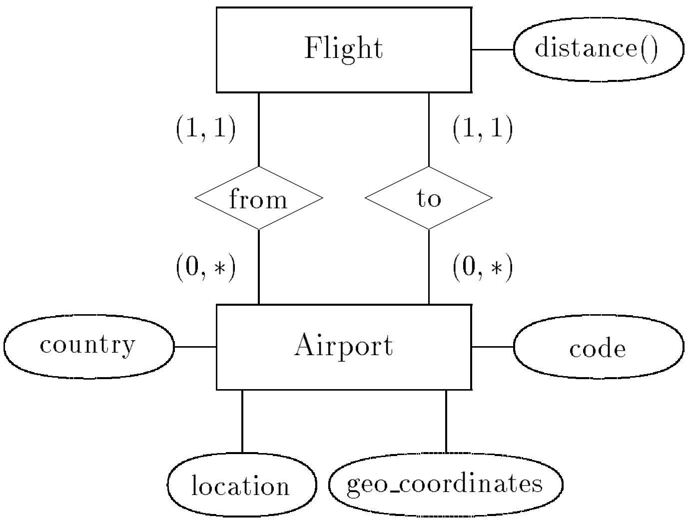



Figure 2: ER-Schema of the Flight Reservation System

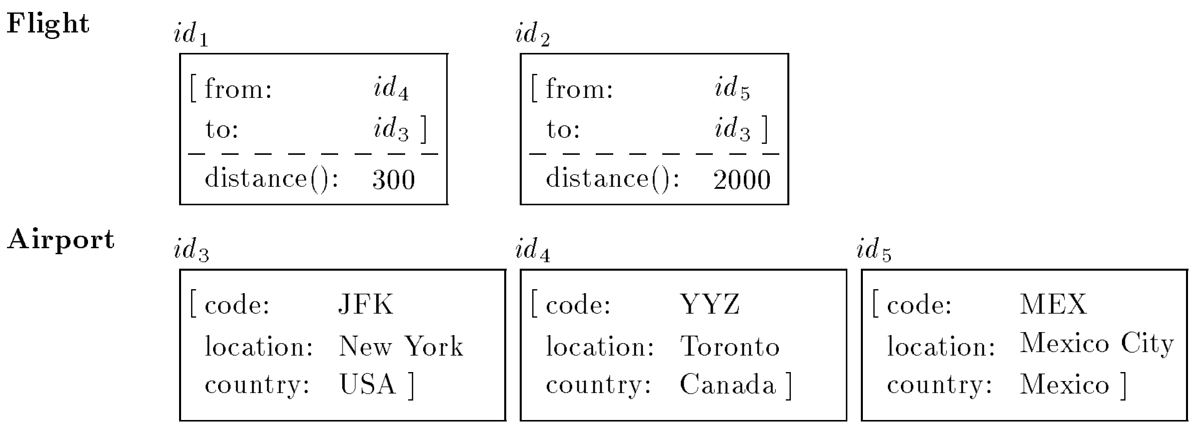



Figure 3: Sample object base

**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 Figure 2 is depicted in Figure 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 Figure 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 1000 objects of type *Airport*

| 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 |

Table 1: Abbreviations and Probabilities for "Immigration Airports"

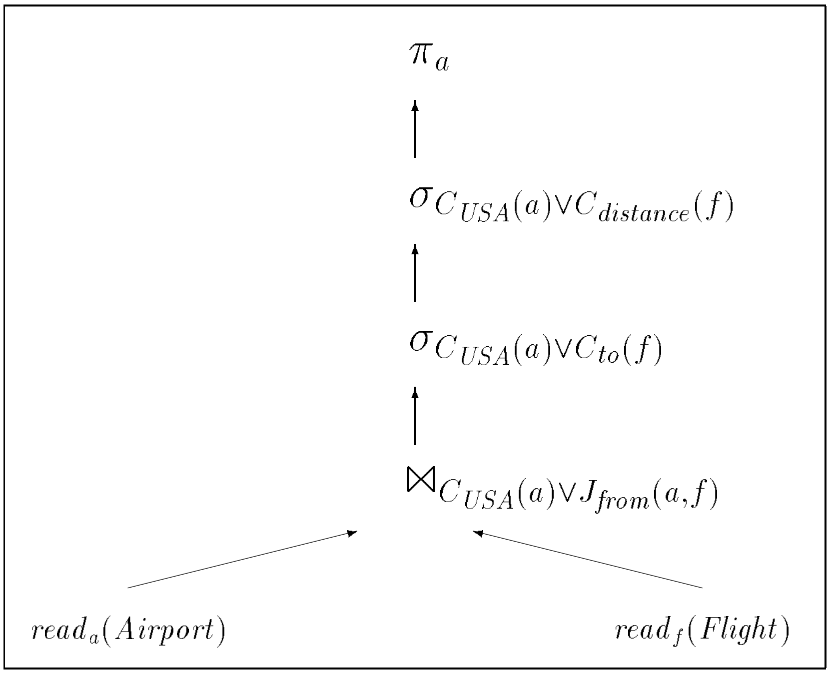



Figure 4: CNF-based Evaluation Plan for "Immigration Airports"
Relative Cost: Factor 150

and 29000 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 base for query evaluation plans. However, both these approaches suffer from certain drawbacks, particularly when dealing with disjunctive queries. In Figure 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

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

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 indispensable if competitive performance is desired. And third, we cannot do without the (duplicates eliminating) projection $\pi_a$ 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 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

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

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 united in a final union operation that represents the DNF's *or*-operations. Figure 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 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 left-hand side of Figure 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 right-hand side with *Flight* objects, which is filtered by the two selection 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 can only be used if duplicate preservation is not required.

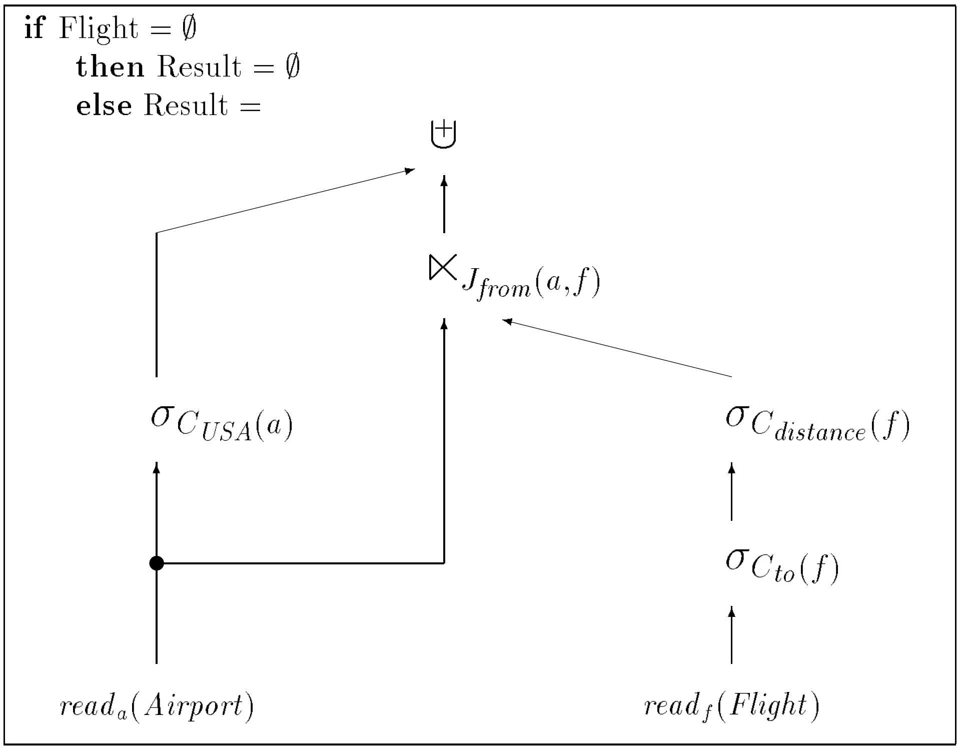



Figure 5: DNF-based Evaluation Plan for "Immigration Airports"
Relative Cost: 240%

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% 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. But before we do so, let us have a look at the way the major commercial relational database systems deal with disjunctions.

## 2.3 Commercial Database Systems

In the previous section, we have seen that both CNF- and DNF-based evaluation plans inhere drawbacks for disjunctive queries. In this section, we examine how commercial relational database systems handle disjunctive queries. For this purpose, we have used the *explain plan* facility of some popular systems to examine their view the generated evaluation plans. In particular, we have analyzed the following systems' optimizers:[1]

- Oracle 7.3.2
- DB2 2.3
- Informix 5.04.UC1
- Sybase 10.0.2

The query is a slightly simplified standard SQL version of the example OQL query (cf. Section 2.1):

**select** **distinct** a.code
**from** Airport a, Flight f
**where** a.country = 'USA' **or**
(a.code = f.from **and** f.distance_val $\leq$ 400)

One of the conditions, namely "f.to.country = 'USA'," has been removed, because a path like "f.to.country" would not be evaluated efficiently by any of the relational systems; furthermore, the computed attribute "distance()" has been replaced by a regular attribute named "distance_val."

### 2.3.1 Evaluation without Indexes

The scenario without indexes is what we have assumed for the cost estimates in the previous section. Optimization resulted in the same simple evaluation plan for all examined systems, shown in Figure 6. The plan consists of a simple nested loops join on the two involved relations. The plan is similar to the CNF-plan with the only difference that the query predicate has not been split into boolean factors. Consequently, the complete join condition must be the predicate stated in the *where* clause of the original query.

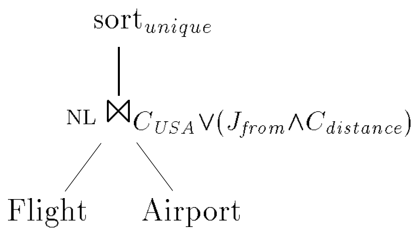
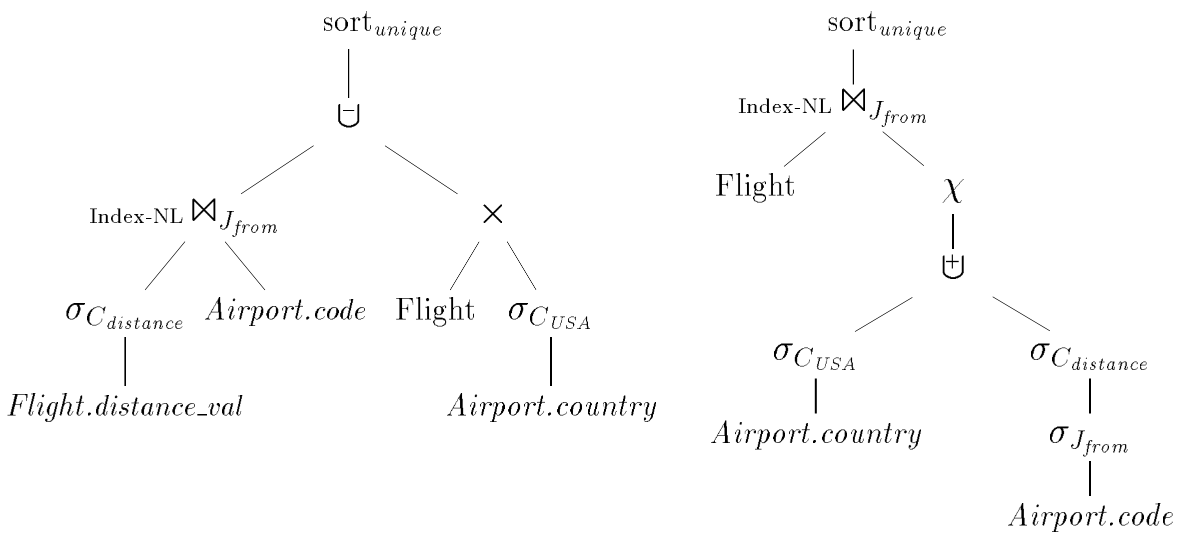



Figure 6: Evaluation Plan without Indexes

[1]Note that the version numbers do not always constitute the most recent release; these were the systems we had access to.

### 2.3.2 Evaluation with Index Support




(a) Oracle (b) DB2

Figure 7: Evaluation Plans exploiting Indexes

In a second step, we fed the same query into the systems again, this time, however, with indexes on all relevant attributes, namely *a.code*, *a.country*, *f.from* and *f.distance_val*. The results were very different this time. Informix and Sybase did not take the indexes into account at all, i.e., they produced the same plan as in Figure 6 again. Oracle and DB2 produced more sophisticated plans. Oracle's evaluation plan (Figure 7a) is now DNF-based and resembles the one in Figure 5. On the left-hand side, an index on the attribute *distance_val* of relation *Flight* expedites the selection operation $\sigma_{C_{distance}}$.[2] The join operation is carried out as an index nested loops join where the index on attribute *code* of the inner relation *Airport* is used for finding matching tuples. On the right-hand side, an index on attribute *country* of relation *Airport* supports the selection operation in the same manner as the corresponding operation on the left-hand side. In addition, a cartesian product with relation *Flight* is required in order to carry out the final union operation (without elimination of duplicates). The only major difference between the evaluation plans in Figures 5 and 7a is that Oracle's plan does not use a semijoin for condition $J_{from}$.

DB2 has produced an even more sophisticated plan, as shown in Figure 7b. The central operator is an index nested loops join. It scans the *Flight* relation without indexes, however uses the index on attribute *code* of *Airport* to find the row identifiers (RID) for *Airport* records matching $J_{from}$ (denoted by $\sigma_{J_{from}}$ on the right-most stream). After that, the predicate $C_{distance}$ is checked for the current *Flight* tuple. The resulting RID set is sorted and merged with the set of RIDs retrieved from the index matching $C_{USA}$ (This merge stands for the *or* operator.). Then the *Airport* records denoted by the RIDs are

[2] In the evaluation plans in Figure 7, italic relation names denote access through an index structure, while upright names denote normal table access.

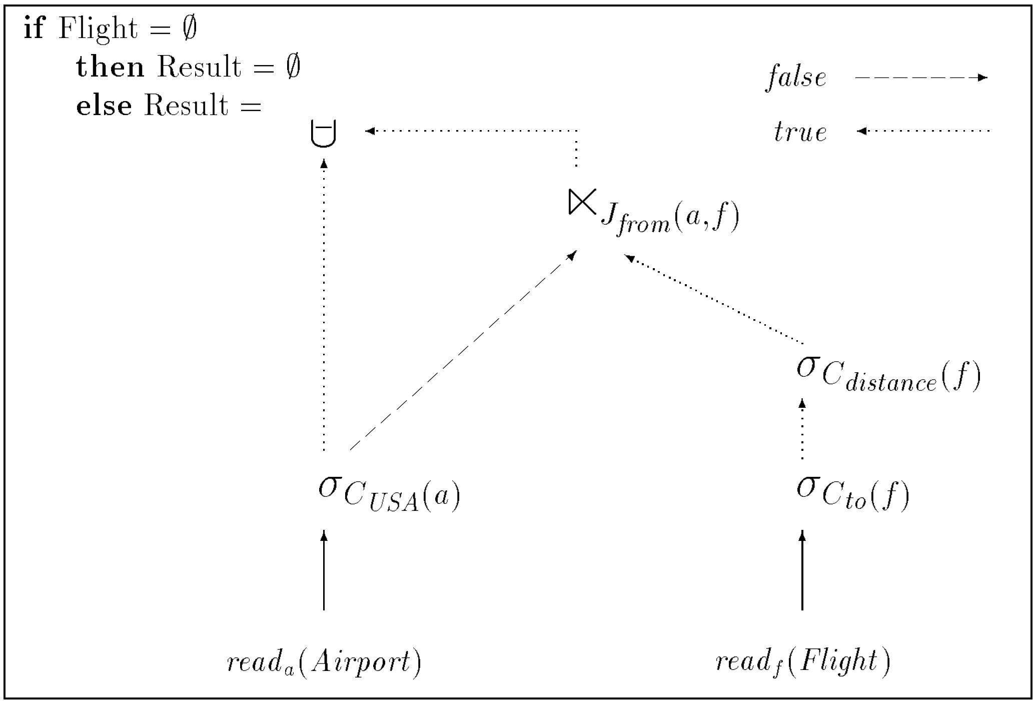



Figure 8: Bypass Evaluation Plan for "Immigration Airports"
Relative Cost: 100% (Optimum)

retrieved by an expand operator (cf. Section 3) to form the result.

### 2.4 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 sub-optimal 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 conditions, not merely of Boolean factors), and it avoids redundant computations entirely. Both achievements are made possible by introducing a new class of operators, so-called *bypass operators*, namely as *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" (Figure 8). This evaluation plan does not merely constitute the lowest-cost bypass plan for the sample query, but the optimal evaluation plan, regardless of the underlying construction method.

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 left-hand side of Figure 8, *Airport* objects are tested whether they satisfy the condition $C_{USA}$, similar to the DNF-based evaluation plan (cf. Figure 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 Figure 8, these two different streams are marked with dotted lines (objects that satisfy the condition) and broken lines (objects that do not satisfy the condition), respectively.

On the right-hand side of Figure 8, objects of type *Flight* must satisfy the two conditions $C_{to}$ and $C_{distance}$, before the semijoin $J_{from}$ is carried out (similar to Figure 5). The last operation in the bypass evaluation plan is the union node $\uplus$, 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 $\uplus$.

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 Figure 8), but also bypasses the disadvantages of the two traditional techniques: the bypass technique does not only allow fine-grained allocation of selection or join predicates to an extent not available in CNF-based plans, but also 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 (Figure 4) is 150 times as expensive, and the best DNF-based plan (Figure 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. [KMPS94]) 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. Third, the benefits of composing query evaluation plans in a so-called *building block approach* are illustrated by means of an example.

## 3.1 Basic (logical) Algebra

For the purpose of this paper, a "slim" object-oriented algebra is sufficient. It consists of the well-known relational operators selection '$\sigma$', projection '$\pi$', 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* '$\chi$'—comparable with [BMG93]'s materialize operator or the map function [SS91]—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):

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

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

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

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 $\sigma^+/\sigma^-$, $\bowtie^+/\bowtie^-$, 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:

$$
\begin{aligned}
\sigma^+_C(S) &:= \{\, s \mid s \in S \wedge C(s) \,\} \\
\sigma^-_C(S) &:= S - \sigma^+_C(S) \stackrel{*}{=} \{\, s \mid s \in S \wedge \neg C(s) \,\} \\
S \bowtie^+_C T &:= \{\, s \circ t \mid s \in S \wedge t \in T \wedge C(s,t) \,\} \\
S \bowtie^-_C T &:= (S \times T) - (S \bowtie^+_C T) \stackrel{*}{=} \{\, s \circ t \mid s \in S \wedge t \in T \wedge \neg C(s,t) \,\} \\
S \ltimes^+_C T &:= \{\, s \mid s \in S \wedge \exists t \in T \colon C(s,t) \,\} \\
S \ltimes^-_C T &:= S - (S \ltimes^+_C T) \stackrel{*}{=} \{\, s \mid s \in S \wedge \neg \exists t \in T \colon C(s,t) \,\}
\end{aligned}
$$

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 re-united 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 $\theta$. For instance, '$=$' becomes '$\neq$'. However, this transformation will be contra-intuitive, if *null*-values are allowed. For example, both conditions $a.country =$ *"USA"* and $a.country \neq$ *"USA"* yield *false* whenever the attribute *country* of an airport object $a$ is not defined, i.e., is *null*. In order to avoid this misleading effect and to exploit the benefits of a two-valued logic, we take advantage of an idea proposed in [vB91]. There, an index '$^+$' which is attached to a comparison operator of a condition $C$ indicates that $C$ evaluates to *true* in the case of a *null*-value. Hence, the negation $\neg C$ of a condition $C$ is performed straightforward—$\theta$ becomes $\not\theta^+$, $\theta^+$ becomes $\not\theta$, and vice versa. Our definitions assume this kind of condition negation as indicated by the set difference semantics in the definitions above.

The differentiation between "positive" and "negative" tuples performed by a bypass operation allows some tuples to *bypass* expensive operations. Then, the two tuple streams are re-united by a merge operation. A regular selection $\sigma$, a regular join $\bowtie$, or a regular semijoin $\ltimes$ can be taken as special cases of bypass operators where the complementary output is discarded—for instance, the two selections on the left-hand side of Figure 8 could be interpreted that way. Hence, since the "positive" side of a bypass operator is nothing else but a regular one-output selection or join/semijoin operation, the integration into the relational algebra is straightforward.

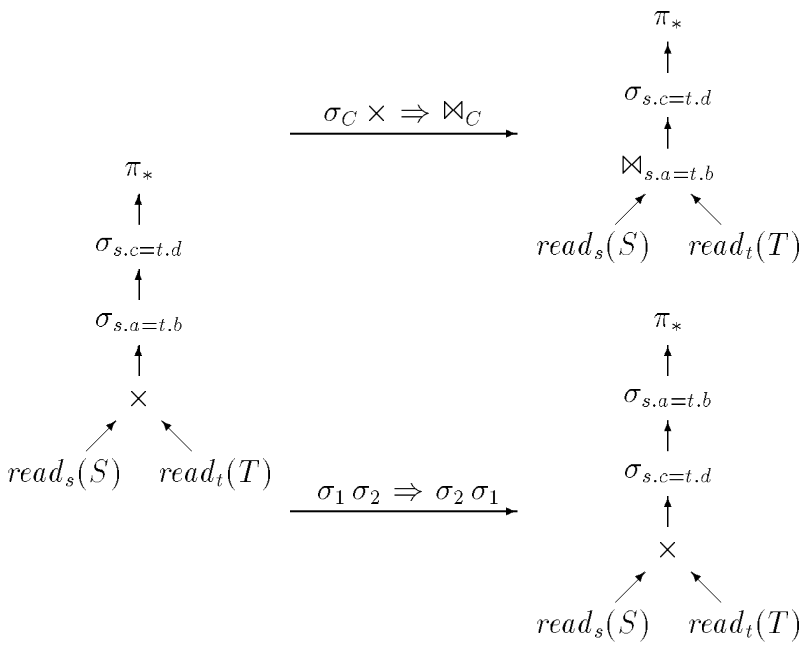



Figure 9: Exemplary Application of Two Transformation Rules

### 3.3 Building Block Approach

The optimization in our prototype system is a composition process—called *building block approach*—as opposed to a sequence of rule-based transformations. In order to capture the differences between those approaches, let us optimize the following abstract query:

$$Q \;\equiv\; \textbf{select } * \quad \textbf{from } \text{S } \textbf{as } \text{s, T } \textbf{as } \text{t} \quad \textbf{where } \text{s.a} = \text{t.b } \textbf{and } \text{s.c} = \text{t.d}$$

The query $Q$ simply applies two join conditions on the extensions $S$ and $T$.

The transformation-based optimization process usually starts from a so-called canonical "$\pi\,\sigma\,\times$"-expression which is derived from the user's query. For the example query $Q$, the left-hand side of Figure 9 depicts such an initial plan. This figure further shows the application of two transformation rules. The first (upper) rule "$\sigma_C \times \Rightarrow \bowtie_C$" introduces a join operation, and the second (lower) rule "$\sigma_1\,\sigma_2 \Rightarrow \sigma_2\,\sigma_1$" swaps two selections. In order to apply these rules, the following execution cycle is performed:

1. The set of rules with left-hand sides matching the current evaluation plan is determined.

2. If the set of matching rules is not empty, the evaluation plan is copied in order to avoid possible side effects.

3. The transformation is performed, i.e., the pattern is substituted by the right-hand side of the rule.

4. This transformation invalidates all cost estimates for subsequent processing nodes. Therefore, the cost estimation for the transformed evaluation plan must be carried out anew.

Only Step 3 advances the optimization process. All other steps—"pattern matching" (Step 1), "copying" (Step 2), and "cost re-computing" (Step 4)—constitute extra work that is accepted only for the sake of extensibility and (provable) partial correctness.

The building block approach also starts from a normal form called MCNF (Most Costly Normal Form). However, the MCNF is not a query evaluation plan as above, but a decomposition of the user's query into its algebraic operations constituting the building blocks [KM93, KMP93]. For instance, the query $Q$ would be decomposed into two *read*-operations, two joins, and one projection. We obtain a set

$$\{read_s(S), read_t(T), \bowtie_{s.a=t.b}, \bowtie_{s.c=t.d}, \pi_*\}$$

which is called *future work*.

The construction process takes building block after building block from the future work set, and composes them to query evaluation plans. Since the construction proceeds bottom-up, the *read*-operations are chosen first in order to lay the foundations.

$$B_1 = \{read_s(S), read_t(T)\}$$

$B_1$ consists of two elements which will be joined by the subsequent process. If indexes on $S$ and/or $T$ were present, additional starting points $B_1'$, $B_1''$, etc. could be obtained.

Second, the join operations are chosen yielding two alternatives $B_2$ and $B_3$:

$$\begin{aligned} B_2 &= \{read_s(S) \bowtie_{s.a=t.b} read_t(T)\} \\ B_3 &= \{read_s(S) \bowtie_{s.c=t.d} read_t(T)\} \end{aligned}$$

Note that both alternatives stem from the same expression $B_1$ whose elements remain unchanged, i.e., the *read*-operations are not modified (e.g., for exploiting an index). Therefore, the optimization process does not have to copy, but can take advantage of common subexpressions among the alternatives. Furthermore, the cost estimate for a given subplan must be computed only once and can be re-used for all enhanced plans containing it as a subexpression.

The composition of one evaluation plan is finished as soon as the corresponding future work set is empty. Thus, a query evaluation plan is generated without pattern matching, copying, and re-calculating, but in a constructive and, therefore, rather efficient way.

The building block approach composes query evaluation plans step by step in a bottom-up fashion. In this respect, our optimizing technique is similar to the well-known dynamic programming approach described in [SAC+79], where join orderings were obtained by augmenting basic *read*-operations step by step into complete evaluation plans.

# 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):

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

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 [Han77] 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):

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

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 [Han77]. 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 [Mur88]. 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 building block procedure for the construction of bypass plans. We first outline the general construction algorithm and then introduce some enhancements by means of projection pushing.

### 4.2.1 The Construction Algorithm

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

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

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

$$\begin{aligned} Q &= \sigma_{g|_{C_i=false}}(R_1 \times \cdots \times \sigma^-_{C_i}(R_j) \times \cdots \times R_n) \uplus \\ &\quad\; \sigma_{g|_{C_i=true}}(R_1 \times \cdots \times \sigma^+_{C_i}(R_j) \times \cdots \times R_n) \end{aligned} \tag{1}$$

$$\begin{aligned} Q &= \sigma_{g|_{C_i=false}}(R_1 \times \cdots \times (R_j \bowtie^-_{C_i} R_k) \times \cdots \times R_n) \uplus \\ &\quad\; \sigma_{g|_{C_i=true}}(R_1 \times \cdots \times (R_j \bowtie^+_{C_i} R_k) \times \cdots \times R_n) \end{aligned} \tag{2}$$

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, equation (1) is employed; otherwise, if $C_i$ is a join condition on relations $R_j$ and $R_k$, equation (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 $\sigma_{g(C_1,\ldots,C_m)}(R_1 \times \cdots \times R_n)$. 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, equation (2) can be rewritten as

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

For a query with a Boolean function $g$ the following algorithms start with the initial bundle $\{R_1, \ldots, R_n\}_g$ and apply equations (1) and (2) repeatedly until a set of bundles with control functions $g' = true$ or $g' = false$ is obtained.

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

$$\Big\{ [a] \mid \exists f\colon a \in Airport \wedge f \in Flight \wedge \Big(C_{USA}(a) \vee \Big(J_{from}(a, f) \wedge C_{to}(f) \wedge C_{distance}(f)\Big)\Big) \Big\}$$

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. We choose $C_{USA}(a)$ and the following two bundles are generated:

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

$B_{11}$'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 $B_{12}$:

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

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:

$$\begin{aligned}
B_{11} &= \{\sigma^{+}_{C_{USA}(a)}(read_a(Airport)), read_f(Flight)\}_{true} \\
B_{1211} &= \{\sigma^{-}_{C_{USA}(a)}(read_a(Airport)), \sigma_{C_{distance}(f)}(\sigma_{C_{to}(f)}(read_f(Flight)))\}_{J_{from}(a,f)}
\end{aligned}$$

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

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

$$\begin{aligned}
B_{11} &= \{\sigma^{+}_{C_{USA}(a)}(read_a(Airport)), read_f(Flight)\}_{true} \\
B_{12111} &= \{\sigma^{-}_{C_{USA}(a)}(read_a(Airport)) \bowtie^{+}_{J_{from}(a,f)} \sigma_{C_{distance}(f)}(\sigma_{C_{to}(f)}(read_f(Flight)))\}_{true}
\end{aligned}$$

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 Figure 10. Figure 11 shows the resulting evaluation plan.

### 4.2.2 Pushing Projections

As a final optimization step, we can push projections that eliminate one relation down in the evaluation plan. This may yield significant savings since cross products may be removed completely and joins may be reduced to semijoins. The transformation rules are:

$$\pi_{\mathcal{A}(S)}(S \times T) = S \qquad \text{if} \quad T \neq \emptyset \tag{3}$$

$$\pi_{\mathcal{A}(S)}(S \bowtie^{\pm} T) = S \ltimes^{\pm} T \tag{4}$$

In both cases, it is assumed that the projection removes the attributes of $T$ completely. The equations (3) and (4) hold only for the "true" projection operator (denoted '$\pi$') with set semantics. If duplicates have to be retained transformations (3) and (4) cannot be applied. Note that the equations (1) and (2) do retain duplicates. Therefore, they can be applied no matter whether set or multi-set semantics is specified. For a proof of correctness we refer to the Appendix A.

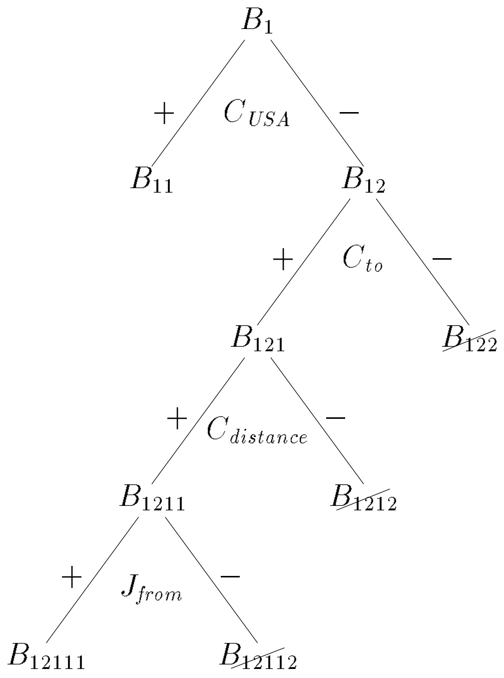



Figure 10: Visualization of the Construction Process

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. Both equations (3) and (4) can be applied once, removing the cross product and substituting $\bowtie_{J_{from}}$ by $\ltimes_{J_{from}}$.[3]

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 Figure 8. The case of an empty *Flight* extension—i.e., the restriction on applying transformation (3)—is treated by the separate *if*-statement in the evaluation plan.

Equation (4) can be applied both for regular joins and for bypass joins. It is feasible to transform the bypass join operator asymmetrically, converting only the *false*-output of the bypass operator pair to a semi-join. This results in the operator combination $\bowtie^{+}/\ltimes^{-}$. The opposite combination $\ltimes^{+}/\bowtie^{-}$ is generally not useful, since it may introduce duplicates into the *false*-stream. !!!!!!!!!!!!!

## 4.3 Generating Alternative Bypass Plans

In principle, the order for introducing the conditions is not fixed. It may sometimes be advantageous to construct evaluation plans where the evaluation orders are not determined

[3]In order to move $\pi$ across $\uplus$, we need the auxiliary equation $\pi_{\mathcal{A}}(S \uplus T) = \pi_{\mathcal{A}}(S) \uplus \pi_{\mathcal{A}}(T)$, where $\mathcal{A}$ denotes any valid attribute set.

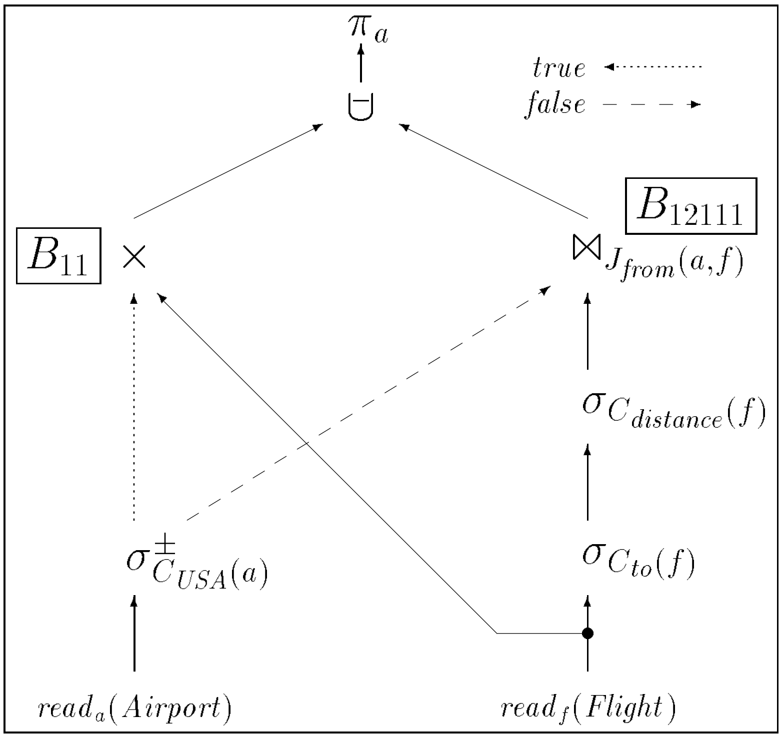



Figure 11: Preliminary Query Evaluation Plan for “Immigration Airports”

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 for the *entire* evaluation plan.

# 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.

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 |

| ¬ | |
|---|---|
| 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 with two-valued logic. As can be derived from the truth tables for *and* and *or* above it does not 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.

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

## 5.1 (Potentially) Maintaining Three Streams

One approach is to convert equations (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:

$$\begin{aligned} Q &= \sigma_{g|_{C_i=false}}(R_1 \times \cdots \times \sigma_{C_i=true}(R_j) \times \cdots \times R_n) \uplus \\ &\quad \sigma_{g|_{C_i=true}}(R_1 \times \cdots \times \sigma_{C_i=false}(R_j) \times \cdots \times R_n) \uplus \qquad (1') \\ &\quad \sigma_{g|_{C_i=unknown}}(R_1 \times \cdots \times \sigma_{C_i=unknown}(R_j) \times \cdots \times R_n) \\ Q &= \sigma_{g|_{C_i=false}}(R_1 \times \cdots \times (R_j \bowtie_{C_i=true} R_k) \times \cdots \times R_n) \uplus \\ &\quad \sigma_{g|_{C_i=true}}(R_1 \times \cdots \times (R_j \bowtie_{C_i=false} R_k) \times \cdots \times R_n) \uplus \qquad (2') \\ &\quad \sigma_{g|_{C_i=unknown}}(R_1 \times \cdots \times (R_j \bowtie_{C_i=unknown} R_k) \times \cdots \times R_n) \end{aligned}$$

Applying the extended rule ($1'$) to our simple example $\sigma_{C \vee \neg C}(R)$ results in three bundles.[4]

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

[4]In the following, bundles whose control function definitely evaluates to *true* are marked with a leading arrow.

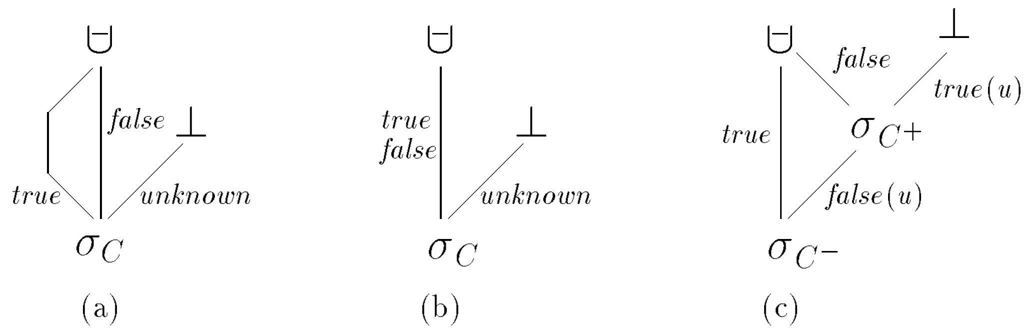



Figure 12: Alternative Evaluation Plans for $\sigma_{C \vee \neg C}(R)$: (a) Three-Valued, (b) Merged Streams, (c) Two-Valued Variant

(a)

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

(b)

| $C$ | $P$ | $\neg C \wedge P$ | $C \wedge P$ |
|---|---|---|---|
| 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** |

(c)

| $C$ | $P$ | $\neg C \vee P$ | $C \vee P$ |
|---|---|---|---|
| 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** |

Table 2: Mapping from Three-Valued Logic to Two-Valued Logic

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

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

for evaluating the query $\sigma_{C \vee \neg C}(R)$ (see Figure 12a). For this example, the two result streams may be collapsed to form a single output stream, as shown in Figure 12b.

## 5.2 Different Polarizations of Condition Evaluation

The second approach is based on an idea proposed by von Bültzingsloewen [vB91]. 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. A condition $C$ without superscript and preceding negations has an implicit negative polarization (i.e., $C^-$), enforcing the standard mapping of *unknown* to *false*. A preceding negation implies a positive polarization. Multiple negations must be removed employing the "rule of double negation" $C = \neg\neg C$ (see Table 2a). Tables 2b and 2c show how the mapping works. Imagine a predicate consisting of a single condition $C$ and the remaining predicate $P$, connected by $\wedge$ or $\vee$. When substituting all possible truth-value combinations for $C$ and $P$, it comes out that the results for $C$=*unknown* are always the same as for $C$=*true* if $C$ is prefixed by a negation (cf. second column of the tables; the matching result groups are highlighted). Consequently, condition $C$ should be *polarized positively* if it is prefixed by a negation. This is denoted by the superscript $^+$ in the condition $C^+$. Similarly, if $C$ is not associated with a negation, $C$=*unknown* always yields the same results as $C$=*false*, thus $C$ can be polarized negatively to $C^-$.

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 $\neg\neg C = C$.
3. (a) Polarize each condition $C$ without a preceding negation as $C^-$.
   (b) Polarize each condition $C$ with a preceding negation to $C^+$.
4. Apply the normal evaluation process as described in Subsection 4.2, however, treating $C^+$ and $C^-$ as different conditions.

## 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 directly to conditions. Since the only negation is already placed correctly, we proceed with 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 Figure 12c. The additional $(u)$ adornment indicates the way 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 solution based on three-valued logic, we get the following bundles:

$$
\begin{aligned}
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)=unknown}(read_r(R))\}_{\neg(unknown \wedge C_2) \vee (unknown \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)=unknown}(\sigma_{C_1(r)=true}(read_r(R)))\}_{\neg(true \wedge unknown) \vee (true \wedge C_3)} \\
B_{31} &= \{\sigma_{C_2(r)=true}(\sigma_{C_1(r)=unknown}(read_r(R)))\}_{\neg(unknown \wedge true) \vee (unknown \wedge C_3) \rightarrow unknown} \\
\rightarrow B_{32} &= \{\sigma_{C_2(r)=false}(\sigma_{C_1(r)=unknown}(read_r(R)))\}_{\neg(unknown \wedge false) \vee (unknown \wedge C_3) \rightarrow true} \\
B_{33} &= \{\sigma_{C_2(r)=unknown}(\sigma_{C_1(r)=unknown}(read_r(R)))\}_{\neg(unknown \wedge unknown) \vee (unknown \wedge C_3) \rightarrow unknown} \\
\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)=unknown}(\sigma_{C_2(r)=true}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge true) \vee (true \wedge unknown) \rightarrow unknown} \\
\rightarrow B_{131} &= \{\sigma_{C_3(r)=true}(\sigma_{C_2(r)=unknown}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge unknown) \vee (true \wedge true) \rightarrow true} \\
B_{132} &= \{\sigma_{C_3(r)=false}(\sigma_{C_2(r)=unknown}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge unknown) \vee (true \wedge false) \rightarrow unknown} \\
B_{133} &= \{\sigma_{C_3(r)=false}(\sigma_{C_2(r)=unknown}(\sigma_{C_1(r)=true}(read_r(R))))\}_{\neg(true \wedge unknown) \vee (true \wedge unknown) \rightarrow unknown} \\
B_{31x} &= \ldots
\end{aligned}
$$

The left-hand side of Figure 13 depicts the corresponding evaluation plan.

2. To make use of the second variant, we first 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:[5]

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

Now, the equations (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 $C_1^-$).

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

Again, the corresponding evaluation plan is shown in Figure 13b.

[5]Simplifications of control functions are indicated by a small subsequent arrow.

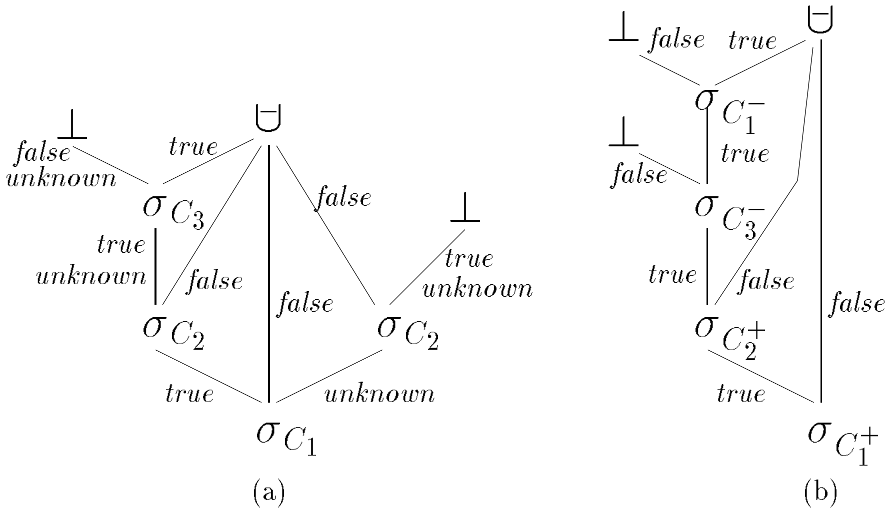



Figure 13: Second Example: $\sigma_{\neg(C_1 \wedge C_2) \vee (C_1 \wedge C_3)}(R)$

### 5.4 Discussion of the Two Approaches

Both presented variants inhere some drawbacks. 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 example. Multiple evaluation of a condition $C$ is only necessary, however, if there are already multiple occurrences of $C$ in the original predicate. 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 negations at all.

# 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, afterwards 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

Before we describe details of our particular implementation, we want to discuss some general alternatives for query evaluation. Apart from the algorithms for implementing single 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, especially parallel and distributed execution, are out of the scope of this paper.

### 6.1.1 Operator Communication

On an abstract level, one can regard all data passed from an operator to others as (virtual) tables, consisting of an (ordered) set of records. In a relational system, operators on the leaves of a query evaluation plan (QEP) either take tables or row identifiers (RIDs, TIDs) from index structures as input. In object-oriented systems, object identifiers (OIDs) are used instead of TIDs. Since these virtual tables may grow to an enormous size, they must either be written to secondary storage or they must be split into chunks, e.g., pages or records. When passing intermediate results on a per-record basis, one can be sure that no I/O is necessary unless the operator implementation explicitly requires to read the complete input before producing output, like sort operations.

While it seems to be advantageous to process queries on a per-record basis, there are some drawbacks, too. First, one has to make sure that passing records and switching control to the next operator is implemented very efficiently, since it is performed very often. Additionally, care should be taken that records are not copied unnecessarily.

Second, even if context switching between operators is very fast, locality may suffer from the alternating activity of several operators. If data is processed in whole tables, there is always only one operator working on them, having exclusive access to all available resources. On the other hand, if several operators are active in turn, they have to share the totally available resources. This problem can be overcome through clever execution planning, e.g. splitting the whole plan into sequentially processed parts. Furthermore, one can decide to pass bulks of records from one operator to the next, e.g., by preserving physical units like pages.

### 6.1.2 Operator Scheduling

If operators do not pass complete tables, 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 does not only facilitate 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.

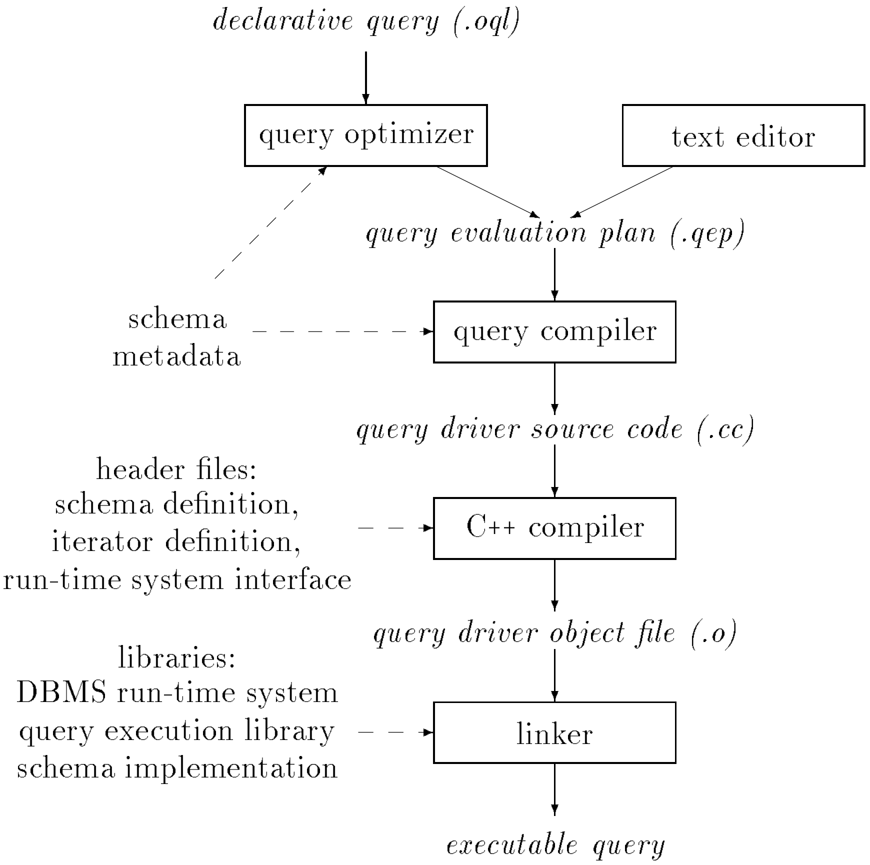


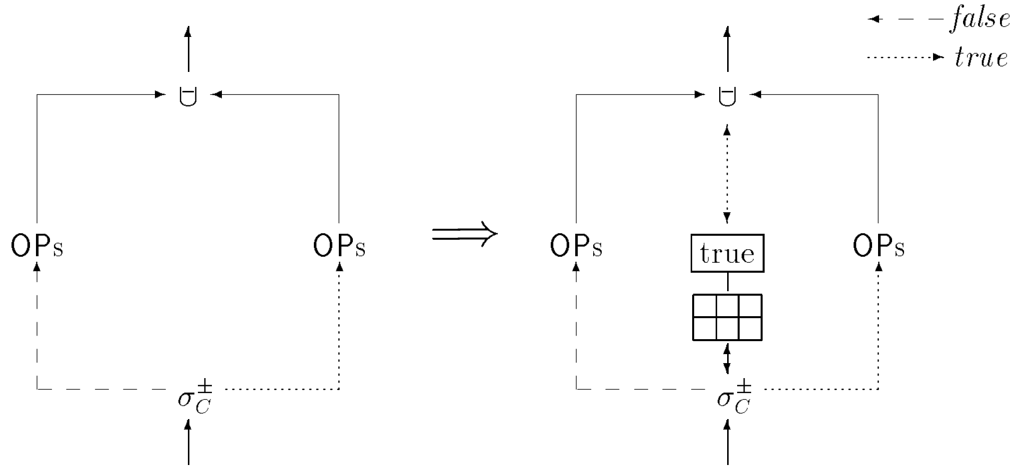

Figure 14: Query Generation from Query Evaluation Plans

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.

### 6.1.3 Implementation

In our (current) query engine, a query is evaluated within a single process, thus keeping management overhead of the operating system to a minimum. 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 [Gra93]. 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.

The query engine is implemented in C++ [Str92]. 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 (OID), another one for scanning a file and returning objects (OBJ). Furthermore, there is




(a) general structure (b) implementation concept

Figure 15: Structure of Bypass Cycles in Evaluation Plans

an expand operator for accessing objects and computing method results ($\chi$, cf. Section 3), nested loops join ($\bowtie^{\mathrm{NL}}$), hybrid hash and sort merge join ($\bowtie^{\mathsf{hash}}$, $\bowtie^{\mathsf{merge}}$), selection ($\sigma$), merge union ($\uplus$), set operations ($\uplus$, $-$, ... ) and a buffering operator (**buf**).

Figure 14 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 [KMP93]. The query compiler processes the query execution plan as input, amends it with some 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 against 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

All bypass operators have two output streams. 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 contain undirected cycles, i.e., they are DAGs. Figure 15 (a) shows the general structure of a bypass cycle.

These cycles do not cause any problems as long as the operators are evaluated strictly sequentially and the virtual tables must be stored anyway. For the evaluation with a query engine based on iterators, the top-down evaluation strategy forces a merge union operator to choose one input stream to retrieve records from each time it is called. But at the moment of the decision, the operator cannot yet know which input stream the next record will arrive at. One approach for resolving this conflict seems to be to break with strict top-down control flow and to evaluate the part of the query beneath the bypass operator first, i.e., the merge union operator "cheats" and looks ahead which branch the record will choose. 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 Figure 15). 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 lookup mechanism fails, we have decided to use an additional buffer in our prototype to collect records that belong to the "wrong" input stream once we have selected one stream. In combination with the preview mechanism, this provides acceptable performance.

The conceptual implementation of the buffering technique is shown in Figure 15 (b). The grid symbolizes the bypass buffer. There is only one buffer for both output streams, and a switch, depicted in Figure 15 by the box containing currently the label "true," is used to mark what records are momentarily cached in the buffer. On whatever side there will arrive a call to produce a record, the bypass select operator either takes records for the corresponding output from the buffer, or it retrieves new records from the input operator and saves failing records 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 on the buffer to empty the buffer. This keeps the buffer quite small in the average case. Since buffer growth cannot be avoided in all cases, the possibility of paging out parts of the buffer to a local disk is provided. Other merge union policies are feasible. Especially if very high or very low selectivity is expected, simply exhausting one input while buffering non-matching records may be useful. If multiple bypass operators occur within one plan, the corresponding $\uplus$ operators may either be nested (the toplevel $\uplus$ communicating with the bottomlevel bypass operator), or an $n$-ary $\uplus$ combines all streams in one step. In this case, $n-1$ bypass buffers are necessary.

A totally different way to avoid large bypass buffers is to cause a rollback of execution: if too many non-matching records are found, control is given back to the calling merge union operator, associated with a hint to call the other input.

## 6.3 Implementing Bypass Joins

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

In contrast to simple selection operators, most join algorithms 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. 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 anti-joins, processing is quite expensive. It does not suffice to find matching records, but additionally non-matching 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: Either one uses a nested loops algorithm in connection with a bypass buffer, making no restrictions at all, or one assumes that one table will fit into main memory (this assumption is taken e.g., for division algorithms, too [GC93]). Then, all conventional join algorithms are applicable.

As with conventional semi-joins, bypass semi-joins 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 effort 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 $\uplus$ 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 implementations offers good performance, too.

# 7 Quantitative Assessment

In this section, we present a quantitative assessment of our novel bypass technique. Unfortunately, there is no (accepted) benchmark for declarative queries. In order to increase the intuitive understanding of the results we derive the initial benchmark setup from a "real" mini-world, i. e., 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 treat single 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 2.4 operating system. The system is equipped with 96 MB of main memory and a single 1 GB disk (Seagate ST31200N, avg. access time 10ms). Since for all queries we measured elapsed real time, the system was taken offline while running the experiments in order to get reproduceable results. As persistent storage system we used the locally developed *Merlin* client-server storage system [Ger96]. 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. The databases were quite small in comparison to main memory size; consequently the whole database could be kept in memory by the server. This induces that page faults are quite cheap in comparison to page faults causing disk I/O. Since it is quite difficult, however, to keep the operating system from caching pages, our solution allows for a constant (although low) cost of a page fault in the query client. If we avoided page caching in the server, all queries would suffer from disk I/O in the same way, i.e., the absolute cost of all queries would be raised, but the relation would stay the same. Temporary files were handled in the same way as persistent data files, i.e., they were transported to the (local) server.

## 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 1000 Airport objects and 29000 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. 8). 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. Figure 16 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 % of bypass cost. The CNF plan, not shown in the figure, took 6450 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

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

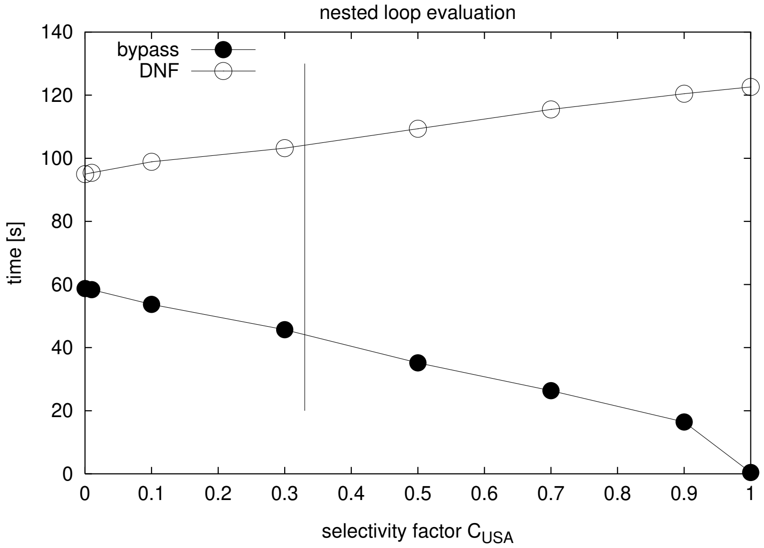



Figure 16: Nested Loops Join Evaluation of "Immigration Airports" Query

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 increasing selectivity factor, the bypassing advantage increases, too.

The same experiment has been performed using a hybrid hash algorithm for join evaluation. The results are depicted in Figure 17. 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.

## 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 30000, 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 parameters. The result of

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

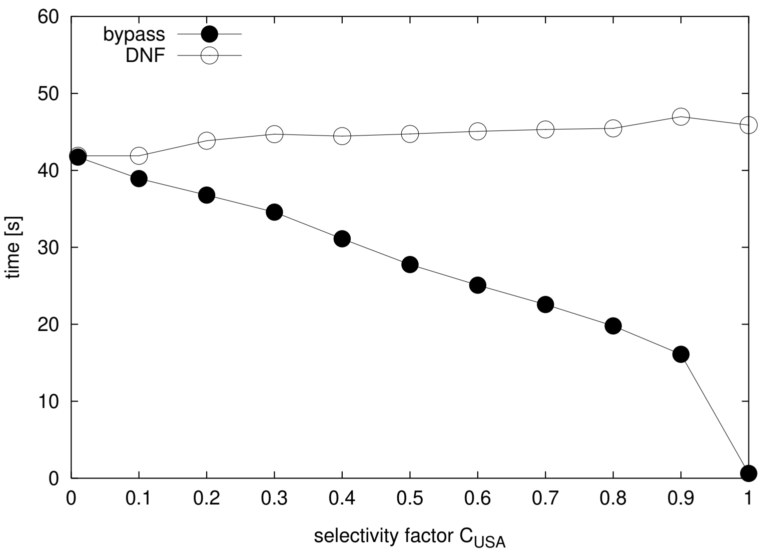



Figure 17: Hash Join Evaluation of "Immigration Airports" Query

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 Figure 18. The number of airports has been varied from 1000 to 29000, 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.

Figure 19 shows the relative cost of the DNF query in comparison to the bypass query of Figure 18 with identical parameters. As one can see, the DNF query nearly always induces 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, DNF is 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 Going more Complex

For generating more benchmark queries, we extend our schema to three object extensions $E_1$, $E_2$, and $E_3$, as e.g., $E_1 = \mathit{Airport}$, $E_2 = \mathit{Flight}$, and $E_3 = \mathit{Airlines}$. 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., $e_1.\mathit{location}$ ="*USA*," and joins $J_{12}$, $J_{23}$, and $J_{13}$ between $E_1$ and $E_2$, $E_2$ and $E_3$, and $E_1$ and $E_3$, respectively, as e.g., $e_1.\mathit{country} = e_3.\mathit{nationality}$. We will only sketch the examined queries here. For further details refer to Appendix B.

The queries always project on $E_1$. Then, a generic query is specified as follows:

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

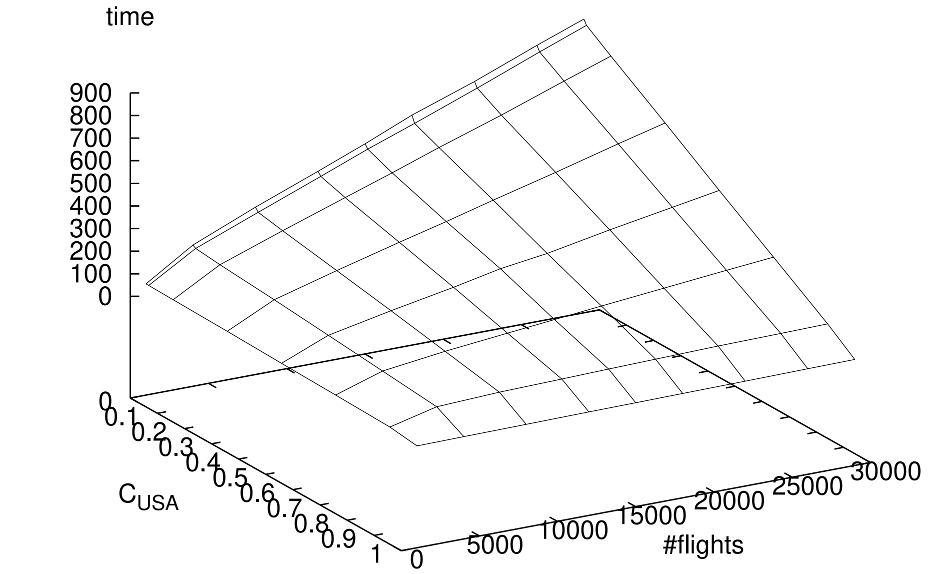



Figure 18: Evaluating "Immigration Airports" on Different Databases with Bypass

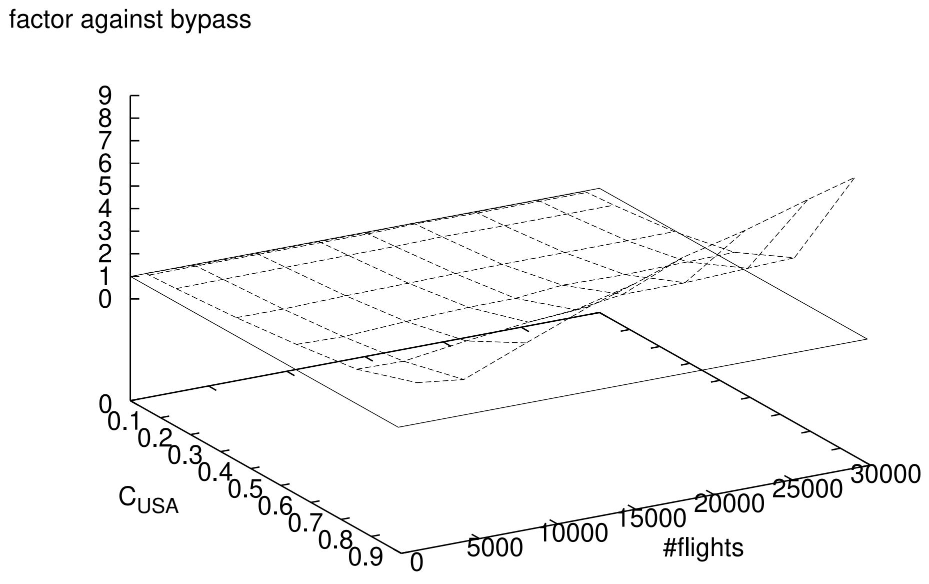



Figure 19: Comparing DNF to Bypass

where *bool* determines one 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 inducing I/O by, e.g., evaluating a path expression. We have analyzed five representative Boolean functions $bool_1$–$bool_5$. Figures 20–24 summarize the results of benchmarking the corresponding queries. Each pair of diagrams contains variants with main memory operations for invoked type-based functions on the left side and object accesses, i.e., I/O operations, on the right 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 caused always run-times of more than an hour and therefore 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 have been run.

In all charts, the selectivities of all conditions $C_i$ ($1 \leq i \leq 3$) were equal. This has the consequence that effects provoked 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 in the same moment, 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.

Let us glance through the single experiments of Figures 20–24. 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 quite cheap part (used as bypass predicate) and one more expensive part (that is bypassed), just as for $bool_1$.

The remaining functions embody two resp. three disjunctions. Figure 24 shows the maximum bypass gain, since the Boolean factors are nested and the innermost Boolean operator is a disjunction. Both properties enlarge the gain of bypassing.

## 7.5 Summary

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

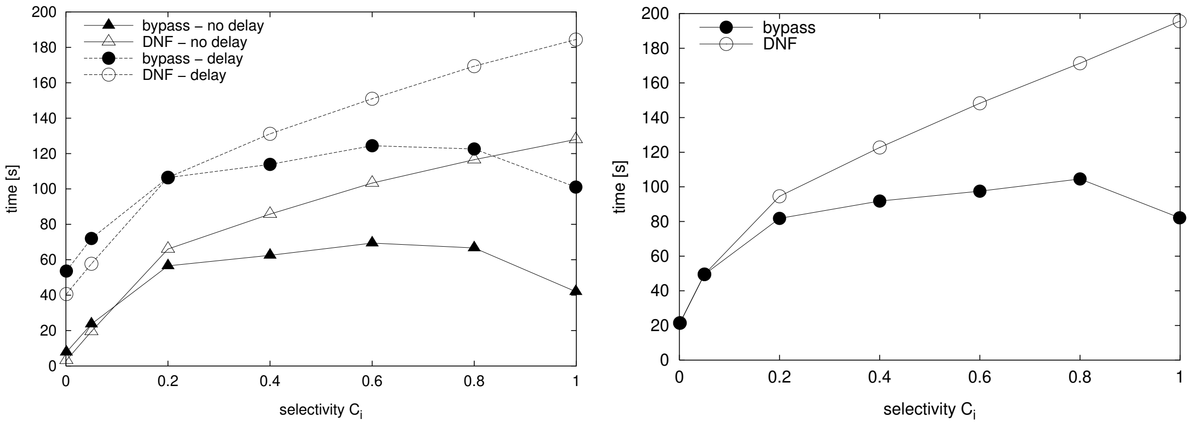



Figure 20: $bool_1 = C_1 \vee (C_2 \wedge C_3 \wedge J_{12} \wedge J_{23})$

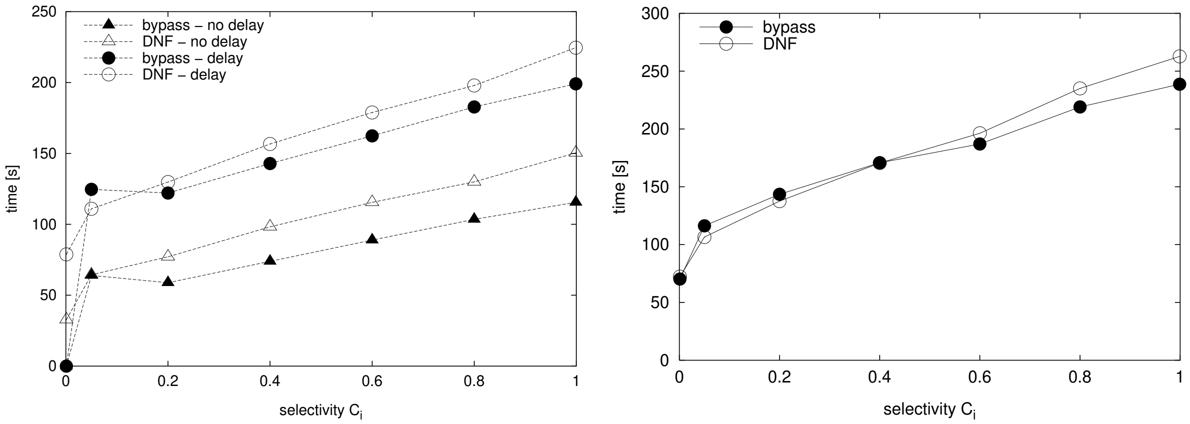



Figure 21: $bool_2 = (C_1 \wedge J_{13} \wedge C_3) \vee (J_{12} \wedge C_2 \wedge J_{23})$

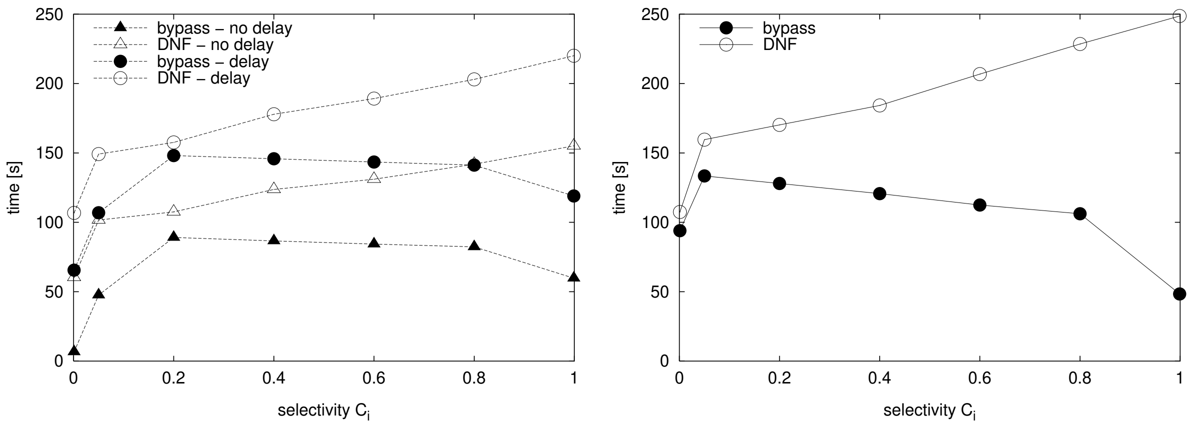



Figure 22: $bool_3 = C_1 \vee (J_{12} \wedge (C_2 \vee (J_{23} \wedge C_3)))$

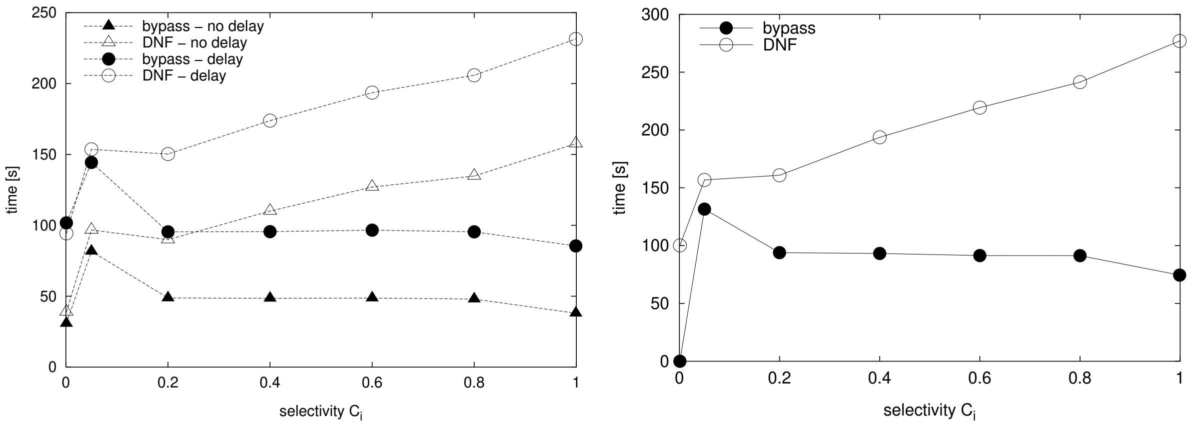



Figure 23: $bool_4 = C_1 \vee (J_{13} \wedge C_3) \vee (J_{12} \wedge C_2 \wedge J_{23})$

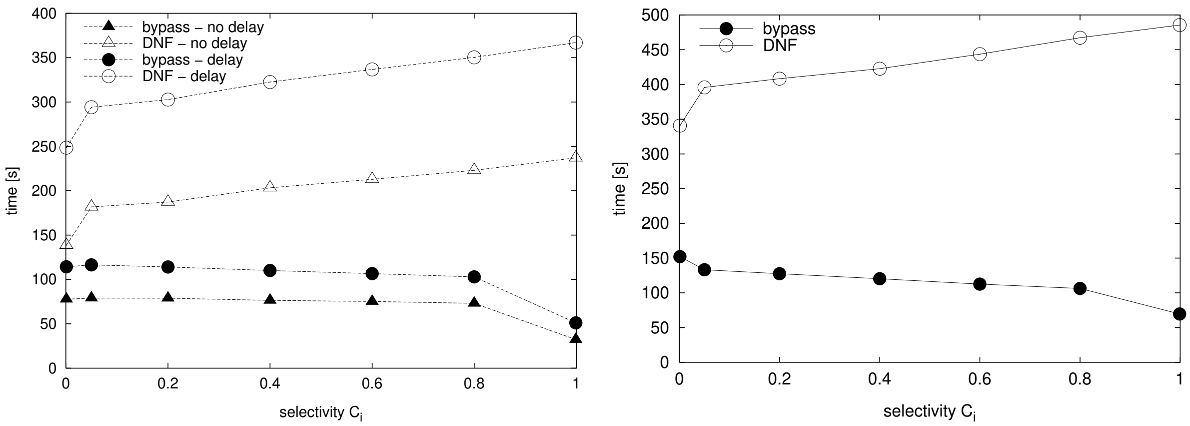



Figure 24: $bool_5 = C_1 \vee (J_{12} \wedge (C_2 \vee (J_{23} \wedge (C_3 \vee J_{13}))))$

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. Thereby it 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: [KMPS94] introduced bypass selection and [SPMK95] 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,
- 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.

# References


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

[BG92] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Trans. on Database Systems*, 17(2):247–303, June 1992.

[BGW+81] P. A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. B. Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Trans. on Database Systems*, 6(4), December 1981.

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

[Bry89] F. Bry. Towards an efficient evaluation of general queries: Quantifiers and disjunction processing revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 193–204, Portland, OR, USA, May 1989.

[Cat94] R. G. G. Cattell, editor. *Object Database Standard.* Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1994.

[CD92] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–392, San Diego, USA, June 1992.

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

[Fre87] J. C. Freytag. A rule-based view of query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 173–180, San Francisco, USA, May 1987.

[GC93] G. Graefe and R. Cole. Fast algorithms for universal quantification in large databases. Technical report, Portland State University and University of Colorado at Boulder, 1993.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, USA, May 1987.

[Ger96] C. A. Gerlhof. *Optimierung von Speicherzugriffskosten in Objektbanken: Clustering und Prefetching.* PhD thesis, Universität Passau, Fakultät für Mathematik und Informatik, D-94030 Passau, 1996. (in Vorbereitung).

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. IEEE Conf. on Data Engineering*, pages 209–218, Vienna, Austria, April 1993.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

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

[Hel94] J. M. Hellerstein. Practical predicate placement. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 325–335, Minneapolis, MI, USA, May 1994.

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

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

[IK84] T. Ibaraki and T. Kameda. Optimal nesting for computing $N$-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.

[IW87] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 9–22, San Francisco, USA, May 1987.

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

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 128–137, Kyoto, Japan, 1986.

[KM90] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 290–301, Brisbane, Australia, 1990.

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

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

[KMPS94] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 336–347, Minneapolis, MI, USA, May 1994.

[KTY82] L. Kerschberg, P. D. Ting, and S. B. Yao. Query optimization in a star computer network. *ACM Trans. on Database Systems*, 7(4):678–711, December 1982.

[Law78] E. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Ann. Discrete Math.*, 2:75–90, 1978.

[LMS94] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 96–107, Santiago, Chile, September 1994.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, IL, USA, May 1988.

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

[MS79] C. Monma and J. Sidney. Sequencing with series-parallel precedence constraints. *Mathematics of Operations Research*, 4:215–224, 1979.

[Mur88] M. Muralikrishna. Optimization of multiple-disjunct queries in a relational database system. Technical Report #750, University of Wisconsin–Madison, February 1988.

[ÖS90] M. T. Özsu and D. D. Straube. Queries and query processing in object-oriented database systems. *ACM Trans. Office Inf. Syst.*, 8(4):387–430, October 1990.

[Pei96] Klaus Peithner. *Optimierung deklarativer Anfragen in Objektbanken.* PhD thesis, Ringstr. 32, 53757 Sankt Augustin, Germany, 1996. Dissertation, Universität Passau.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.

[SPMK95] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper. Bypassing joins in disjunctive queries. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 228–238, Zürich, Switzerland, September 1995.

[SS91] D. Stemple and T. Sheard. A recursive base for database programming primitives. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science (LNCS)*, April 1991.

[Ste96] M. Steinbrunn. *Heuristic and Randomised Optimisation Techniques in Object-Oriented Database Systems.* infix-Verlag, Ringstraße 32, 53757 St. Augustin, Germany, 1996. Dissertation, Universität Passau.

[Str92] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, USA, second edition, 1992.

[Swa89] A. Swami. Optimization of large join queries: Combining heuristics and combinational techniques. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 367–376, Portland, OR, USA, May 1989.

[vB91] G. von Bültzingsloewen. *SQL-Anfragen - Optimierung für parallele Bearbeitung.* FZI-Berichte Informatik. Springer-Verlag, New York, Berlin, etc., 1991.

# A Correctness of Bypass Rules

In this section, we will prove that our transformation rules (1) and (2) introducing bypass operators maintain correctness with respect to SQL/OQL semantics.

**Theorem 1** *The evaluation plan transformations described by equations (1) and (2) maintain correct SQL semantics, i.e., all evaluation plans generated by applying these transformations yield the same result table as the canonical form* $Q = \sigma_{g(C_1,\ldots,C_m)}(R_1 \times \cdots \times R_n)$.

The general idea is that each application of a bypass transformation rule splits the initial cartesian product $R_1 \times \cdots \times R_n$ into one more partitions, thus simply decomposing the result set.

To get the idea, here is a simple explanation for rule (1): This rule changes only one of the participating relations, namely $R_j$. By definition (cf. equations in Section 3) $\sigma_C^-(R_j) = R_j - \sigma_C^+(R_j)$, such that the pair $\sigma^+/\sigma^-$ splits $R_j$ into two partitions with $\sigma_C^+(R_j) \cup \sigma_C^-(R_j) = R_j$ and $\sigma_C^+(R_j) \cap \sigma_C^-(R_j) = \emptyset$. Consequently, $(R_1 \times \cdots \times \sigma_{C_i}^-(R_j) \times \cdots \times R_n)$ and $(R_1 \times \cdots \times \sigma_{C_i}^+(R_j) \times \cdots \times R_n)$ is a partition of $(R_1 \times \cdots \times R_j \times \cdots \times R_n)$.

In the following proofs, we will rely on the definition of bypass operations from Section 3.

## A.1 Proof of Rule (1)

Note that since we want to prove SQL semantics, in the following proofs we always talk about multi-sets (bags). We use, however, normal braces as bag constructors. The same proof is applicable if set semantics (without duplicates) is required.

We will start with the initial cross product and rewrite it in bag notation:

$$\sigma_{g(C_1,\ldots,C_m)}(R_1 \times \cdots \times R_n) = \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g(C_1, \ldots, C_m)(r)\}$$

Without changing the contents, we can introduce the predicate $C_i(r_j)$ and $\neg C_i(r_j)$ where $\neg C_i(r_j)$ has to evaluate to *true* for a null value.

$$\begin{aligned} &= \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g(C_1, \ldots, C_m)(r) \wedge \mathit{true}\} \\ &= \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g(C_1, \ldots, C_m)(r) \wedge (\neg C_i(r_j) \vee C_i(r_j))\} \end{aligned}$$

The additional predicate $C_i$ can be distributed to form two *disjoint* collections:

$$\begin{aligned} = \;& \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge (g(C_1, \ldots, C_m)(r) \wedge \neg C_i(r_j) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee g(C_1, \ldots, C_m)(r) \wedge C_i(r_j))\} \\ = \;& \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge (g(C_1, \ldots, C_m)(r) \wedge \neg C_i(r_j))\} \\ \uplus \;& \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge (g(C_1, \ldots, C_m)(r) \wedge C_i(r_j))\} \end{aligned}$$

Finally, $g$ can be restricted to $C_i = \mathit{false}$ resp. $C_i = \mathit{true}$ and the definition for $\sigma^+/\sigma^-$ is applied.

$$\begin{aligned} = \;& \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g|_{C_i(r_j)=\mathit{false}}(C_1, \ldots, C_m)(r) \wedge \neg C_i(r_j)\} \\ \uplus \;& \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g|_{C_i(r_j)=\mathit{true}}(C_1, \ldots, C_m)(r) \wedge C_i(r_j)\} \\ = \;& \sigma_{g|_{C_i=\mathit{false}}}(R_1 \times \cdots \times \sigma_{C_i}^-(R_j) \times \cdots \times R_n) \uplus \sigma_{g|_{C_i=\mathit{true}}}(R_1 \times \cdots \times \sigma_{C_i}^+(R_j) \times \cdots \times R_n) \end{aligned}$$

Thus we know that introducing one bypass selection operator splits the initial cross product into two partitions with all tuples assigned to exactly one partition.

## A.2 Proof of Rule (2)

We will start as before and introduce predicate $C_i(r_j, r_k)$, this time depending on two relations, i.e. $C_i$ is a join condition.

$$\begin{aligned}
&\sigma_{g(C_1,\ldots,C_m)}(R_1 \times \cdots \times R_n) = \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g(C_1, \ldots, C_m)(r)\} \\
&= \quad \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g(C_1, \ldots, C_m)(r) \wedge \mathit{true}\} \\
&= \quad \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g(C_1, \ldots, C_m)(r) \wedge (\neg C_i(r_j, r_k) \vee C_i(r_j, r_k))\}
\end{aligned}$$

$C_i$ is distributed again to form two disjoint collections:

$$\begin{aligned}
= \quad & \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge (g(C_1, \ldots, C_m)(r) \wedge \neg C_i(r_j, r_k) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \vee\, g(C_1, \ldots, C_m)(r) \wedge C_i(r_j, r_k))\} \\
= \quad & \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g(C_1, \ldots, C_m)(r) \wedge \neg C_i(r_j, r_k)\} \\
\uplus \; & \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g(C_1, \ldots, C_m)(r) \wedge C_i(r_j, r_k)\}
\end{aligned}$$

$C_i = \mathit{false}$ resp. $C_i = \mathit{true}$ and the definition for $\bowtie^+ / \bowtie^-$ is applied.

$$\begin{aligned}
= \quad & \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g|_{C_i(r_j,r_k)=\mathit{false}}(C_1, \ldots, C_m)(r) \wedge \neg C_i(r_j, r_k)\} \\
\uplus \; & \{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge g|_{C_i(r_j,r_k)=\mathit{true}}(C_1, \ldots, C_m)(r) \wedge C_i(r_j, r_k)\} \\
= \quad & \sigma_{g|_{C_i=\mathit{false}}}\{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge \neg C_i(r_j, r_k)\} \\
\uplus \; & \sigma_{g|_{C_i=\mathit{true}}}\{r | r = (r_1, \ldots, r_n) \in R_1 \times \cdots \times R_n \wedge C_i(r_j, r_k)\} \\
= \quad & \sigma_{g|_{C_i=\mathit{false}}}(R_1 \times \cdots \times \bowtie^-_{C_i}(R_j, R_k) \times \cdots \times R_n) \\
\uplus \; & \sigma_{g|_{C_i=\mathit{true}}}(R_1 \times \cdots \times \bowtie^+_{C_i}(R_j, R_k) \times \cdots \times R_n)
\end{aligned}$$

# 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 $\chi$-operations—one for a restriction and two for a join. Thus, we obtain the following dependencies between scanning the objects of the extensions, performing $\chi$-operations, and evaluating conditions:

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

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

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

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:

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

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

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:

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

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:

$$\begin{array}{rcl} type(T_i) & = & int \\ f_i(a_i) & = & for\ (j = 1; j < a_i; j{+}{+});\ //\ return\ value\ is\ a_i \\ \phi_i & = & \left\{ \begin{array}{ll} `<' & for\ 1 \leq i \leq 3 \\ `=' & for\ 4 \leq i \leq 9 \end{array} \right. \\ c_i & = & variable \end{array}$$

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

2. Dereferencing: The function $f_i$ causes additional page faults. The functions $f_i$

operate on different types:

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

Both variants are printed side-by-side in Figures 20–24.

The extensions of all object types $E_i$ and $T_i$ had a cardinality of 10000 objects each, with objects of the constant size of 100 bytes. The client page buffer was always allocated to a size of 200kB.