Optimizing Boolean Expressions in Object Bases

Alfons Kemper* Guido Moerkotte+ Michael Steinbrunn*

*Lehrstuhl für Informatik III
RWTH Aachen
Ahornstraße 55
W-5100 Aachen
Germany
kemper
steinbrunn @informatik.rwth-aachen.de

+Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten
W-7500 Karlsruhe
Germany
moer@ira.uka.de

Abstract

In this paper we address the problem of optimizing the evaluation of boolean expressions in the context of object-oriented data modelling. We develop a new heuristic for optimizing the evaluation sequence of boolean expressions based on selectivity and cost estimates of the terms constituting the boolean expression. The quality and efficiency of the heuristic is evaluated based on a quantitative analysis which compares our heuristic with the optimal, but infeasible algorithm and other known methods. The heuristic is based on the selectivity and evaluation-cost estimates of the terms of which the boolean expression is composed. Deriving these inputs of the heuristics, i.e., the selectivity and cost estimates, is then addressed. We use an adaptation of well-known sampling for estimating the selectivity of terms. The cost estimation is much more complex than in the relational context due to the possibility of invoking functions within a boolean expression. We develop the *decapsulation* method that derives cost estimates by analysing the implementation of (encapsulated) functions.

Proceedings of the 18th VLDB Conference Vancouver, British Columbia, Canada 1992

1 Introduction

Object-oriented database systems are emerging as the next-generation database technology—especially for new applications, e.g., mechanical CAD/CAM. However, despite the increased functionality compared to relational systems, the object-oriented technology will only be widely accepted in the "non-standard" database application domain if it yields the needed performance since engineers are in general not willing to trade performance for functionality. Of course, the large body of knowledge of optimization techniques that was gathered over the last 15 years in, e.g., the relational area, provides a good starting point for optimizing object-oriented database systems. But lastly, only those optimization techniques that are specifically tailored for the object-oriented model will yield—the much-needed—drastic performance improvements.

In this paper we describe one (further) piece in the mosaic of performance enhancement techniques that we incorporated in our experimental object base system GOM [KM90b]: the optimization of boolean expressions. The use of complex boolean expressions is not restricted to declarative queries; they are also frequently encountered in the realization of methods within object-oriented databases. The need for optimizing methods in object bases has been motivated by [GM88, LD91]. Thus, optimizing the evaluation of boolean expressions seems worthwhile from the standpoint of declarative query optimization as well as method optimization.

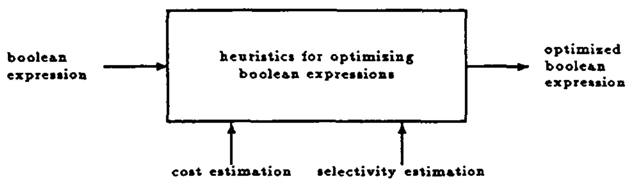
Our approach exploits knowledge from different areas and customizes these known concepts to the needs of the object-oriented data model(s). This paper describes the adaptation and synthesis of three formerly isolated areas related to the optimization of boolean expressions:

optimization algorithms: There exist optimization algorithms for boolean expressions which where developed mainly in the areas of compiler construction and operations research. An algorithm guaranteeing the optimal result has a terrible run time behaviour and is of no use for expressions with more than four or five variables. This is the reason why several heuristics have been developed to approximate the optimum with better run time performance. Within this paper we introduce a new algorithm for optimizing boolean expressions which performs both, in run time and outcome, better than the best known approximation. All algorithms for optimizing boolean expressions need two equally important inputs: *selectivity estimates* and *cost estimates.*

selectivity estimation: In order to determine the selectivity of the boolean terms we adapt the sampling methods developed in the relational context for our purpose.

cost estimation: In order to generate a (near-) optimal evaluation sequence the optimization algorithm needs to determine the evaluation cost of boolean terms. In the context of object-oriented databases the boolean expression will generally contain function invocations. For the cost estimation of functions we develop the so-called *decapsulation* that inspects the function implementation in order to estimate the costs in terms of physical I/O-cost. This estimate can then be enhanced by sampling the actual execution costs of the functions under consideration during the usage of the object base.

Each of these three issues is dealt with in a spearate section of the paper; there we cite (and briefly analyse) the related work. The interplay of these three concepts is shown schematically as follows:


boolean expression
heuristics for optimizing boolean expressions
optimized boolean expression
cost estimation
selectivity estimation

Based on randomly generated boolean expressions with a varying number of variables, we provide an evaluation of the heuristics based on the efficiency (in terms of runtime) of the algorithm and the quality of the generated result. It turns out that our heuristic is very close to the optimal algorithm in those cases where the optimal result could be generated within reasonable time.

Bearing
InDiameter, OutDiameter, Length, Geometry, Weight, Lubricants, MaxRPM, Price, OperatingLife, LubricantPossible

- **AntiFrictionBearing**
  Tolerance, C, C0, Encapsulation, StatCoefficient, DynCoefficient
- **SleeveBearing**
  Material, Clearance, EmergencyLubrication
  - **HydrostaticBearing**
    LubricationPressure, LubGrooveNr
  - **HydrodynamicBearing**
    MinRPM, InitialMaxLoad

Figure 1: Type Hierarchy of "Bearing"

The remainder of this paper is organized as follows. In the next section we motivate the optimization method by way of a "real" mechanical engineering example. Then, in Section 3, we describe our new heuristic for optimizing boolean expressions. In Section 4.1 we review and adapt existing techniques for selectivity estimation. In Section 4.2 we develop the *decapsulation*, a new approach for estimating the cost of encapsulated functions based on code analysis. In Section 5, the virtues of this new algorithm is evaluated based on a quantitative analysis. Section 6 concludes this paper with an assessment of the achieved optimization.

2 Examples and Evaluation Costs

To illustrate and motivate the concepts presented in this paper, we will introduce two examples. The first one, derived from a (real-life) engineering application, consists of the definition of the type "Bearing" with subtypes "AntifrictionBearing" and "SleeveBearing"; the latter being further specialized into "HydrostaticBearing" and "HydrodynamicBearing" (cf. Figure 1); they are supposed to be included in a database of ISO[1] standard parts to support engineers during the process of construction.

The following expression might be used to test whether a particular bearing *bx* is suitable for the intended purpose. Please note that it consists of ten conditions, so a slightly more complex expression of this kind is beyond any attempt of "manual" optimization; even with today's computing power it is not possible to obtain the optimal solution within reasonable time limits, as we shall see in Section 3.

[1]International Organization for Standardization

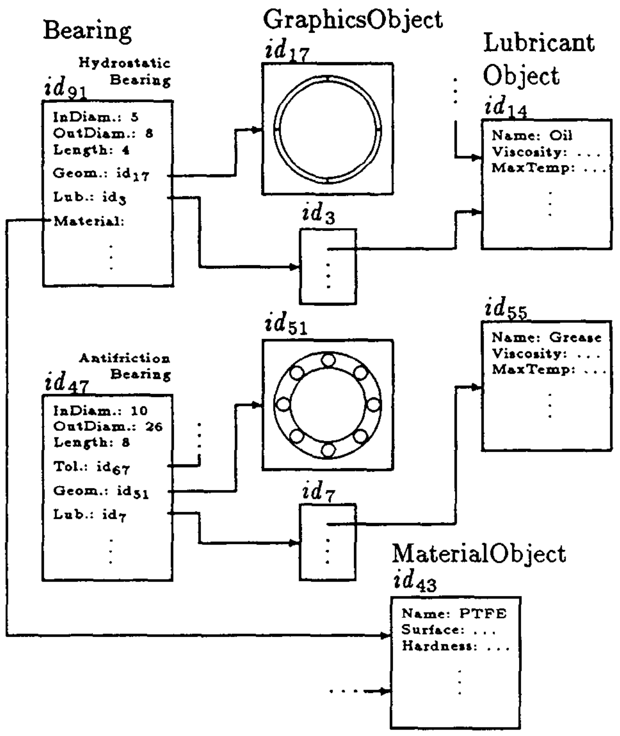


Figure 2: Sample Object Base (some attributes omitted)

bx.InDiameter = 10 ∧ bx.OutDiameter = 30 ∧
bx.Length ≥ 7 ∧ bx.Length ≤ 10 ∧
bx.MaxRPM ≥ 15000 ∧
bx.OperatingLife ≥ 10 ∧
(bx.Price ≤ 7.50 ∨
ord(bx.Tolerance.Class) ≥
AverageTolerance(ext(AntifrictionBearing))) ∧
(bx.Encapsulation ∨ bx.LubricantPossible("Grease"))

The expression checks whether the bearing has certain sizes and stands at least 15000 RPM. Further, there is a limit for the price as long as the precision is less than average, the bearing should last at least ten years and lubrication can be provided either by grease or by encapsulation with life-lubrication. The items shown in Figure 2 constitute a few objects in our sample object base. The objects id_3 and id_7 denote *set objects* that contain all the lubricants suitable for the particular bearing.

Please note that both the definition of our sample bearing database as well as the above predicate constitute extreme simplifications. But even so, it is evident that complex boolean expressions with more than five conditions are not uncommon in this kind of application. Furthermore, the selectivities as well as the costs for evaluating the conditions can differ considerably.

	Condition	Cost	Probability
x_1	InDiameter = 10	1	0.04
x_2	OutDiameter = 30	1	0.02
x_3	Length ≥ 7	1	0.94
x_4	Length ≤ 10	1	0.17
x_5	MaxRPM ≥ 15000	1	0.28
x_6	Price ≤ 7.50	1	0.13
x_7	Tolerance.Class ≥ AverageTolerance(...)	1802	0.11
x_8	OperatingLife ≥ 10	2	0.81
x_9	Encapsulation	1	0.40
x_{10}	LubricantPossible("Grease")	12	0.85

Table 1: Costs and Probabilities of the Conditions in the Sample Expression

Therefore, the first problem to be solved is the generation of an evaluation plan that takes the different costs and selectivities into account. The second problem is answering the question "How can these data possibly be obtained?". The human reader "sees" at once that the condition "ord(bx.Tolerance.Class) ≥ AverageTolerance(ext(AntifrictionBearing))" is quite costly (it involves iteration over a set) and that the likelihood of "bx.MaxRPM ≥ 15000" is high (a bearing of this size does usually reach this limit). A formal analysis of this idea is deferred to Section 4.1 and 4.2.

Let us investigate this example a bit more thoroughly. Defining one cost unit as a dereferentiation operation (i.e., one object access), each of the conditions can be assigned the appropriate probability and cost (cf. Table 1). It is assumed that the object base contains about 900 "AntifrictionBearing" objects and each bearing may be lubricated by about ten different lubricants.[2] A poor evaluation plan that tests each condition and computes the result in a second step induces a cost of 1823 units. In programming languages like PASCAL, this is the usual evaluation procedure. As we shall see later, a more sophisticated plan causes average cost of only 1.25 units. This is due to the fact that conditions x_1 and x_2 are very cheap to evaluate, but provide with probabilities of 0.04 and 0.02, respectively, a very high selectivity.[3]

Admittedly, this expression is not too hard to optimize. Even simple strategies, such as "skip remaining conditions as soon as the result is determined," are close to the optimum; programming languages like C or Modula-2 apply this method. But even so, the user has to analyse the expression to determine the correct (i.e., the least expensive) order, a task that is error-prone and therefore better left to the optimizer.

[2]Basically, most of this information is derived from a bearing manufacturer's catalogue.

[3]Conditions x_1 to x_{10} are *not* stochastically independent, so the probability of the conjunction cannot be computed by multiplying the probabilities of x_1 to x_{10}.

In the second, more abstract example, the optimal evaluation strategy is far less obvious (x_1, x_2 and x_3 are assumed to be stochastically independent):

$$f(x) = \overline{x_1}\,\overline{x_3} \vee x_1 x_3 \vee x_1 x_2$$

	Cost	Probability
x_1	2.0	0.5
x_2	0.8	0.7
x_3	1.5	0.4

Straightforward evaluation costs 4.3 units, the optimal evaluation on the average only 3.14 units, a reduction of more than 25% although neither probabilities nor costs show extreme variation. The evaluation plan and the generating algorithm that leads to the minimal average cost is presented in the next section (Figure 3).

3 Algorithms for Optimizing Boolean Expressions

In order to be able to perform optimizations in the first place, we must assume that all conditions are side-effect free, so the optimizer can rearrange the evaluation order or may even choose not to evaluate a condition. As side-effect free functions neither modify non-local data nor invoke other functions that do, the validity of this assumption can be checked automatically. Conditions that do comprise side-effects must be excluded from the optimization process, but the remaining, side-effect free ones open up a vast number of possible evaluation strategies. An algorithm that computes the optimal solution has been published by Reinwald and Soland [RS66], although in a different context. Other algorithms have a lower computational complexity, but do not necessarily yield the optimal result.

3.1 Optimal Evaluation Strategy and Approximations

This algorithm, published by Reinwald and Soland [RS66], guarantees the optimal evaluation plan for a boolean expression. The underlying principle, the so-called *limited-entry decision table*, has been discussed in early papers, e.g., [Pol65, GR73]. Their original purpose is the formalization of rule sets and their appropriate actions. Table 2 shows a sample decision table.

These decision tables are transformed to *decision trees* that consist of different actions as their leaves and the conditions of the rules as their inner nodes. A path through the tree involves testing the condition that is attached to the current node and determining the next step, depending on the outcome. After testing the last rule (i.e., the last inner node), the leaf determines the action to be carried out.

Conditions	Rule 1	Rule 2	Rule 3
C_1	Yes	No	No
C_2	—	Yes	No
	Action 1	Action 3	Action 2

Table 2: Sample Decision Table

The evaluation of boolean expressions is just a special case, the conditions being the variables and the actions the two possible values of the expression ("0" and "1"). The minterms can be viewed as the rules.

The optimal algorithm is a "Branch and Bound Strategy" of the kind that was introduced in the context of the "Travelling Salesman" problem. It can be sketched as follows (with n conditions $x_1, \ldots, x_n$)—details can be derived from [RS66]:

- Start with all decision trees that are equivalent to the given decision table
- Partition the set into n subsets that contain trees that test x_i first
- Estimate a lower bound for the evaluation cost for each subset.
- The subset with the lowest lower bound is chosen and sub-partitioned into $(n-1)^2$ subsets that test x_j ($j \in \{1, \ldots, i-1, i+1, \ldots, n\}$) first.
- Pick the subset with the lowest lower bound from the $(n-1) + (n-1)^2$ "leaf" subsets, etc.
- Continue partitioning until one of the subsets with the lowest lower bound contains only one element. This is the optimal tree.

In the first step of this algorithm, $(n-0)^1 = n$ possibilities are to be considered, in the second step $(n-1)^2$ possibilities, and in the k-th step $(n-k+1)^{2^{(k-1)}}$ possible extensions are considered. It is evident, that this algorithm, although yielding the optimal solution, is of no great practical value, because for more than about five conditions the resulting run time is intolerable. This view is also backed by benchmarks that we carried out to assess the qualities of different optimization algorithms (cf. Section 5).

To reduce the computational complexity, Ganapathy and Rajamaran [GR73] published an algorithm that makes use of a heuristic to determine the order of evaluation. The basic idea is to take the information content (according to Shannon's well-known formula) of a condition as a measure for its importance with respect to the outcome of the boolean expression. These figures are related to the cost for evaluation to decide which condition is to be tested next. The result is an algorithm that performs on the average slightly inferior to our algorithm that is presented in the next subsection, but with about 30% higher running time. There also exist other approximative algorithms, which turned out to behave not as well as our algorithm—they were even inferior to [GR73].

3.2 Our Algorithm, Based on the Boolean Difference Calculus

Our algorithm is based on a heuristic that applies the boolean difference calculus [Sch89] to determine the order in which conditions shall be tested. A decision, once made, is never reverted (i.e., no backtracking), so the algorithm is of much lower computational complexity compared to the optimal algorithm. Even though the result is not guaranteed to be optimal, it is, however, still very close (less than 5% worse in almost all cases) to the optimum.

3.2.1 Basic Idea

The basic idea is trying to find the condition with the best achievement/cost ratio for testing. The boolean difference calculus provides a heuristic that enables us to compare conditions on this basis: the boolean difference. Analogously to the notion of the differential, the boolean difference $\Delta_{x_i} f$ is defined as:

$$\begin{aligned} &\Delta_{x_i} f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \overset{\text{def}}{=} \\ &\quad f(x_1, \ldots, x_{i-1}, x_i = 0, x_{i+1}, \ldots, x_n) \not\equiv \\ &\quad f(x_1, \ldots, x_{i-1}, x_i = 1, x_{i+1}, \ldots, x_n) \end{aligned} \quad (1)$$

From now on, we shall use x as a shorthand for a vector like $(x_1, \ldots, x_n)$; in the examples, the value of n should be clear from the context.

The probability of the boolean difference being true is a measure of the influence of the condition x_i on the result of the expression, similarly to the gradient of a continous function. The higher the gradient, the higher the "leverage" of the variable x_i with respect to the function's value. The same principle applies to the boolean difference: The higher the probability $p(\Delta_{x_i} f)$ of $\Delta_{x_i} f$ being true, the higher the influence of the variable x_i on the value of the original boolean formula f.

Note the two extremes that may, intuitively, illustrate the approach of our heuristics. If a variable x_i is irrelevant for an expression $f(x)$, then $\Delta_{x_i} f(x) \equiv 0$ and, of course, $p(\Delta_{x_i} f(x)) = 0$. If x_i alone determines the result, $\Delta_{x_i} f(x) \equiv 1$ and, consequently, $p(\Delta_{x_i} f(x)) = 1$.

As a trivial example consider the expression $f(x) = x_1 x_2 \vee x_1 \overline{x_2}$ that can be reduced to $f(x) = x_1$; thus, the value of x_2 is irrelevant whereas the value of x_1 determines the value of $f(x)$. According to the definition (1), $\Delta_{x_1} f(x) = ((0x_2 \vee 0\overline{x_2}) \not\equiv (1x_2 \vee 1\overline{x_2})) = (0 \not\equiv 1) = 1$, and $\Delta_{x_2} f(x) = ((x_1 0 \vee x_1 1) \not\equiv (x_1 1 \vee x_1 0)) = (x_1 \not\equiv x_1) = 0$, confirming that x_2 is indeed irrelevant and x_1 alone important.

The following example shows the application of the boolean difference in order to determine the influence of an expression's variable values on the result.

Example: Consider the expression $f(x) = x_1 x_2 x_3 \vee x_4$. The boolean difference for x_1 is:

$$\begin{aligned} \Delta_{x_1} f(x) &= (0x_2x_3 \vee x_4) \not\equiv (1x_2x_3 \vee x_4) \\ &= x_4 \not\equiv x_2x_3 \vee x_4 \\ &= \overline{x_4}(x_2x_3 \vee x_4) \vee x_4\overline{(x_2x_3 \vee x_4)} \\ &= x_2x_3\overline{x_4} \end{aligned}$$

On the other hand, the boolean difference for x_4 is:

$$\begin{aligned} \Delta_{x_4} f(x) &= (x_1x_2x_3 \vee 0) \not\equiv (x_1x_2x_3 \vee 1) \\ &= x_1x_2x_3 \not\equiv 1 \\ &= \overline{x_1x_2x_3} \wedge 1 \vee x_1x_2x_3 \wedge 0 \\ &= \overline{x_1x_2x_3} \\ &= \overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \end{aligned}$$

If we assume that the probability for being true is 1/2 for all variables x_i (and all x_i are independent), we get

$$p(\Delta_{x_1} f(x)) = (1/2)^3 = 1/8$$

whereas

$$p(\Delta_{x_4} f(x)) = 1 - 2/16 = 7/8.$$

Thus, the boolean difference confirms what is already clear from "close observation:" The value of x_4 has a much greater impact on the result of the expression $f(x)$ than either one of x_1, x_2 or x_3 (as $\Delta_{x_2} f(x) = x_1x_3\overline{x_4}$, and $\Delta_{x_3} f(x) = x_1x_2\overline{x_4}$). □

3.2.2 The Algorithm

Our algorithm develops the evaluation plan for an expression by exploiting the boolean difference heuristic. For each variable (condition) x_i of the expression, $s_i = p(\Delta_{x_i} f)/cost(x_i)$ is computed. The variable with the highest value of s_i (i.e., the highest influence on the result of the expression) is selected for the first test. The algorithm is then applied recursively to the functions of the *false*- and *true*-branches of the first test, i.e., to $f_{x_i=0}$ and $f_{x_i=1}$. The complete algorithm runs as follows (let n be the number of conditions, f the expression to be evaluated):

1. If $n = 0$, there is nothing to evaluate. If $n = 1$, test x_1. Otherwise, proceed to Step 2.
2. For each variable x_i, determine
$$s_i := \frac{p(\Delta_{x_i} f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n))}{c(x_i)}$$
where $p(X)$ denotes the probability for X being true, and $c(x_i)$ denotes the cost for testing x_i.
3. Choose x_i with $s_i = \max\limits_{j=1,\ldots,n} \{s_j\}$ as the variable to be tested first (next).
4. Set $x_i := 0$ and $x_i := 1$ and apply this algorithm to the resulting two functions (with $n-1$ conditions) recursively.

The question arises, how the numerator of the fraction in Step 2 can be obtained efficiently. In order to compute the boolean difference $\Delta_{x_i} f(x)$ for a variable x_i, we assume that the truth table of $f(x)$ is available as input. The truth table is the most general way to specify $f(x)$ as well as the probability of each variable, because no assumptions about their stochastical independence are necessary: the probability can be specified for the entire minterm, not just for a single variable.

We shall illustrate the algorithm and the actual computation of the boolean difference on the basis of our example from Section 2 (repeated here for convenience):

$$f(x) = \overline{x_1}\,\overline{x_3} \vee x_1 x_3 \vee x_1 x_2$$

	Cost	Probability
x_1	2.0	0.5
x_2	0.8	0.7
x_3	1.5	0.4

The truth table that serves as the input to our algorithm looks as follows:

x_1	x_2	x_3	$f(x)$	$p(x)$
0	0	0	1	0.09
0	0	1	0	0.06
0	1	0	1	0.21
0	1	1	0	0.14
1	0	0	0	0.09
1	0	1	1	0.06
1	1	0	1	0.21
1	1	1	1	0.14

According to the definition of the boolean difference (1),
$$\Delta_{x_1} f(x_2, x_3) = f(0, x_2, x_3) \not\equiv f(1, x_2, x_3)$$
holds. For $x_2 = 0$ and $x_3 = 0$, this term yields $\Delta_{x_1} f(0,0) = f(0,0,0) \not\equiv f(0,0,1)$; $p(x_2 = 0 \wedge x_3 = 0) = p(0,0,0) + p(0,0,1) = 0.18$.[4] Filling in all possible values for x_2 and x_3, we get

[4] $p(0,0,0)$ denotes the probability $p(x_1 = 0 \wedge x_2 = 0 \wedge x_3 = 0)$

x_2	x_3	$\Delta_{x_1} f(x)$	$p(x)$
0	0	1	0.18
0	1	1	0.12
1	0	0	0.42
1	1	1	0.28

and $p(\Delta_{x_1} f(x)) = 0.18 + 0.12 + 0.28 = 0.58$. Similarly, for $\Delta_{x_2} f(x)$ and $\Delta_{x_3} f(x)$:

x_1	x_3	$\Delta_{x_2} f(x)$	$p(x)$
0	0	0	0.30
0	1	0	0.20
1	0	1	0.30
1	1	0	0.20

x_1	x_2	$\Delta_{x_3} f(x)$	$p(x)$
0	0	1	0.15
0	1	1	0.35
1	0	1	0.15
1	1	0	0.35

Finally, we derive $p(\Delta_{x_1} f(x)) = 0.58$, $p(\Delta_{x_2} f(x)) = 0.3$ and $p(\Delta_{x_3} f(x)) = 0.65$ from the tables above. Now the gain/cost ratios s_1, s_2 and s_3 can be computed as $s_1 = 0.58/2.0 = 0.29$, $s_2 = 0.3/0.8 = 0.375$ and $s_3 = 0.65/1.5 = 0.433$. Thus, the first condition that will be tested in our decision tree is x_3. The same procedure is now applied to the *false*- and *true*-branch of the first test, i.e., the functions $f_{\overline{x_3}}(x) = \overline{x_1} \vee x_1 x_2 = \overline{x_1} \vee x_2$ and $f_{x_3}(x) = x_1 \vee x_1 x_2 = x_1$. The *true*-branch consists of only one condition, namely x_1, that has to be tested in the next step and that finally determines the result of $f(x)$. The *false*-branch $f_{\overline{x_3}}$, however, must be handled analogously to the complete expression $f(x)$. $\Delta_{x_1} f_{\overline{x_3}}(x) = \big(f_{\overline{x_3}}(x_2, 0) \not\equiv f_{\overline{x_3}}(x_2 1)\big) = \big(f(0, x_2, 0) \not\equiv f(0, x_2, 1)\big)$, and for x_2 the boolean difference is $\Delta_{x_2} f_{\overline{x_3}}(x) = \big(f_{\overline{x_3}}(0, x_1) \not\equiv f_{\overline{x_3}}(1, x_3)\big) = \big(f(0, 0, x_1) \not\equiv f(0, 1, x_1)\big)$:

x_2	$\Delta_{x_1} f_{\overline{x_3}}(x)$	$p(x)$
0	1	0.18
1	0	0.42

x_1	$\Delta_{x_2} f_{\overline{x_3}}(x)$	$p(x)$
0	0	0.30
1	1	0.30

That yields $s_1 = 0.18/2.0 = 0.09$ and $s_2 = 0.30/0.8 = 0.375$. Thus, if $x_3 = 0$, the variable x_2 is tested next. If $x_3 = 0$ and $x_2 = 1$, then $f(x) = 1$; otherwise, a third test (the last remaining variable, x_1) is necessary to determine the result.

Finally, the optimization has led to the decision tree shown in Figure 3, the output of our algorithm. It turns out that this tree constitutes (in this particular case) the optimal evaluation procedure. Its average cost can be calculated as $cost_{total} = cost_{root} + p(\textit{false-branch}) \cdot cost(\textit{false-branch}) + p(\textit{true-branch}) \cdot cost(\textit{true-branch})$, where the expression $p(branch)$ indicates the probability for traversing the particular branch and $cost(branch)$ the cost for doing so. For the tree in Figure 3, the average evaluation cost $cost_{total} = c(x_3) + (1 - p(x_3)) \cdot (c(x_2) + (1 - p(x_2)) \cdot c(x_1)) + p(x_3) \cdot c(x_1) = 3.14$. □

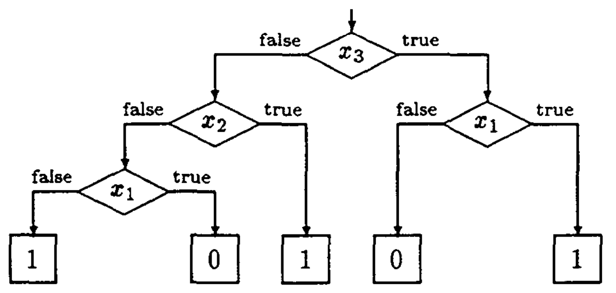


Figure 3: Result of the Optimization of the Expression $\overline{x_1}\,\overline{x_3} \vee x_1x_3 \vee x_1x_2$

4 Cost and Selectivity Estimation

By now, we have not yet addressed the problem of obtaining the input data necessary for the algorithm to work on. These input data consist of the conditions' selectivities (probability for being false) and evaluation costs.

4.1 Selectivity Estimates

In the context of query optimization in relational databases, strategies for the estimation of selectivities have been thoroughly investigated. In principle, there are two approaches: parametric and nonparametric methods. In the former, the distribution is presumed known, only the parameters have to be extracted from the database. In contrast, nonparametric methods estimate the distribution itself. A third possibility is sampling. It may be used by itself as an estimation strategy, but can also be applied to the other two. In the case of parametric methods this means that the parameters are estimated using samples rather than by scanning the entire database. But also with nonparametric methods, where a profile of the database is being maintained, this profile can as well be based on samples. Using sampling as a selectivity estimator by itself means that for each expression the samples have to be taken anew, because no information about the database is kept.

4.1.1 Parametric Methods

Parametric methods assume a certain distribution and determine the parameters, such as average, variance and so on. One of the oldest examples, the estimation strategy in System R [SAC$^+$79], is parametric: it presumes uniform distribution and stochastically independent conditions. Both assumptions constitute vast simplifications, and the techniques have been greatly enhanced in e.g., [Dem80, Chr83, Fed84, Lyn88].

4.1.2 Nonparametric Methods

In nonparametric methods, some kind of profile of the database is maintained, which allows estimation without actually knowing anything about the distribution. The simplest strategy is the so-called *(equal-width) histogram*. The domain is subdivided into a number of intervals with equal size (buckets); each of them contains the number of values that fall into the appropriate interval. To overcome certain limitations of equal-width histograms, Piatetsky-Shapiro and Connell habe proposed equal-height histograms [PSC84], where the estimation error solely depends on the number of buckets. Approaches for the selectivity estimation of conditions that are *not* stochastically independent utilize multidimensional histograms [KK85, MD88].

4.1.3 Sampling

Because the cost for a selectivity estimator can be quite high, the question arises whether a sample of the database is sufficient for determining the parameters in parametric estimators or the profile (histogram) in nonparametric estimators. Both [PSC84] and [MD88] suggest the use of sampling to avoid scanning the entire database, which proved to be a very cost-effective technique. In addition, Lipton et al. propose in [LNS90] an algorithm with an adaptive behaviour that shows that sampling by itself can be a very efficient strategy for selectivity estimation.

4.2 Decapsulation based Cost Estimation

The second part of the optimization algorithm's input consists of evaluation cost estimates. In relational database systems, only atomic conditions have to be considered. In object bases, however, it is quite common to have function invocations as part of conditions, thus it is indispensable to design a scheme that permits the cost estimation of complex conditions, i.e., functions. There are two ways to achieve this: First, one may look "into" the function definition and decompose it into atomic operations. Second, the function remains a "black box," but its invocation behaviour, i.e., the expenses, are recorded and used as a basis for the cost estimate of the future. These two possibilities are now discussed in turn.

The cost for evaluating atomic conditions is easy to estimate: all necessary informations, like the number of dereferentiations, object types involved, etc. are readily available. If a condition consists of an operation invocation, these informations are hidden behind the operation's interface. In order to look behind the scenes, the encapsulation has to be broken. The process of decomposing the implementation of a function is called *decapsulation*. This is the same basic idea that is being pursued in the revelation project [GM88]. Decapsulation opens many paths to optimization in object bases, where the mostly procedural approach makes algebraic optimization difficult. First steps in this direction were made by Kemper, Kilger and Moerkotte in [KKM91], where decapsulation is applied to determine accessed attributes in the context of function materialization, and by DeWitt and Lieuwen in [LD91] in their work about loop optimization.

To estimate the cost for evaluating a function, it is necessary to use an adequate cost model. In Section 4.2.1 we provide a simple model that counts each dereferentiation (object access) as one cost unit. Basically, this means that we assume a zero buffer and use the number of object accesses as the cost measure. The cost function $\mathcal{C}$ is used to extract this information from the implementation of the function.

4.2.1 Zero Buffer

The dereferentiation count, and thus the cost estimation, is performed by the cost function $\mathcal{C}$, that analyses the implementation of a function and extracts the necessary information. The only expression that can possibly cause expenses are so-called *path expressions* [KM90a]. Path expressions are defined as follows:

Definition (Path Expressions) *Let* $t_0, \ldots, t_n$ *be object types. An expression* $t_0.A_1.\cdots.A_n$ *is a* path expression *iff for all* i*, with* $1 \leq i \leq n$*, one of the following conditions holds:*

1. t_{i-1} *is a tuple type with attribute* A_i *of type* t_i
2. t_{i-1} *is a tuple type with attribute* A_i *of the set type* $t'_i = \{t_i\}$ *with "pseudo" attributes* $element_j$ *of type* t_i*, where* $1 \leq j \leq |A_i|$[5]

where the "pseudo" attributes denote set elements.

On the basis of path expressions, the definition of the extraction function (both for expressions and statements) runs:

Definition (Cost Function) *Let* v *be a variable,* e *an expression,* A *an attribute,* s *a statement and* $f(p_1, \ldots, p_n)$ *a (type associated or free) function with*

Expressions:

$\mathcal{C}(v)$	:=	0
$\mathcal{C}(e.A)$	:=	$\mathcal{C}(e) + 1$
$\mathcal{C}(e.f(p_1, \ldots, p_n))$	:=	$\mathcal{C}(e) + \mathcal{C}(f) + \mathcal{C}(p_1) + \ldots + \mathcal{C}(p_n)$
$\mathcal{C}(e_1 \theta e_2)$	:=	$\mathcal{C}(e_1) + \mathcal{C}(e_2)$
$\mathcal{C}(f)$	:=	$\mathcal{C}(\textbf{body})$ (provided f is defined as $f(p_1, \ldots, p_n)$ **body**;)

Statements:

$\mathcal{C}(v := e)$	:=	$\mathcal{C}(e)$
$\mathcal{C}(e_1 := e_2)$	:=	$\mathcal{C}(e_1) + \mathcal{C}(e_2)$
$\mathcal{C}(s_1;\ s_2)$	:=	$\mathcal{C}(s_1) + \mathcal{C}(s_2)$
$\mathcal{C}(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2)$	:=	$\mathcal{C}(e) + p \cdot \mathcal{C}(s_1) + q \cdot \mathcal{C}(s_2)$ p is the probability for condition e being true; $q = (1 - p)$
$\mathcal{C}(\textbf{foreach } v \textbf{ in } e \textbf{ do } s)$	:=	$\sum_{i=1}^{\text{floop}(e,s)} (\mathcal{C}(v := e.element_i) + \mathcal{C}(s))$
$\mathcal{C}(\textbf{while } e \textbf{ do } s)$	:=	$\mathcal{C}(e) + \text{wloop}(e,s) \cdot (\mathcal{C}(s) + \mathcal{C}(e))$
$\mathcal{C}(\textbf{return } e)$	:=	$\mathcal{C}(e)$

Table 3: Cost function $\mathcal{C}$

```
declare AverageTolerance: {Bearing} → float;
define AverageTolerance(BearingSet) is
        begin
            var bx: Bearing;
                sum, size: float;

            sum := 0;  size := 0;
            foreach bx in BearingSet do
            begin
                sum := sum + ord(bx.Tolerance.Class);
                size := size + 1;
            end;
            return(sum/size);
        end define AverageTolerance;
```

Figure 4: Free Operation "AverageTolerance"

parameters $p_1, \ldots, p_n$*. The cost function* $\mathcal{C}$ *is then defined as in Table 3;* floop *is the number of* **foreach***-iterations, and* wloop *the number of* **while***-iterations.*

Example: The following example illustrates the application of the decapsulation strategy with the cost estimator $\mathcal{C}$ to the function *AverageTolerance* (Figure 4).

1. $\mathcal{C}(\text{sum} := 0;\ \text{size} := 0) = 0 + 0 = 0$
2. $\mathcal{C}(\textbf{foreach} \ldots)$
$$= \sum_{i=1}^{\text{floop}} \Big(\mathcal{C}(\text{bx} := \text{BearingSet.element}_i) + \mathcal{C}(\text{sum} := \text{sum} + \text{ord}(\text{bx.Tolerance.Class})) + \mathcal{C}(\text{size} := \text{size} + 1)\Big)$$

[5]Nested sets are excluded in this definition, but can be easily added.

$$= \sum_{i=1}^{\text{floop}} \Big(1 + (0+2) + 0\Big) = 3 \cdot \text{floop}$$

"floop" can be estimated as the cardinality of the "iterated" set, unless there exists a **return** statement within the body of the loop. In this case, "floop" has to be adapted according to the probability that this **return** statement is executed. In our case, the loop iterates always through the entire set, so floop = |BearingSet|, and, thus, the cost estimate is 3 · |BearingSet|. □

Analogously to "floop," "wloop" denotes an estimate for the number of loop iterations of a **while** loop. However, it is usually not possible to get accurate figures just by considering the loop condition, because the probability changes within the body of the loop.

4.2.2 Infinite Buffer

If the object buffer is capable of storing a significant fraction of an transaction's working set, the zero buffer model will overestimate the actual cost. Looking at today's main memory sizes, the notion of an infinite buffer is not as far from reality as it looks at first glance. To simulate its behaviour, the dataflow of a function has to be traced, using a technique similar to dataflow analysis in the field of compiler construction (cf. e.g., [ASU86]). Specifically, all paths that may be traversed during the execution of an operation must be collected in order to detect multiple traversals of the same path. This set of traversed paths serves as the basis for the cost estimate. A cost model that is founded on this idea is currently under development [KMS92].

4.2.3 Sampling

Sampling, the other approach mentioned above, is just the opposite of decapsulation. It treats all conditions as black boxes and merely monitors their behaviour over time. The monitor records thus derived can be viewed as samples, and as the number of samples that have to be taken to ensure a certain accuracy does *not* depend on the cardinality of the set being sampled, the precision can be determined in advance, i.e., at compile time. The compiler allocates enough memory for the demanded number of samples, and with each run new statistics are collected.

Sampling and decapsulation should be combined: The cost function provides a rough estimate that serves as a starting point, which is backed by more and more samples taken from actual program runs. As soon as enough samples are taken to ensure the desired accuracy, the decapsulation derived data can be dropped and the optimizer may rely solely on the samples.

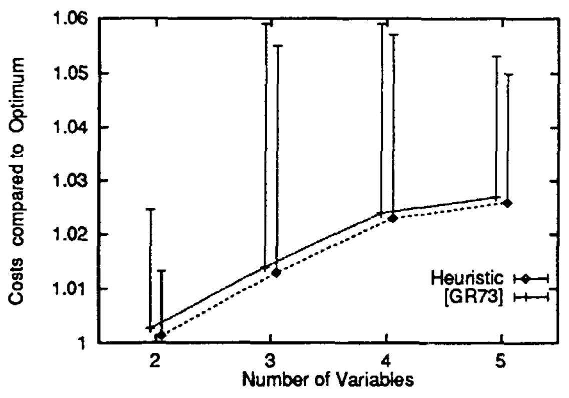


Figure 5: Comparison with the algorithm in [GR73]

A major advantage of this approach is the adaptive behaviour of the estimation. To capture the implications of index structures, the cost model for the decapsulation appoach has to be modified, and the programs that rely on the decapsulation derived data have to be recompiled. On the other hand, with sampling the effects of a new index structure become immediately apparent, and the optimizer can generate new evaluation strategies.

5 Quantitative Analysis

In order to assess the quality of our optimization algorithm, we carried out many tests; Figures 5–11 show some typical results. The underlying conditions common to each one are:

- Uniform or exponential distribution of costs (in the interval $[0, 100]$) and probabilities.
- Test expressions were derived by randomly filling in truth tables.
- Each test comprises 2500 optimized evaluation plans.

The diagram in Figure 5 shows the difference between our algorithm and the one proposed in [GR73]. For varying numbers of variables (2–5) we compare the percentage costs of the evaluation plans generated by the two heuristics with respect to the optimal evaluation plan which is normalized to 1. In addition to the average cost ratio, we show the standard deviation as errorbars. We see that our algorithm computes slightly lower evaluation plans, and the standard deviation of the factor to the optimum is better, too.

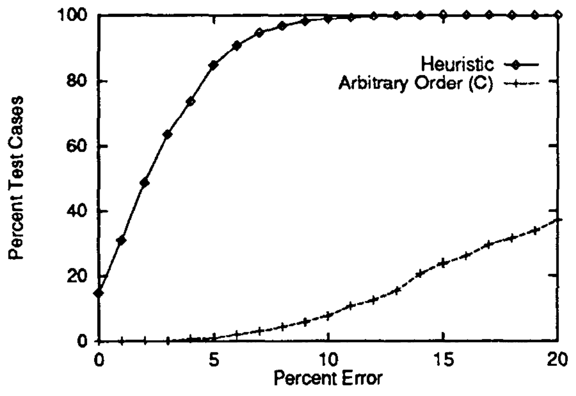


Figure 6: Quality of the Heuristic

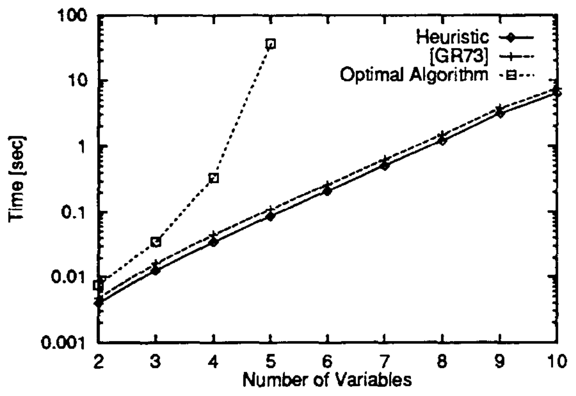


Figure 7: Run Time of the Heuristic and of the Optimal Algorithm

The second diagram (Figure 6) shows how well our optimization algorithm performs, compared with a strategy that evaluates the conditions in arbitrary order and stops as soon as the result is certain—this resembles the method that is used in C. The results for optimizing the evaluation of expressions with five variables are depicted. More than 95% of the evaluation plans yielded by the heuristic have average costs of less than 7% worse than the average cost of the optimal plan.

In Figure 7, the running times of the optimal algorithm, the heuristic from [GR73] and our heuristic are compared. All algorithms are implemented in LISP, the figures are derived from benchmarks on a Sun SPARCstation 1+ (12.5 MIPS) with 40 MB of main memory. It is evident that the generation of evaluation plans with the optimal algorithm is hardly feasible, even for few conditions, due to the super-exponential running time: Extrapolation for six conditions yields about 23.7 hours, and for seven conditions about 1522 years.

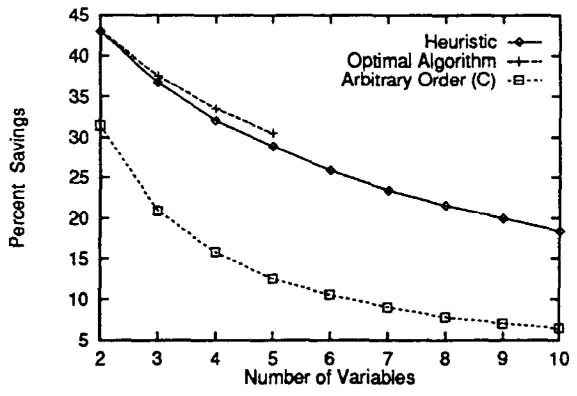


Figure 8: Achievements by Optimization: (1a) Relative Savings, Uniform Distribution of Costs and Probabilities

We see that the algorithm [GR73] has a running time of about 30% above our heuristic.

The diagrams in Figure 8 and 10 focus on the question of the possible achievements by applying the optimization algorithm on boolean expressions with increasing numbers of variables. Unfortunately, because of the extreme running time, we cannot generate the optimum for more than five variables as the measure of comparison. Therefore, in the first diagram in Figure 8, the achievements are compared against the sum of the evaluation costs of all conditions (worst case). Even expressions with ten variables can be evaluated with average cost of about 20% below worst case, whereas with arbitrary evaluation order the worst case is almost reached.

In Figure 8 we observe that the relative savings compared to the worst case decreases with increasing number of variables—this is not a peculiarity of our heuristic but is also true for the optimum. At first glance, this behaviour may appear astonishing. It is caused by the fact that a single condition has less impact on the outcome of a boolean expression as the number of variables increases. For two-variable expressions it is more likely that one variable is irrelevant than for a ten-variable expression that five variables are irrelevant. The opposite is true for absolute savings: The absolute savings diagram (Figure 10) shows that the difference of optimized cost and worst case cost *increases* with increasing number of variables, whereas the arbitrary evaluation strategy yields only a constant absolute difference.

Note that all distributions in the benchmarks explained so far (probabilities, costs, truth tables) are uniform. Figures 9 and 11 depict the relative resp. absolute savings for an exponential distribution of costs

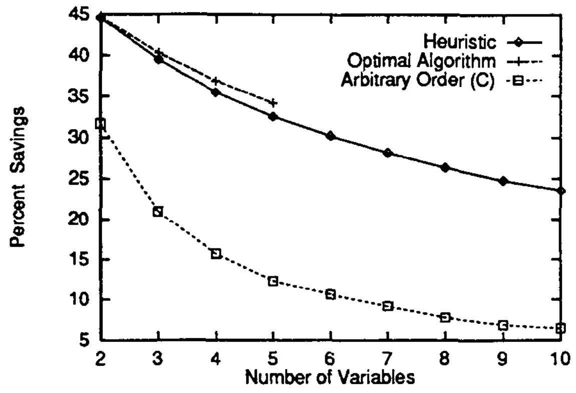


Figure 9: Achievements by Optimization: (1b) Relative Savings, Exponential Distribution of Costs and Probabilities

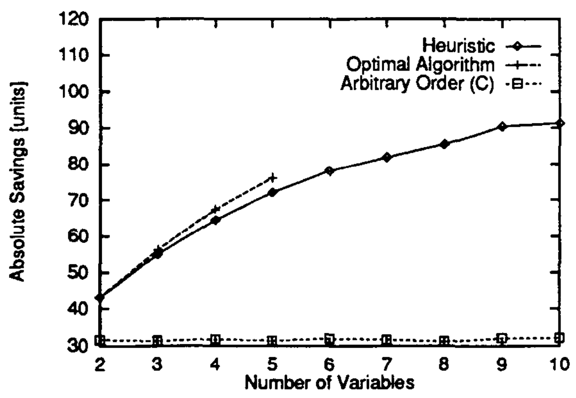


Figure 10: Achievements by Optimization: (2a) Absolute Savings, Uniform Distribution of Costs and Probabilities

and probabilities.[6] As greater differences of costs and probabilities become more likely with this distribution, relative as well as absolute savings increase. The more the distributions are skewed, the higher is the optimization potential. On the other hand, we found that this potential is minimal if a "one-point" distribution (i.e., all costs 50 units, all probabilities 0.5) is used as the basis for the benchmarks.

These results suggest that our optimization scheme is especially well suited for applications in object bases, where this situation (great differences) is quite common (e.g., cost of main memory vs. disk access). We can also see that the heuristic performs asymptotically as well as the optimum—which, however, cannot be determined for more than five variables.

[6]Probability density function $p(x) = \frac{1}{50}e^{-\frac{1}{50}x}$ (costs)

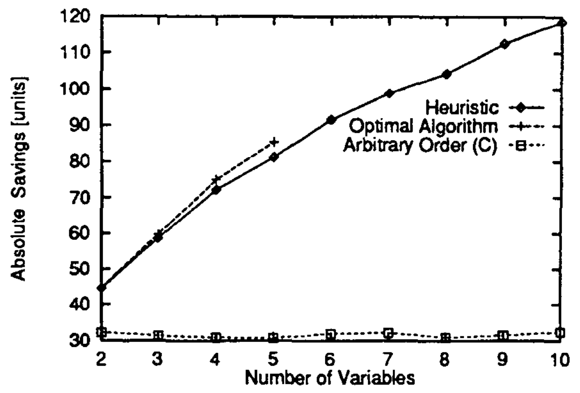


Figure 11: Achievements by Optimization: (2b) Absolute Savings, Exponential Distribution of Costs and Probabilities

6 Conclusion

In this paper we addressed the problem of optimizing boolean expressions in object-oriented databases. We motivated the need for this optimization by way of a "real" engineering example. There exists a known algorithm for deriving the optimal evaluation plan; unfortunately this algorithm exhibits terrible running times which makes it infeasible for complex boolean expressions with more than four or five variables. Therefore, heuristics with (far) lower computational complexity are required. We developed one such heuristic which—in a quantitative analysis—proved to be superior to all other known heuristics in both respects, quality of the generated evaluation plan as well as performance, i.e., running time. In cases where the optimal algorithm could still be employed our heuristic proved to generate results comparable to the optimum; for example, the outcome of the heuristic was within a 5% error range to the optimum in 85% of the cases and in 99% of the cases within a 10% error range (for randomly generated boolean expressions with five variables). The comparatively low execution time of our heuristic facilitates its utilization not only at compile time but even at run time in order to rearrange the evaluation plan based on dynamically derived cost and selectivity estimates.

Aside form developing the heuristics, we had to address the problem of estimating these input parameters to the optimizer, i.e., selectivity and cost estimates of the terms constituting the boolean expression. For selectivity estimation we heavily built upon the well-known relational techniques whereas the cost estimation required a new approach—called *decapsulation*—because of the possibility of invoking functions in boolean expressions in object bases.

The optimization algorithms as well as the decapsulation method were implemented in LISP. In order to perform the quantitative analysis we implemented all known heuristics and the optimal algorithm. The analysis presented in this paper is based on randomly generated boolean expressions and their associated cost and selectivity estimates. Currently we are working on a quantitative assessment of our optimization method based on realistic boolean expressions derived from various usage profiles of (mostly) mechanical engineering applications running on our object base system GOM.

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

We thank H. Schreiber for carrying out the implementation and extensive benchmarking of the various optimization algorithms.

References

[ASU86] A. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[Chr83] S. Christodoulakis. Estimating record selectivities. *Information Systems*, 8(2):105–115, 1983.

[Dem80] R. Demolombe. Estimation of the number of tuples satisfying a query expressed in predicate calculus language. In *Proc. International Conference on Very Large Data Bases*, pages 55–63, 1980.

[Fed84] J. Fedorowicz. Database evaluation using multiple regression techniques. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 70–76, 1984.

[GM88] G. Graefe and D. Maier. Query optimization in object-oriented database systems: a prospectus. In K. R. Dittrich, editor, *Advances in Object-Oriented Systems*, Lecture Notes in Computer Science 334, pages 358-363. Springer, 1988.

[GR73] S. Ganapathy and V. Rajaraman. Information theory applied to the conversion of decision tables to computer programs. *Communications of the ACM*, 16:532–539, 1973.

[KK85] N. Kamel and R. King. A model of data distribution based on texture analysis. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 319–325, 1985.

[KKM91] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 258–268, 1991.

[KM90a] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 364–374, 1990.

[KM90b] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 290–301, Brisbane, Australia, Aug 1990.

[KMS92] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object bases. Technical report, RWTH Aachen, 1992. (in preparation).

[LD91] D. F. Lieuwen and D. J. DeWitt. Optimizing loops in database programming languages. Computer Sciences Technical Report 1020, University of Wisconsin-Madison, Madison, 1991.

[LNS90] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 1–11, 1990.

[Lyn88] C. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distribution of column values. In *Proc. International Conference on Very Large Data Bases*, pages 240–251, 1988.

[MD88] M. Muralikrishna and D.J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 28–36, 1988.

[Pol65] S. L. Pollack. Conversion of limited entry decision tables to computer programs. *Communications of the ACM*, 8(11):677–682, 1965.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 256–276, 1984.

[RS66] L. T. Reinwald and R. M. Soland. Conversion of limited entry decision tables to optimal computer programs I: minimum average processing time. *Journal of the ACM*, 13(3):339–358, 1966.

[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 International Conference on Management of Data*, pages 23–34, 1979.

[Sch89] W. G. Schneeweis. *Boolean Functions with Engineering Applications and Computer Programs.* Springer, 1989.