Bypassing Joins in Disjunctive Queries*

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

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

[‡]RWTH Aachen
Lehrstuhl für Informatik III
52074 Aachen, Germany
moer@gom.informatik.rwth-aachen.de

Abstract

In this paper we develop a novel optimization strategy for disjunctive queries involving join predicates. This work is an extension of our previously published approach [KMPS94] for optimizing disjunctive selection predicates by generating two output streams from selection operators: a "true"-stream for objects (tuples) satisfying the selection predicate and a "false"-stream for those objects not satisfying the predicate. Then, each stream undergoes an individual, "customized" optimization.

Here, we extend the basic idea of [KMPS94] to disjunctive queries with join operators. Analogously to selections, we propose to generate two output streams from a join operator: one "true"-stream for pairs of objects (of the two input streams) that satisfy the join predicate and one "false"-stream for those pairs not satisfying it. In combination with the extended selection predicate processing, this provides a large potential for efficiently evaluating disjunctive queries because it allows to "bypass" expensive or non-selective joins as well as selections whenever the outcome of the query can be determined otherwise. In the paper, we develop techniques for effectively generating optimal and near-optimal "bypass" evaluation plans and quantitatively assess their quality in comparison to traditional plans for a range of benchmark queries.

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]

*This work was supported by the German Research Council under contract DFG Ke 401/6-2.

made rule-based query optimization popular, which was later adopted in the object-oriented context [OS90, KM90, CD92]. Many researchers have worked on optimizer architectures that facilitate flexibility: [GD87, Bat86, BMG93, GM93] are proposals for optimizer generators; [BG92, HFLP89] described extensible optimizers in the extended relational context; [MDZ93, KMP93] proposed architectural frameworks for query optimization in object bases.

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. This approach yields the optimal evaluation sequence for *conjunctive* selection predicates.

However, it is striking that in all these works the optimization of disjunctive query predicates tends to be neglected. The traditional approaches transform a query predicate (i.e., either selection or join predicate) 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$^+$79]) or the predicate is subdivided into several conjunction threads that are optimized separately (disjunctive normal form, e.g., [BGW$^+$81, KTY82, OS90, Mur88]).

In this paper, we show that both approaches fail to exploit a vast optimization potential, because a sufficiently fine tuned adaptation to a particular query's characteristics cannot be done that way. Our so-called *bypass technique* fills the gap between the achievements of traditional query optimization and the theoretical potential. In [KMPS94], the basic idea of bypassing, namely dynamically determining the smallest subsets of a predicate's conditions necessary to cover the result, was applied to queries that did not contain join operations. The main component of the evaluation plans introduced in [KMPS94] is a special selection operator that separates its input set into two disjoint output sets of tuples that satisfy the selection predicate and those that do not, respectively. Already this rather limited area of application proved to be quite profitable, so we may expect an even higher optimization potential if the bypass technique is applied to queries that comprise joins as well as selections.

In Section 2, the idea of join bypassing and its superiority to traditional techniques is illustrated by means of an example query. Section 3 goes into different construction methods for bypass evaluation plans, and Section 4 provides a quantitative performance analysis of the generated plans with respect to its traditional counterparts. Section 5 concludes the paper.

2 Why Bypassing Joins?

2.1 Example Query

In order to illustrate the potential savings, let us consider the following example query from the domain of a bookshop database. The underlying schema consists of five object types, namely *Book*, *Work*, *Publisher*, *Order* and *Person*. The attributes of these object types are:

$$\begin{array}{rl} \textit{Book} & [\textit{work}: \textit{Work}, \textit{publisher}: \textit{Publisher}, \textit{stock}: \textit{integer}] \\ \textit{Work} & [\textit{author}: \textit{Person}, \textit{title}: \textit{string}] \\ \textit{Publisher} & [\textit{pname}: \textit{string}, \textit{paddress}: \textit{string}] \\ \textit{Order} & [\textit{customer}: \textit{Person}, \textit{book}: \textit{Book}, \textit{quantity}: \textit{integer}] \\ \textit{Person} & [\textit{name}: \textit{string}, \textit{first}: \textit{string}, \textit{address}: \textit{string}] \end{array}$$

A *Work* is written by an author (a *Person*) and bears a title, that, if published by a *Publisher*, makes up a *Book*. A *Work* may be published by more than one *Publisher* (e.g., different publishers for hardcover and paperback versions). The purchase of a *Book* requires an *Order*, which involves a customer (a *Person*, too) and comprises a certain quantity. Note that even though this schema is designed for an object-oriented database system similar to the ODMG standard [Cat94], the application of the bypassing technique described below is definitely not limited to this kind of systems, but can also be used in conventional relational and extended relational database systems without any modification.

Based on this schema, we might be interested in particular authors (together with the books they have written) with the following properties:

1. They buy the books they have written themselves **or**
2. There is an order for at least one of their books with a quantity exceeding the number in stock **or**
3. They publish themselves (for the purpose of the query, determined as those authors that share their address with a publisher)

This query can be stated—resembling OQL [Cat94]—as:

select	**distinct** w.author.name, w.author.first, w.title
from	w **in** Work, o **in** Order, p **in** Publisher
where	(o.book.work.author = o.customer **and** w = o.book.work) **or**
	(o.quantity > o.book.stock **and** w = o.book.work) **or**
	w.author.address = p.paddress

Abbreviating the four conditions within the selection predicate as $C_{customer}(o)$ (o.book.work.author = o.customer), $C_{work}(w, o)$ (w = o.book.work), $C_{quantity}(o)$ (o.quantity > o.book.stock) and $C_{address}(w, p)$ (w.author.address = p.paddress), the result of the query can be expressed as the set:

$$\begin{aligned}&\{\, w \in \mathit{Work} \mid \exists o \in \mathit{Order}, p \in \mathit{Publisher}: \\ &\quad C_{customer}(o) \wedge C_{work}(w, o) \vee C_{quantity}(o) \wedge C_{work}(w, o) \vee C_{address}(w, p) \,\}\end{aligned} \tag{1}$$

The generated query evaluation plans are (logical) algebraic expressions dealing with relations. There is one scan operator[1] (*scan*) loading the objects of a particular type extension (or tuples of a relation) into a main memory buffer. Furthermore, our algebra is restricted to projection π, selection σ, join $\bowtie$, semi-join $\ltimes$ and $\rtimes$, cross product $\times$ and two union operators ($\overset{+}{\cup}$ and $\overset{-}{\cup}$) with and without duplicate elimination, respectively. Due to this restricted set of operators that comprises the lowest common denominator for any relational or object-oriented database system, our results are not limited to a particular data model.

2.2 Evaluation Plan Alternatives

In this section, we shall show the benefits of bypass evaluation plans by means of the example query stated above. Figure 1 depicts the optimal bypass evaluation plan for the query. For comparison purposes, the optimal evaluation plans that are based on the conjunctive normal form (CNF) and the disjunctive normal form (DNF) of the query predicate, respectively, are shown as well (Figure 2 and 3). The latter two approaches are the prevailing strategies in existing database systems for dealing with predicates in general and disjunctive predicates in particular.

However, comparing the average evaluation cost figures for these three alternatives shows that the two "classic" strategies are not capable of computing the query result with costs as low as the bypass plan's (16,000 units). A closer look at the three evaluation plans will reveal the reason for the differences. Please note that in this section, we shall give just the results of the cost calculations according to our cost model. The cost model itself and a sample application (namely, for the bypass plan) is provided in Appendix A.

The bypass evaluation plan starts on the left-hand side with the semijoin operation $\rtimes_{C_{address}}$ of *Publisher* and *Work* and the selection $\sigma_{C_{quantity}}$ of *Order* on the right-hand side. Tuples satisfying $C_{address}$ are certain to be elements of the result set, hence they may *bypass* the other operation nodes of the evaluation plan. A similar reasoning applies to the two selection operations on the right-hand side: satisfying either $C_{quantity}$ or $C_{customer}$ suffices in order to qualify for further processing. The merge union node $\overset{-}{\cup}$ re-unites these two streams and,

[1]Additional scan operators, e.g., supported by indices, are not relevant with respect to the topic of this paper.

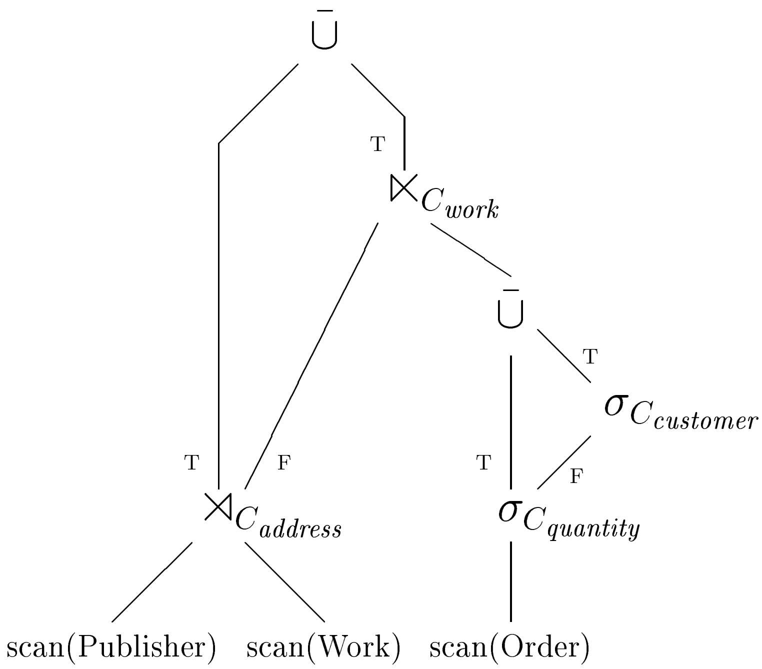


Figure 1: Optimal (Bypass) plan for example query (average cost: $16,000$ units)

in turn, provides one of the semijoin $\ltimes_{C_{work}}$'s input streams. The second input stream consists of all the tuples that do not satisfy the already mentioned other semijoin $\rtimes_{C_{address}}$. The output of $\ltimes_{C_{work}}$ is the second of the two (disjoint) subsets that make up the query's result. It can be noted as the main characteristic of bypass evaluation plans that enhanced selection and join operators are employed which do not merely provide those tuples that satisfy the operation's predicate, but those that do not as well. The resulting *two* tuple streams are necessarily disjoint, which makes expensive duplicate eliminating union operators dispensable. Subsequently, each of these two streams undergoes an individual, “customized” optimization process which entails the very efficient query evaluation plans—in contrast to traditional techniques. The plans derived by these traditional techniques are depicted in Figure 2 (CNF) and 3 (DNF). Both CNF- and DNF-based evaluation plans are common approaches for evaluating predicates in join and selection operations.

Let us first turn to the CNF-based evaluation plan in Figure 2. CNF-based plans employ a limited kind of bypassing within Boolean factors (for instance, if $C_{address}$ in the join predicate of Figure 2's plan turns out to be true, evaluation of C_{work} is bypassed) and between Boolean factors (if the Boolean factor $C_{address} \vee C_{work}$ turns out to be false, evaluation of $C_{address} \vee C_{quantity} \vee C_{customer}$ is not

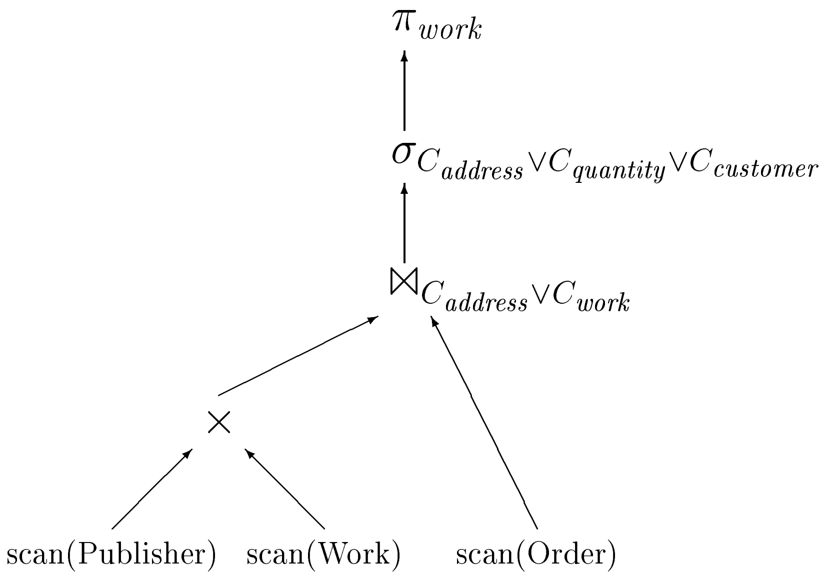


Figure 2: CNF plan for example query (average cost: 1,460,130 units)

carried out). Hence, CNF-based plans are a proper subset of bypass evaluation plans: every CNF-based plan can be expressed as a bypass plan, but not vice versa. However, being the optimal CNF-based evaluation plan for the example query, Figure 2 suggests that the conjunctive normal form (namely, $(C_{address} \vee C_{work}) \wedge (C_{address} \vee C_{quantity} \vee C_{customer})$) is probably not the construction base of choice for disjunctive queries. The division into Boolean factors requires joining of all relations involved in the query before the first selection takes place. If there are more than two relations, cartesian products are unavoidable. This fact is reflected in the CNF-based plan's average evaluation cost of 1,460,130 units, almost ninety times the cost of the equivalent bypass plan. The join operation is the biggest contributor, due to high input cardinalities. This figure already takes into account the implicit bypassing expressed by the CNF-plan as well as caching of condition evaluation results. It is striking that no cost-reducing semijoin can be employed, because attributes from both of the join's input relations are needed further on. Clearly, CNF-based query evaluation plans cannot be the answer to our problem, although they work quite well for conjunctive predicates.

In contrast, DNF-based plans are much more capable of dealing with disjunctive queries, as Figure 3 shows. As the main difference compared to CNF-based plans, the "cloning" of tuple streams can be noted. For each of the disjuncts of the DNF (the query has already been stated as DNF, cf. equation (1) on page 4),

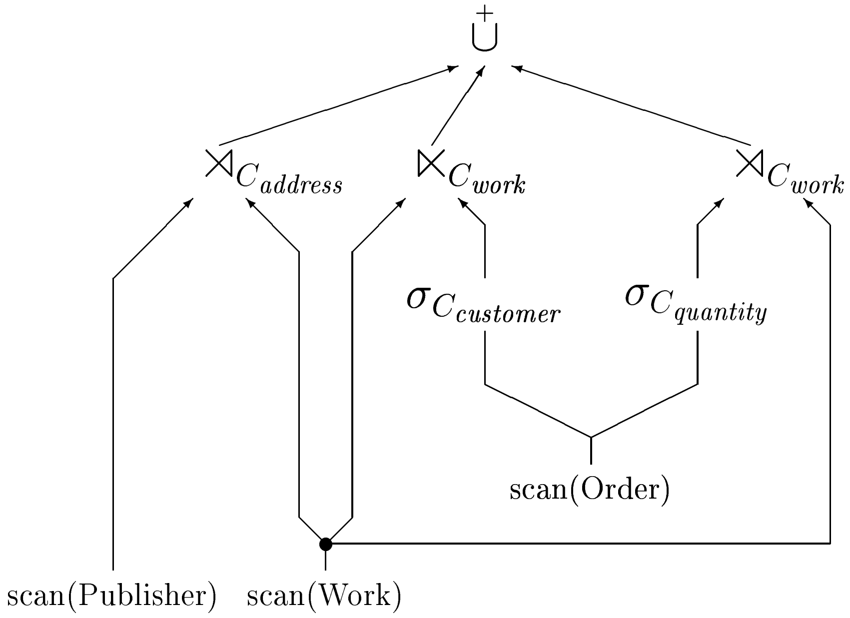


Figure 3: DNF plan for example query (average cost: 22,175 units)

a separate tuple stream is generated. The resulting plan is far less expensive to evaluate than the CNF-based plan, but with 22,175 units it still comes short of the bypass plan (in terms of saved cost) by about 6,000 units. The reason for this shortcoming is chiefly the need to eliminate duplicates in the final union operation and the fact that a condition has to be evaluated repeatedly for a given tuple (namely, once for each stream). A predicate like $a \wedge (c \vee d \vee e)$ illustrates this phenomenon: because the DNF is $(a \wedge c) \vee (a \wedge d) \vee (a \wedge e)$, condition a would appear in each of the three streams, and the higher condition a's relative cost were, the more apparent the DNF approach's weakness would be.

From these considerations the conclusion can be drawn that the bypass technique is a universally applicable strategy that combines the advantages of both CNF- and DNF-based approaches and avoid their disadvantages: neither duplicate elimination nor caching is required for bypass evaluation plans to work.

3 Constructing Evaluation Plans

In this section, we shall outline two algorithms that generate bypass plans with joins, and—for comparison purposes—the traditional construction algorithms. Their working principles will be explained by means of the example query from

Section 2.1, equation (1). For convenience, we repeat the query predicate below.

$$C_{customer}(o) \wedge C_{work}(w,o) \vee C_{quantity}(o) \wedge C_{work}(w,o) \vee C_{address}(w,p)$$

3.1 Bypass Plans

Before we start, we make some definitions. First, substituting a condition C_i by the constant "true" ("false") is denoted as $g_{C_i:=\text{true}}$ ($g_{C_i:=\text{false}}$). For instance, $g_{C_1:=\text{true}} = C_2 \vee C_3$ for $g = C_1 \vee C_2 \vee C_3$.

The second definition introduces the notion of a *bundle* with a *control function*: Let e_1, e_2, ..., e_n be some algebraic expressions, g be a Boolean function, and C be a condition of g. Then,

$$\begin{aligned} \sigma_g(e_1 \times \cdots \times e_n) &= \sigma_{g_{C:=\text{true}}}(e_1 \times \cdots \times \sigma_C(e_i) \times \cdots \times e_n) \,\bar{\cup} \\ &\quad\ \sigma_{g_{C:=\text{false}}}(e_1 \times \cdots \times \sigma_{\neg C}(e_i) \times \cdots \times e_n) \end{aligned} \quad (2)$$

holds if e_i binds the free variable(s) of C and

$$\begin{aligned} \sigma_g(e_1 \times \cdots \times e_n) &= \sigma_{g_{C:=\text{true}}}(e_1 \times \cdots \times e_i \ltimes_C e_j \times \cdots \times e_n) \,\bar{\cup} \\ &\quad\ \sigma_{g_{C:=\text{false}}}(e_1 \times \cdots \times e_i \ltimes_{\neg C} e_j \times \cdots \times e_n) \end{aligned} \quad (3)$$

holds if e_i and e_j together bind the free variables of C. Recall that $\bar{\cup}$ denotes the union operator that needs not perform duplicate elimination, as duplicates cannot occur here. For the description of the construction algorithms, we shall use the more convenient notation $\{e_1, \ldots, e_n\}_g$ for the expression $\sigma_g(e_1 \times \cdots \times e_n)$. We call $\{e_1, \ldots, e_n\}_g$ a *bundle* with *control function* g. Employing this notation, equation (2) can be written as

$$\begin{aligned} \{e_1, \ldots, e_n\}_g &= \{e_1, \ldots, \sigma_C(e_i), \ldots, e_n\}_{g_{C:=\text{true}}} \,\bar{\cup} \\ &\quad\ \{e_1, \ldots, \sigma_{\neg C}(e_i), \ldots, e_n\}_{g_{C:=\text{false}}} \end{aligned} \quad (4)$$

and equation (3) as

$$\begin{aligned} \{e_1, \ldots, e_n\}_g &= \{e_1, \ldots, e_i \ltimes_C e_j, \ldots, e_n\}_{g_{C:=\text{true}}} \,\bar{\cup} \\ &\quad\ \{e_1, \ldots, e_i \ltimes_{\neg C} e_j, \ldots, e_n\}_{g_{C:=\text{false}}} \end{aligned} \quad (5)$$

For a query $\sigma_g(R_1 \times \cdots \times R_n)$, the following algorithms start with the initial bundle $\{R_1, \ldots, R_n\}_g$ and apply equation (4) or (5) repeatedly until a set of bundles with control functions $g' = \text{true}$ or $g' = \text{false}$ is obtained.

FIX The first algorithm is called "FIX" since it integrates the conditions C_1, C_2, ..., C_n of the entire selection predicate in a FIXed order. An exhaustive search of all possible orders leads to the solution of "FIX." Let us consider the example:

First, there is a single bundle consisting of the scan operators and the entire selection predicate as control function ($\{scan(Publisher), scan(Work), scan(Order)\}$ is denoted as $\{P, W, O\}$):

$$\{P, W, O\}_{(C_{customer}(o) \wedge C_{work}(w,o) \vee C_{quantity}(o) \wedge C_{work}(w,o) \vee C_{address}(w,p))}$$

Now, the conditions $C_{address}$, $C_{quantity}$, $C_{customer}$ and C_{work} (in exactly this order) are moved into the bundle. After the first step, the introduction of $C_{address}$, the following two bundles are obtained:

$$\begin{array}{l} \{(P \bowtie_{C_{address}} W), O\}_{\text{true}} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W), O\}_{(C_{customer}(o) \wedge C_{work}(w,o) \vee C_{quantity}(o) \wedge C_{work}(w,o))} \end{array}$$

The first of these two bundles does not need further consideration, because the control function "true" indicates that tuples in this bundles are elements of the result set.

The next condition to be incorporated is $C_{quantity}$. As a result, the second bundle is split into two, which makes a total of three bundles so far:

$$\begin{array}{l} \{(P \bowtie_{C_{address}} W), O\}_{\text{true}} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W), \sigma_{C_{quantity}}(O)\}_{C_{work}(w,o)} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W), \sigma_{\neg C_{quantity}}(O)\}_{C_{customer}(o) \wedge C_{work}(w,o)} \end{array}$$

The third condition from our list is $C_{customer}$. After its introduction, one bundle bears the control function "false" and can thus be discarded (its tuples are certain not to be elements of the result set).

$$\begin{array}{l} \{(P \bowtie_{C_{address}} W), O\}_{\text{true}} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W), \sigma_{C_{quantity}}(O)\}_{C_{work}(w,o)} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W), \sigma_{C_{customer}}(\sigma_{\neg C_{quantity}}(O))\}_{C_{work}(w,o)} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W), \sigma_{\neg C_{customer}}(\sigma_{\neg C_{quantity}}(O))\}_{\text{false}} \end{array}$$

Now there are two bundles with the same control function. They can be merged into a single bundle; this operation yields:

$$\begin{array}{l} \{(P \bowtie_{C_{address}} W), O\}_{\text{true}} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W), (\sigma_{C_{quantity}}(O) \bar{\cup} \sigma_{C_{customer}}(\sigma_{\neg C_{quantity}}(O)))\}_{C_{work}(w,o)} \end{array}$$

And finally, after introducing the last condition C_{work}, two true- and one false-bundle remain.

$$\begin{array}{l} \{(P \bowtie_{C_{address}} W), O\}_{\text{true}} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W) \bowtie_{C_{work}} (\sigma_{C_{quantity}}(O) \bar{\cup} \sigma_{C_{customer}}(\sigma_{\neg C_{quantity}}(O)))\}_{\text{true}} \bar{\cup} \\ \{(P \bowtie_{\neg C_{address}} W) \bowtie_{\neg C_{work}} (\sigma_{C_{quantity}}(O) \bar{\cup} \sigma_{C_{customer}}(\sigma_{\neg C_{quantity}}(O)))\}_{\text{false}} \end{array}$$

The two true-bundles are led together by a merge union $\bar{\cup}$ resulting in the optimal evaluation plan in Figure 1. The only difference to Figure 1, the kind of join nodes employed, will be discussed in Section 3.2 below. If a bundle consists of more than one expression which binds variables interesting for the outcome of the query, a cross product of the expressions will be applied before performing the union.

For a selection predicate with n conditions, the "FIX" strategy has to consider $n!$ permutations as candidates for the optimal fixed order bypass evaluation plan.

OPT The "FIX" strategy constructs evaluation plans where the conditions' evaluation order is the same for all possible paths from the first stage (relation scan) to the final stage (union of all disjoint streams). In other words, the evaluation order is always determined for the *entire* evaluation plan.

However, sometimes it is advantageous to construct evaluation plans where the evaluation orders are not determined globally, but independently for each possible path 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, the search space examined by "OPT" is considerably larger: up to $(n-0)^1$ alternatives have to be considered in the first step, $(n-1)^2$ in the second step, and $(n-k+1)^{2^{k-1}}$ in the kth step, resulting in a worst-case total of $\prod_{i=1}^{n}(n-i+1)^{2^{i-1}}$ instead of a total of $n!$ (as for "FIX") evaluation plan candidates, but in contrast to "FIX," "OPT" is certain to come up with the optimal bypass evaluation plan. However, our quantitative assessment indicates that in almost all practical cases "FIX" generates the optimal evaluation plan, although it considers fewer alternatives.

3.2 Semijoins

One open issue concerning both algorithms ("FIX" and"OPT") remains to be discussed: the introduction of semijoins. In the final evaluation plan for the example query (cf. Figure 1), both join operators are replaced by semijoins. This last step in the construction of the evaluation will now be discussed. A semijoin $R_1 \ltimes_C R_2$ is defined as:

$$R_1 \ltimes_C R_2 = \{\, r_1 \in R_1 \mid \exists r_2 \in R_2 \colon C(r_1, r_2) \,\}$$

Therefore, the cardinality of $R_1 \ltimes_C R_2$ is bounded by the cardinality of R_1 instead of the cardinality of the cartesian product $R_1 \times R_2$ as in ordinary join operations. This property makes the semijoin especially well suited for bypass evaluation plans: since the false-output from a join operation is needed as well as the true-output, a total of $|R_1|$ tuples have to be processed for a semijoin node, but $|R_1 \times R_2| = |R_1| \cdot |R_2|$ tuples for a join node.

Because the evaluation cost depends heavily on the number of tuples processed, it is obvious that join operators ought to be replaced by semijoin operators whenever possible. In Figure 1, both joins could be replaced since none of the attributes from the respective second operand were needed in later stages of the evaluation plan. Fortunately, introducing semijoins into bypass plans is rather straightforward: we choose the "cheapest" operator (i.e., $\rtimes$, $\ltimes$ or $\bowtie$) depending on the set of attributes needed further on.

3.3 Traditional Plans

For comparison purposes, we also implemented the two traditional approaches that are based on normal forms. Since these approaches are well-known [JK84] we shall only sketch them.

CNF For the CNF approach, the entire selection predicate is transformed into the Conjunctive Normal Form (CNF), and each disjunct of this normal form is regarded as a Boolean factor. Then, a two-phase optimization is performed—ordering the Boolean factors and ordering the conditions within the Boolean factors.

However, as shown for the running example (Section 2.2), the CNF approach is likely to lead to plans that produce intermediate results of enormous cardinality—a source of high costs. Furthermore, particular conditions may appear in more than one Boolean factor, which makes caching indispensable if repeated evaluation of those conditions is to be avoided. That is especially true for "expensive" conditions [HS93]. Anyway, the optimal CNF-based evaluation plan cannot possibly perform better than the optimal bypass plan, since the set of CNF-plans is a proper subset of the set of bypass plans.

DNF For the DNF approach, the entire selection predicate is transformed into the Disjunctive Normal Form (DNF), and each conjunct of this normal form is regarded as a Boolean factor. Then, each Boolean factor is independently optimized by ordering selections [MS79], ordering joins [KBZ86], and ordering selections into join orderings [HS93].

However, the computed evaluation plans contain non-disjoint tuple streams that must be united by union operators that eliminate duplicates (unlike the special-case "merge" union operators for disjoint operands that can be employed in bypass plans). Furthermore, exactly as for the CNF approach, conditions may be evaluated more than once for a given tuple (namely, if they appear in more than one stream).

4 Quantitative Assessment

The quantitative assessment described in this section compares the bypass evaluation technique with the conventional techniques based on a normal form on one hand, and the optimization algorithms "OPT" and "FIX" on the other hand. In order to carry out these comparisons, two parameters have to be varied: the queries and the profile of the object base.

The queries are specified as Boolean functions with sets of projections which are, in turn, subsets of the extensions involved. The profile of the object base is expressed as a set of so-called *basic values*. These basic values comprise the cardinalities of the object extensions (relations), the selectivities of the conditions and the conditions' evaluation cost per invocation.

In this section, the optimization potential of the bypass evaluation technique with respect to conventional techniques is determined first. Because of the lack of a standardized query benchmark, only simple queries are used, which is, in our opinion, sufficient for the purpose of this comparison. But in order to fully appreciate the performance of the two bypass optimization flavours "OPT" and "FIX," generating more complex queries is imperative. Based on a particular object base profile, we generate a large set of different queries of that kind to be optimized.

4.1 Optimizing a very simple function

In the first benchmark series, we want to examine the following two questions:

- What is the impact of the basic values, i.e., the database profile?
- Which optimization potential is obtained by applying the bypass evaluation technique—even for a simple query?

For that, we optimized the function

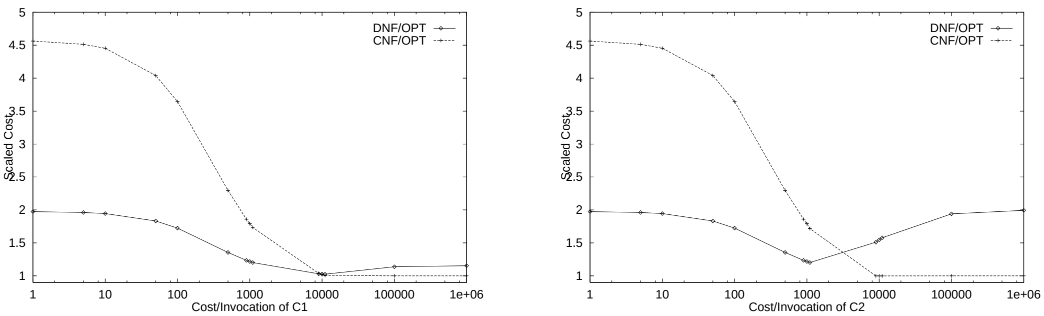
$$C_1(r_1) \vee (C_2(r_2) \wedge J(r_1, r_2)) \quad \text{for } r_1 \in R_1 \text{ and } r_2 \in R_2$$

with projection on attributes only of R_1 choosing 89 different settings of the basic values. But, before we outline the experimental results, a closer look at the Boolean functions reveals us the following possibilities for optimizations:

1. The expression $C_2(r_2) \wedge J(r_1, r_2)$ can be bypassed by objects satisfying $C_1(r_1)$.
2. The condition $C_1(r_1)$ can also be bypassed; however, in this case the join $J(r_1, r_2)$ and the condition $C_2(r_2)$ have to be evaluated before $C_1(r_1)$.
3. The join $J(r_1, r_2)$ can be transformed into a semijoin if the condition $C_2(r_2)$ is applied before the join.

	R_1	R_2		$C_1(r_1)$	$C_2(r_2)$	$J(r_1, r_2)$
Cardinality	100	100	Selectivity Factor	0.5	0.5	0.01
			Cost/Invocation	10	10	10

Table 1: Default Values of $C_1(r_1) \vee (C_2(r_2) \wedge J(r_1, r_2))$



Varying cost/invocation of C_1 Varying cost/invocation of C_2

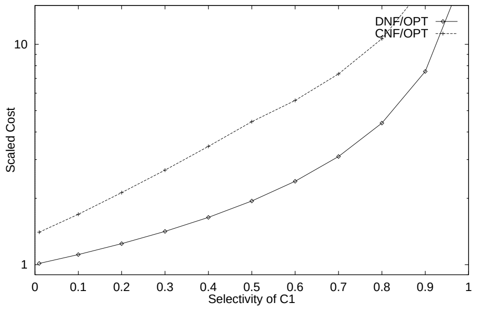
Figure 4: Impact of the conditions' cost values

4. The inputs of $J(r_1, r_2)$ can be reduced to those objects r_1 which do not satisfy $C_1(r_1)$ and those objects r_2 which satisfy $C_2(r_2)$, respectively.

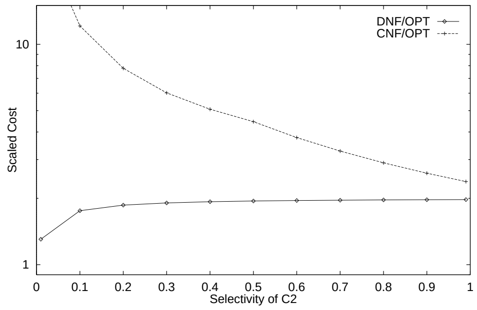
The conventional evaluation techniques fail at least in one of these points. For example, the DNF-based plans can bypass neither $C_2(r_2) \wedge J(r_1, r_2)$ nor $C_1(r_1)$. And, since both Boolean factors of the conjunctive normal form select R_1 as well as R_2, the CNF-based plans cannot take advantage of semijoins. Now, we want to assess quantitatively how strong these shortcomings of the conventional evaluation techniques are.

In the following diagrams, the two quotients CNF/OPT and DNF/OPT are depicted—we omit the quotient FIX/OPT since for this simple function "FIX" always computes the optimal plan, hence the quotient always equals 1.0. We varied the invocation costs and the selectivity factors of the conditions and the cardinalities of the extensions. All parameters except the varied one are set to default values which are depicted in Table 1.

Varying the cost values of $C_1(r_1)$ and $C_2(r_2)$ results in the diagrams of Figure 4. The CNF-based optimization cannot transform the join $J(r_1, r_2)$ into a semijoin. This will lead to more than four times higher evaluation costs if the query is not dominated by high cost values of $C_1(r_1)$ or $C_2(r_2)$. The DNF-based optimization is not able to bypass the condition $C_1(r_1)$—a fact which is less important for low evaluation cost values of C_1. However, if the cost value of $C_1(r_1)$ exceeds 10,000, the optimal bypass plan applies the expression $C_2(r_2) \wedge J(r_1, r_2)$



Varying selectivity factor of C_1



Varying selectivity factor of C_2

Figure 5: Impact of the conditions' selectivity factors

before $C_1(r_1)$ which yields approximately 15% better performance. Bypassing $C_2(r_2) \wedge J(r_1, r_2)$ will even yield 100% better performance if the query's evaluation costs are dominated by the join or by high evaluation costs of $C_2(r_2)$.

The bypass effect is increased by a high selectivity factor of $C_1(r_1)$, which is demonstrated in the left-hand side diagram of Figure 5. The higher this selectivity factor is, the more objects of R_1 can bypass the join and the better the bypass plan performs with respect to conventional plans. The diagram on the right-hand side of Figure 5 shows another interesting behavior. The CNF-based optimization cannot apply $C_2(r_2)$ as a restriction on R_2, since its Boolean factor also contains $C_1(r_1)$. If the selectivity factor of $C_2(r_2)$ is low, it results in a low input cardinality of the join in DNF-based and in bypass plans. Hence, these plans are about ten times more efficient than the equivalent CNF-based evaluation plan.

There is no impact of the cardinalities; the plots of the quotients CNF/OPT and DNF/OPT are almost straight lines parallel to the x-axis. Therefore, these diagrams are not shown.

4.2 Optimizing pure disjunctive joins

The second benchmark series assesses the benefits of bypassing joins in a disjunctive "star query", i.e., the join graph of the query forms a star. Especially, the DNF-based evaluation plans cannot exploit the bypass effect in queries with disjunctively connected joins, since these plans evaluate all disjuncts independently of each other.

We optimized the Boolean functions f_0, f_1, …, f_9 which are recursively defined as follows:

$$\begin{aligned} f_0 &= J_0(r, r_0) \\ f_i &= f_{i-1} \vee J(r, r_i) \quad \text{for } 1 \leq i \leq 9 \end{aligned}$$

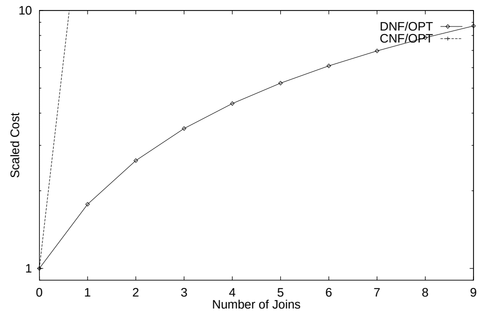


Figure 6: Impact of the number of joins

where r, r_0, ..., r_9 are bound to R, R_1, ..., R_9, respectively. As setup, we took one large extension ($card(R)$ = 10,000) and some small joining extensions ($card(R_i) = 200$; $0 \leq i \leq 9$). The costs of the joins are always 10 and the selectivity factors are always 0.01.

The results of this benchmark are depicted in Figure 6. For this kind of queries, the CNF-based plans are inordinately bad, since they have to generate the cross product of all involved extensions. But also, the DNF-based plans are extremely bad because of the inability of bypassing: the evaluation costs of the optimal DNF-based plans are between 77% and 771% worse than the optimal bypass plans' costs for $1 \leq x \leq 9$, i.e., from two to ten joins.

4.3 Comparison of OPT and FIX

The third benchmark series evaluates the quality of "FIX" in comparison with "OPT." For that, we took four extensions with cardinality 10, 100, 1000, and 10000. For choice for the randomly generated Boolean functions, there were one restriction per extension, and two joins per pair of extensions. The cost values and the selectivity factors of the conditions were also randomly chosen within a given range. The range of the cost values was $[1, 30000]$, and the selectivity ranges were $[0.1, 0.5]$ for the restrictions and $[0.0001, 0.5]$ for the joins.

As Boolean function, we took up to five conditions from the restrictions and the joins and connected them by **and**/**or** randomly. As projections, we took a random number of the extensions which are involved in the chosen Boolean function. In this manner, we generated 100 queries and optimized them by "FIX" and "OPT."

In total, "FIX" was only 2% worse than "OPT." But, in order to obtain a better idea of the quality of "FIX" we subdivided the queries according to the quotients FIX/OPT. We counted the number of queries and we determined the average of the quotients of generated alternatives between "FIX" and "OPT" within the intervals $[1, 1]$, $(1, 1.01]$, $(1.01, 1.05]$, $(1.05, 1.2]$, $(1.2, 2]$, and $(2, \infty)$.

Cost FIX/OPT	$[1,1]$	$(1,1.01]$	$(1.01,1.05]$	$(1.05,1.2]$	$(1.2,2]$	$(2,\infty)$
Number of queries	90	2	2	3	3	0
Alternatives considered FIX/OPT	0.48	0.67	0.44	0.47	0.51	0.00

Table 2: Comparison between OPT and FIX

The results are depicted in Table 2. According to the third row in Table 2, "OPT" considered only about twice as many alternatives than "FIX," a figure that is far lower than the worst case $\prod_{i=1}^{n}(n-i+1)^{2^{i-1}}/n!$ (cf. Section 3). The reason is that a bundle's control function, once reduced to "true" or "false," does not need to be considered further, which effectively prunes the search space. To summarize, we note that "FIX" computed the optimal evaluation plan for 90% of the queries, although it considered on the average only about one half of OPT's search space.

5 Conclusion

In this paper, we introduced the new bypass evaluation technique, a technique that is especially well suited for disjunctive queries in any kind of database system, be it relational, extended relational or object-oriented. The bypass technique minimizes the amount of processing required to compute the result of selection or join operations by distributing the respective input set into two output sets. One of the output sets contains the tuples that satisfy the selection or join predicate, the other those that do not.

Furthermore, we introduced two possible construction methods named "FIX" and "OPT" for bypass plans and compared them with plans derived by traditional techniques, namely based on the conjunctive or disjunctive normal form. The quantitative assessment confirmed the presumption that the bypass technique as an evaluation method that does not perform superfluous computations is superior to the traditional methods employed in current database systems. Although the cost reductions that can be achieved depend on the particular shape of the query predicates, the cost relations of their conditions and the database profile, it turned out that bypass evaluation plans performed never worse than traditional plans, but much better in the vast majority of cases—as much as an order of magnitude.

The development of efficient heuristics that facilitates the construction of bypass plans that are not necessarily optimal, but can be derived very quickly is an area of future work. One possible approach that we are currently pursuing is an appropriate adaptation of the Boolean difference calculus, which already has been successfully employed for the solution of related problems [KMS92, KMPS94]. But even the algorithms presented in this paper that perform an (in the worst case) exhaustive search of the solution space tap an enormous optimization potential that well rewards the expenses.

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), Dec 1981.

[BMG93] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the Open OODB Query Optimizer. In SIGMOD [SIG93], pages 287–295.

[Cat94] R. G. G. Cattell. *Object Database Standard.* Morgan-Kaufmann Publ. Co., 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, San Mateo, USA, 1993.

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

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

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

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

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In SIGMOD [SIG93], pages 267–276.

[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, August 1990.

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

[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, USA, May 1994.

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

[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, Dec 1982.

[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, 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, USA, 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, August 1993.

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

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

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

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

[SIG93] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Washington, DC, USA, May 1993.

A Comparing Evaluation Costs

A.1 A Simple Cost Model

In order to assess the query evaluation plans generated by the different optimization algorithms, we use a simple but meaningful cost model. This cost model is founded on the statistical profile of the database, the so-called *basic values*. These values comprise

- The cardinality $|R|$ of every type extension (relation) R,
- the number of different values v_a for each extension R's attributes a (denoted as e-value $v_a(R)$ for extension R's attribute a)
- the probability of a condition C being true, i.e., its selectivity factor $s(C)$ and
- the cost $c(C)$ for evaluating the condition C (expressed as cost units per invocation).

They capture the most important characteristics of the database and suffice for computing a coarse, yet reliable cost estimate.

The basic values include the conditions' selectivity factors, because the—at first glance—obvious computation from the involved extensions' e-values is feasible only for very simple conditions (such as equality of two attributes). As soon as the conditions get more complicated, this approach becomes increasingly difficult or even impossible. Table 3 lists the basic numbers for the bookshop database. They can be obtained, for instance, by sampling the database or similar methods—the exact algorithms are not relevant in the context of this paper. For each operation in the query evaluation plan, from

- the input cardinalities,
- the input e-values,
- the conditions' costs and
- the conditions' selectivities

	Work	Book	Publisher	Order	Person
Cardinality	100	200	10	500	2000

(a) Type Extension Cardinalities

Path	e-values
o.book.work.author	500.80.70.50
o.customer	500.400
o.book.work	500.80.70
o.quantity	500.300
w.author.address	100.80.80
p.paddress	10.10

(b) e-values

	$C_{customer}(o)$	$C_{work}(w, o)$	$C_{quantity}(o)$	$C_{address}(w, p)$
Selectivity Factor	0.04	0.01	0.05	0.05
Cost/Invocation	4	2	3	3

(c) Query Condition Characteristics

Table 3: Basic Values

Operation	Cost
$scan(R)$	0
$\bar{\cup}(R_1, \ldots, R_n)$	0
$\overset{+}{\cup}(R_1, \ldots, R_n)$	$\sum_{i=1}^{n} \lvert R_i \rvert \cdot \log_2 \sum_{i=1}^{n} \lvert R_i \rvert$
$\sigma_{C(a_1)}(R)$	$\lvert R \rvert + v_{a_1}(R) \cdot c(C)$
$R_1 \bowtie_{C(a_1,a_2)} R_2$	$\lvert R_1 \rvert \cdot \lvert R_2 \rvert + v_{a_1}(R_1) \cdot v_{a_2}(R_2) \cdot c(C)$
$R_1 \ltimes_{C(a_1,a_2)} R_2$	$\lvert R_1 \rvert \cdot \max(1, (1 - s(C)) \cdot \lvert R_2 \rvert)$
	$+ v_{a_1}(R_1) \cdot c(C) \cdot \max(1, (1 - s(C)) \cdot v_{a_2}(R_2))$
$R_1 \times R_2$	$\lvert R_1 \rvert \cdot \lvert R_2 \rvert$

(a) Costs

Operation	Output Cardinality	Output e-value v_{a_1}
$scan(R)$	basic value	basic value
$\bar{\cup}(R_1, \ldots, R_n)$	$\sum_{i=1}^{n} \lvert R_i \rvert$	$\max_{i \in \{1,\ldots,n\}}\{v_{a_1}(R_i)\}$
$\overset{+}{\cup}(R_1, \ldots, R_n)$	$\max_{i \in \{1,\ldots,n\}}\{\lvert R_i \rvert\}$	$\max_{i \in \{1,\ldots,n\}}\{v_{a_1}(R_i)\}$
$\sigma_{C(a_1)}(R)$	$\lvert R \rvert \cdot s(C)$	$v_{a_1}(R) \cdot s(C)$
$R_1 \bowtie_{C(a_1,a_2)} R_2$	$\lvert R_1 \rvert \cdot \lvert R_2 \rvert \cdot s(C)$	$v_{a_1}(R_1) \cdot (1 - (1 - s(C))^{v_{a_2}(R_2)})$
$R_1 \ltimes_{C(a_1,a_2)} R_2$	$\lvert R_1 \rvert \cdot (1 - (1 - s(C))^{\lvert R_2 \rvert})$	$v_{a_1}(R_1) \cdot (1 - (1 - s(C))^{v_{a_2}(R_2)})$
$R_1 \times R_2$	$\lvert R_1 \rvert \cdot \lvert R_2 \rvert$	$v_{a_1}(R_1)$

(b) Cardinalities and e-values

Table 4: Formulas for Costs, Cardinalities and e-values

the following data are computed according to the formulas in Table 4:

- the operation's cost
- the output cardinalities and
- the output e-values.

The cost for the entire query evaluation plan is computed bottom-up as sum of the costs of every operation involved.

In Table 4, a_1 and a_2 are attributes belonging to R_1 (R), and R_2, respectively. Due to limited room, only one output e-value is listed. It is assumed that an operator accessing attribute a_1 (such as $\sigma_{C(a_1)}$), but not attribute a_2, does not change the e-value of the latter. Furthermore, the e-value calculation of symmetric binary operators is completely symmetric as well, so v_{a_2} is easily induced from v_{a_1}.

The cost estimates are derived from the cardinalities, e-values, selectivities, and basic numbers. Since the scan operators are the same for all evaluation plans of a particular query, their costs are neglected. Evaluating a merge union $\bar{\cup}(R_1, \ldots, R_n)$ of disjoint $R_1, \ldots, R_n$ is assumed to be free, whereas the set union $\overset{+}{\cup}(R_1, \ldots, R_n)$ incurs the cost for the elimination of duplicates. The formulas for the cardinality of $\overset{+}{\cup}$ and e-values for either $\overset{+}{\cup}$ or $\bar{\cup}$ denote the lower bound of the range of possible values (for instance, cardinality of $\overset{+}{\cup}(R_1, \ldots, R_n)$ may be as high as $\sum_{i=1}^{n} |R_i|$). However, the accuracy of these estimates does not have a high impact on the total cost estimate, since these values are hardly ever needed.

A.2 Sample Cost Computation

In Section 2.2, the costs for the three different evaluation plans for our example query were discussed without providing the computation of these costs. Now we shall carry out such a computation for the evaluation plan in Figure 1, the bypass evaluation plan. As the scan operations' costs are neglected, we start the calculation with the semijoin $\rtimes_{C_{address}}$. The semijoin cost is determined as

$$\begin{aligned} & |\mathit{Work}| \cdot \max(1, (1 - s(C_{address})) \cdot |\mathit{Publisher}|) + \\ & \quad v_w(\mathit{Work}) \cdot c(C_{address}) \cdot \max(1, (1 - s(C_{address})) \cdot v_p(\mathit{Publisher})) \\ = \; & 100 \cdot \max(1, 9.5) + 100 \cdot 3 \cdot \max(1, 9.5) \\ = \; & 3800 \end{aligned}$$

The emanating tuple streams' cardinalities and e-values are:

1. true-stream:

   (a) cardinality

$$\begin{aligned} & |\mathit{Work}| \cdot (1 - (1 - s(C_{address}))^{|\mathit{Publisher}|}) \\ = \; & 100 \cdot (1 - (1 - 0.05)^{10}) \\ = \; & 40.13 \end{aligned}$$

   (b) e-value

$$\begin{aligned} & v_w(\mathit{Work}) \cdot (1 - (1 - s(C_{address}))^{v_p(\mathit{Publisher})}) \\ = \; & 100 \cdot (1 - (1 - 0.05)^{10}) \\ = \; & 40.13 \end{aligned}$$

2. false-stream:

   (a) cardinality

$$\begin{aligned} & |\mathit{Work}| \cdot (1 - s(C_{address}))^{|\mathit{Publisher}|} \\ = \; & 100 \cdot (1 - 0.05)^{10} \\ = \; & 59.87 \end{aligned}$$

(b) e-value

$$\begin{aligned} & v_w(Work) \cdot (1 - s(C_{address}))^{v_p(Publisher)} \\ = \; & 100 \cdot (1 - 0.05)^{10} \\ = \; & 59.87 \end{aligned}$$

Next, we calculate the data for the selection node $\sigma_{C_{quantity}}$. This node's cost is:

$$\begin{aligned} & |Order| + v_o(Order) \cdot c(C_{quantity}) \\ = \; & 500 + 500 \cdot 3 \\ = \; & 2000 \end{aligned}$$

Cardinalities for the true-/false-streams are

$$\begin{aligned} |Order| \cdot s(C_{quantity}) &= 500 \cdot 0.05 = 25 \quad \text{(true)} \\ |Order| \cdot (1 - s(C_{quantity})) &= 475 \quad \text{(false)} \end{aligned}$$

e-values are

$$\begin{aligned} v_o(Order) \cdot s(C_{quantity}) &= 500 \cdot 0.05 = 25 \quad \text{(true)} \\ v_o(Order) \cdot (1 - s(C_{quantity})) &= 475 \quad \text{(false)} \end{aligned}$$

For the selection $\sigma_{C_{customer}}$, we derive analogously:

$$\begin{aligned} \text{Cost} &= 2375 \\ \text{Cardinality}/e\text{-value} &= 19 \quad \text{(true)} \\ &= 456 \quad \text{(false, discarded)} \end{aligned}$$

The subsequent merge union operator does not incur any cost, it merely re-unites the "surviving" tuples from $\sigma_{C_{quantity}}$ and $\sigma_{C_{customer}}$ into a common stream. Thus, $25 + 19 = 44$ tuples arrive at the second semijoin operator $\ltimes_{C_{work}}$. This node's cost is:

$$\begin{aligned} & 59.87 \cdot \max(1, (1 - s(C_{work})) \cdot 44) + \\ & \qquad 59.87 \cdot c(C_{work}) \cdot \max(1, (1 - s(C_{work})) \cdot 44) \\ = \; & 59.87 \cdot \max(1, 43.46) + 59.87 \cdot 2 \cdot \max(1, 43.46) \\ = \; & 7823.81 \end{aligned}$$

The cardinalities/e-values are:

$$\begin{aligned} 59.87 \cdot (1 - (1 - s(C_{work}))^{44}) &= 21.4 \quad \text{(true)} \\ 59.87 \cdot (1 - s(C_{work})^{44} &= 38.47 \quad \text{(false, discarded)} \end{aligned}$$

The evaluation plan's total average cost is the sum of every operation node's cost, namely

$$3800 + 2000 + 2375 + 7823.81 = 15998.81 \approx 16000 \text{ cost units.}$$