Proceedings of Fith International Conference on Deductive and Object–Oriented Databases (DOOD'97), December 8–12, 1997, Montreux, CH

Rule–Based Generation of Logical Query Plans with Controlled Complexity

Karl Aberer, Dunren Che, and Klemens Böhm
GMD–IPSI, Dolivostr. 15, 64293 Darmstadt, Germany
E–mail: {aberer, che, kboehm}@darmstadt.gmd.de

Abstract

Rule–based query optimizers are recognized as particularly valuable for extensible and object–oriented database management systems by providing a high flexibility in adapting query optimization strategies to nonstandard application needs. On the other hand rule–based optimizers are problematic with regard to run–time behavior for more complex queries as the generation of query plans based on a declarative rule base tends to be difficult to control. In this paper we show that this is not a fundamental problem of rule–based optimizers, but rather a question of careful design of the rule system. We exemplify this for one fundamental optimization problem, namely join enumeration for linear queries. There, a rule–based optimization strategy can achieve the theoretically optimal complexity. The design principles used to achieve this have been derived from and are used in the design of the VODAK query optimizer developed at GMD–IPSI.

1 Introduction

There exists a broad spectrum of potential (nonstandard) applications of DBMS, which impose very different demands on database systems. Thereby much of the leading research work of the database community in the recent years focuses on investigating extensible database systems [Batory+88, Becker+92, Blakeley+93, Finance+94, Graefe+87, Graefe+93, Guting88, Haas+90, Ozsu+95, Stonebraker+86]. Several research groups have introduced the extensibility concept to surmount the limitations of traditional RDBMSs, e.g., Starburst [Haas+89, Haas+90], Gral [Becker+92], EDS ESPRIT [Finance+94]. Extensibility is even more necessary for object–oriented and object–relational database systems. There are many more alternatives (such as storage structure, object query model, object algebra, transformation rules, cost model, search strategy, and so on) in the OODBMS realm. As a consequence, OODBMSs must be more open, extensible, and customizable.

Modularity and hence extensibility of the query processor (optimizer) is probably the most challenging aspect when constructing an extensible database system since it is the most complex component of a DBMS [Becker+92]. As the first effort towards a modular query optimizer, applying rule–based techniques to query optimization was originally suggested by Freytag [Freytag87], while conventionally the query optimizer is implemented in a *hard–coded* or *hard–wired* manner [Ozsu+95].

Compared with hard–coded approaches, rule–based techniques bear the following advantages:

(1) A straightforward advantage of the rule–based approach is that it allows the definition of new transformation rules, which are frequently required by nonstandard applications. Thus it is possible to easily change and extend the set of possible transformations and underlying implementations, i.e., methods.

(2) The rule–based approach ensures modularity, even if it is applied in the transformation phase of an optimizer only, since transformations alone represent a major portion of the body of a query optimizer. Actually, the application of the rule–based approach to query optimization has already gone far from transformation/term–rewriting, as in [Finance+94] and [Becker+92] where (meta–)rules were used to uniformly model multiple search strategies also.

(3) The rule–based approach is also highly appropriate for research prototype development due to the great convenience it provides for experimenting with different alternatives, e.g., different transformations, improved statistics and cost models, different algebraic operators and implementations of algorithms.

Regardless of the attractiveness of the rule–based approach, e.g., extensibility and declarative nature, it may suffer from several serious problems which are likely to dramatically reduce the potential interest in it. The design of rule systems for query optimizers typically starts in an *ad–hoc* manner by encoding a set of transformation rules that are derived from valid equivalences of the query algebra. Such transformation rules consist of an expression pattern that is transformed into a different equivalent expression, possibly restricted by conditions on the expression parameters. Due to the heuristic nature of this approach the rule application process in enumerating equivalent expressions is not understood and thus also the impact of selecting different heuristics is difficult to judge. Among others, there are two important issues to consider: many uninteresting expressions may be generated; and the same expression may be produced in many different ways for many times. That means a great deal of futile work is performed during optimization. Thus the resulting optimizers tend to have poor performance as compared to hard–coded optimizers with well understood algorithmic behavior. One methodology at this stage is experimentation with heuristics to reduce the number of rule applications. This is however an unsatisfying approach as long as no real understanding is gained on the structure of the rule transformation process.

In this paper we show that such an understanding can be gained and that there are no principal obstacles in developing efficient rule–based optimizers. The goal of this paper is to show that it is possible, based on an intuitively appealing heuristic, to achieve complexity results that correspond to the inherent complexity of exhaustive search strategies by designing appropriate rule systems. Thus, the rule system does not introduce any unnecessary overhead. Although this might require an increased design effort, as compared to the design of hard–coded optimizers, this has to be outweighed against the advantages of ease of implementation (when the rule system has been designed implementation is trivial) and the additional flexibility, modularity and declarativity, which is in particular useful in the context of extensible database management systems.

We will show for one important and well analyzed class of queries, namely linear queries, that a rule–based optimizer can achieve the same performance (from a complexity viewpoint) as a typical hard–coded optimizer does. Linear queries are interesting since they are a relational equivalent to path expressions of object–oriented models, i.e., a path expression is split up into a linear query in SQL. This class is well analyzed so far, and there exists a join enumeration algorithm with complexity $O(k^3)$ for k join predicates [Ono+90] which is used in the Starburst optimizer.

This paper is based on our work on investigating and developing a rule–based optimizer for the object–oriented database management system VODAK [VODAK95] relying on an algebraic representation of object–oriented queries that are specified in an OQL–style query language [Cattell93]. To this extent the Volcano Optimizer Generator [Graefe+93] has been used. This work is targeted on supporting semantic optimization rules in the context of advanced applications, like document management and biomolecular databases [Aberer+95a, Aberer+95b], at the level of the logical query algebra. We distinguish general–purpose rules from application–specific rules, which are the basis of application–specific semantic query optimization. In the course of our investigation we have recognized that as a sound basis for this kind of algebraic semantic query optimization an efficient general–purpose rule system is strongly required. In the rest of this paper we reflect those findings when developing the general–purpose rule system consisting of application–independent rules and some generic design principles and complexity results that can be derived from this work.

The paper is organized as follows. In the next section we elaborate on how the need for semantic optimization rules guides the overall transformation strategy. Then in Section 3 we provide a short presentation of the logical algebra and a high level description of the adopted transformation algorithm. Section 4 addresses our complexity results on join enumeration of linear queries under the chosen strategy. After reviewing some related work in Section 5, we conclude this paper in Section 6 with an outlook on future work.

2 Rationale of the Approach

When optimizing object–oriented queries we have two goals in mind:

1. the query should be optimized with regard to all standard (object–) relational optimizations (ordering of selections and joins, pull–down of selections etc.)
2. the query should be optimized semantically by replacing expensive functions by potentially better alternatives (application–specific index structures, alternative paths etc., examples can be found in [Aberer+95a])

Both types of optimizations can be performed by algebraically transforming the query. In the following we illustrate by means of an example what difficulties can occur when doing this. Assume the query is given by

$$\begin{aligned}&\mu_{a6=p4(a0)}(\\&\quad\sigma_{a5=c2}(\mu_{a5=g(a4)}(\mu_{a4=p3(a0)}(\\&\qquad\sigma_{a3=c1}(\mu_{a3=f(a1,a2)}(\mu_{a2=p2(a0)}(\mu_{a1=p1(a0)}(\Gamma_{a0,C}))))))))\end{aligned}$$

where μ is a materialization operator, in the following called map operator, for expression evaluation, σ is a selection operator, and Γ is an access operator for getting the extension of a class (precise definitions are given in the next section). The references a_i denote materialized attributes. In the following we also denote this query shortly by

$$\mu_6 (\sigma_2 (\mu_5 (\mu_4 (\sigma_1 (\mu_3 (\mu_2 (\mu_1 (\Gamma))))))).$$

Furthermore assume that there exists a method *m* such that *m(x) = f(p1(x), p2(x))*. Then two possible optimizations need to be considered:

1. commuting the selections
2. replacing $\mu_3 (\mu_2 (\mu_1 (\Gamma)))$ with $\mu_{a3=m(a0)} (\Gamma_{a0,C})$

Usually in relational query optimization the select commutativity rule can cover the ordering of the selections. The presence of the materialization operators complicates things. In order to commute the two selections different approaches can be considered:

1. use rules for map–select and map–map commutativity. However this leads to an explosion in the number of possible transformations and leads to many useless byproducts, of which we give a few examples:
   $\mu_6 (\sigma_2 (\mu_5 (\mu_4 (\sigma_1 (\mu_3 (\mu_1 (\mu_2 (\Gamma)))))))$
   $\sigma_2 (\mu_5 (\mu_4 (\sigma_1 (\mu_3 (\mu_2 (\mu_1 (\mu_6 (\Gamma)))))))$
   (μ_6 can be anywhere)
   $\sigma_1 (\sigma_2 (\mu_5 (\mu_4 (\mu_3 (\mu_2 (\mu_1 (\mu_6 (\Gamma)))))))$
   (and most permutations of the μ operators)
   The main problem is that commutativity rules can be applied in both directions, and from the information available when matching a rule to a subexpression the promising direction cannot be predicted. Even worse, the same expressions are generated in many different ways leading to even more unnecessary work.
2. introduce "metarules" like
   $\sigma_1 (\mu_1{}^* (\sigma_2 (\mu_2{}^* (E)))) \Rightarrow \sigma_2 (\mu_2{}^* (\sigma_1 (\mu_1{}^* (E))))$
   where $\mu_i{}^*$ stands for an arbitrarily long sequence of map operators. This is however a substantial extension of the concepts of existing rule transformation engines. In addition, this alternative does not prevent from the combinatorial explosion described in item 1.
3. Extend the definition of map operators, such that they can contain more complex expressions in their parameters. However then in order to match the "semantic transformations" we have either to provide for the collapsing/uncollapsing of a complex map operator into the existing atomic ones, or the pattern matching process has to be performed at the expression parameter level and no longer at the level of algebraic expressions. While the first alternative does not really help with regard to the explosion of the number of possible transformations, the second alternative is expensive to implement and contradicts our original objective to use a rule–based query optimizer to do the semantic transformations.

So we approached the problem from a different side. Experimentation has shown that a lot of efficiency can be gained if by "directing" the rule application more control is imposed on the transformation process. The extreme case of a directed rule engine is expression normalization, which performs simplifications of an expressions only. In fact, some rule–based optimizers use this approach. On the other hand normalization does not generate alternative solutions. Thus we targeted towards the combination of strongly directed transformations with generation of alternative expressions. When transforming in a directed manner the starting expression becomes an important matter. We illustrate this for our example. Using the join operator *x*, as a starting representation of the query we choose the expression

$$\mu_6 (\sigma_2 (\mu_5 (\mu_4 (\Gamma))) \, x \, \sigma_1 (\mu_3 (\mu_2 (\mu_1 (\Gamma)))))$$

This expression by itself is definitely not optimal. However, we need now to apply commutativity rules only in one direction, namely for moving selection and map operators upwards and subsequently idempotency rules for eliminating redundant subexpressions. This appears at first to be a quite *counter–intuitive* approach, e.g. because selections are pushed up instead of pulled down. Indeed, if the rules for moving operators upwards are applied to an expression that is in a "reasonably optimal" form (e.g. the linear order of the selections given at the beginning), such a restricted rule set does not allow to find an equivalent superior alternative. A possible interpretation of the approach is that it corresponds to a (deterministic) hill–climbing method, i.e. the transformations chosen lead always directly (or indirectly) to improvements. Thus, the starting point should typically be non–optimal in order to avoid being captured in a local optimum.

Interestingly – and this is the focus of this paper – it turned out that this heuristic is not only beneficial for the transformation of query expressions in the presence of materialization operators, but that under this strategy classical relational query optimization goals, like selection and join ordering, can be achieved efficiently. This opens the possibility to combine algebraic semantic optimizations with standard query transformations, and achieve an efficient behavior at the same time. After introducing the optimization framework in the next section we give such a specific result for the problem of join enumeration for linear queries in section 4.

3 The Transformation of Logical Query Expressions

3.1 The Logical Algebra

Queries are mapped to logical algebra expressions. The operators of the logical as well as the physical query algebra are applied to complex values of type $\{[a_1\colon D_1, ..., a_n\colon D_n]\}$ where $D_1,...,D_n$ are complex data types. We assume that the record components are unordered. Operator arguments of this type are denoted by S, S_1 and S_2. The operator parameters are represented as subscripts of the operators. We define

$$ref(S) := \{a_1,...,a_n\} \text{ for } Type(S)=\{[a_1\colon D_1, ..., a_n\colon D_n]\}.$$

and refer to a_1, ..., a_n as the references of S. In the following v_i denotes a value of type D_i, Θ is one of the boolean binary operations on built–in data types (e.g., ==, !=, <, >, IS–IN, IS–SUBSET), and C is a class name.

The principal operators of the logical algebra we need for the purposes of this paper are as follows.

1. Access to classes:

$$\Gamma_{a,C} := \{ [a: o, a_1: o.attr_1, ..., a_n: o.\ attr_n] \mid o \in extension(C) \}$$

where a $a \notin ref(extension(C)) = \{ a_1,...,a_n \}$, C is a class name, and $attr_1, ..., attr_n$ are immediately materialized attributes of the objects (e.g. values of atomic data types). We assume attribute names are globally unique for all class types.

2. Materialization of expressions (map operator):

$$\mu_{a := expr(a1,\ a2,\ ...,ak)} (S) :=$$
$$\{[a: v, a_1: v_1, ..., a_n: v_n] \mid [a_1: v_1, ..., a_n: v_n] \in S \wedge v = expr(v_1, v_2, ..., v_k) \}$$

where $ref(S) = \{ a_1,...,a_n \}$ and $a \notin ref(S)$, $k \leq n$, where *expr* is a scalar expression. In the current implementation the following expressions are supported without nesting: constant values, property access, elementary operations on data types and method calls. The references are related one–to–one with the generating expressions. If we do not need the detailed structure of the expression we also will denote this operator shortly by μ_a. For set–valued expressions we provide a corresponding operator that performs immediate unnesting:

$$\nu_{a := expr(a1,\ a2,...,ak)} (S) :=$$
$$\{[a: v, a_1: v_1, ..., a_n: v_n] \mid [a_1: v_1, ... ,a_n: v_n] \in S \wedge v \in expr(v_1, v_2, ...,v_k) \}$$

where $ref(S) = \{ a_1,...,a_n \}$, $a \notin ref(S)$, $k \leq n$ and *expr* is a set-valued expression.

3. Selection:

$$\sigma_{a1\ \Theta\ a2} (S) :=$$
$$\{[a_1: v_1, ..., a_n: v_n] \mid [a_1: v_1, ..., a_n: v_n] \in S \wedge (a_1\ \Theta\ a_2) = TRUE \}$$

where $ref(S) = \{ a_1,...,a_n \}$, and both a_1 and a_2 belong to $ref(S)$.

4. Natural join:

$$S_1 \text{ x } S_2 := \{[a_1: v_1, ..., a_i: v_i, ..., a_k: v_k, ..., a_n: v_n] \mid$$
$$\exists [a_1: v_1, ..., a_i: v_i, ..., a_k: v_k] \in S_1 \wedge \exists\ [a_i: v_i, ..., a_k: v_k, ..., a_n: v_n] \in S_2\}$$

where $ref(S_1 \text{ x } S_2) = ref(S_1)\ U\ ref(S_2)$ for $ref(S_1) = \{a_1, ... , a_k\}$, $ref(S_2) = \{a_i, .., a_n\}$ and $1 \leq i \leq k+1 \leq n$ (if $i=k+1$ this is the Cartesian product).

3.2 Canonical Equivalences

With the above definition of the algebra several equivalences hold that can be used for the generation of alternative expressions in query optimization. It has to be pointed out that a rule–based optimizer using these equivalences directly and unrestricted as transformation rules will not work efficiently as will be illustrated later.

Select commutativity

$$\sigma_1(\sigma_2(E)) \Leftrightarrow \sigma_2(\sigma_1(E))$$

Join associativity

$$(E_1 \; x \; E_2) \; x \; E_3 \Leftrightarrow E_1 x \; (E_2 x \; E_3)$$

Join commutativity

$$E_1 \; x \; E_2 \; \Leftrightarrow \; E_2 \; x \; E_1$$

Select–join commutativity

$$\sigma_{a1\Theta a2}(E_1) \; x \; E_2 \Rightarrow \sigma_{a1\Theta a2}(E_1 \; x \; E_2)$$

$$\sigma_{a1\Theta a2}(E_1 \; x \; E_2) \Rightarrow \sigma_{a1\Theta a2}(E_1) \; x \; E_2 \qquad \text{if } \{a1, a2\} \subseteq ref(E_1)$$

Idempotency

$$E_1 \; x \; E_2 \Leftrightarrow E_1 \qquad \text{if } \; ref(E_2) \subseteq ref(E_1) \text{ and } E_2 \text{ free of selections}$$

Map–select commutativity

$$\mu_a(\sigma_{a1\Theta a2}(E)) \Rightarrow \sigma_{a1\Theta a2}(\mu_a(E))$$

$$\sigma_{a1\Theta a2}(\mu_a(E)) \Rightarrow \mu_a(\sigma_{a1\Theta a2}(E)) \qquad \text{if } \; a \notin \{a1, a2\}$$

Map–join commutativity

$$\mu_{a:=expr(a1,\dots,an)}(E_1 \; x \; E_2) \Rightarrow \mu_{a:=expr(a1,\dots,an)}(E_1) \; x \; E_2 \quad \text{if } \{a1, \dots, an\} \subseteq ref(E_1)$$

$$\mu_a(E_1) \; x \; E_2 \Rightarrow \mu_a(E_1 \; x \; E_2)$$

Map–map commutativity

$$\mu_{a:=expr(a1,\dots,an)}(\mu_b(E)) \Rightarrow \mu_b\left(\mu_{a:=expr(a1,\dots,an)}(E)\right) \qquad \text{if } b \notin \{a1, \dots, an\}$$

3.3 Transformation Algorithm

In the optimization process Volcano proceeds in two phases. In the first phase all possible logical query plans are generated by a transformation algorithm. In the second phase a search algorithm finds the optimal physical plan starting from the set of generated logical plans. In the following we consider the first phase.

The transformation algorithm for logical query expressions makes heavy use of the concept of equivalence class. Expressions that are equivalent under the above equivalences are collected in equivalence classes. During the transformation process expressions are not stored explicitly but with the arguments of the root operator replaced by the corresponding equivalence class of the argument expression.

Definition: An *equivalence class* $\mathfrak{E}$ is the set of expressions that are equivalent under a given set of equivalences. A *representative* of an equivalence class $\mathfrak{E}$ is an expression $E_{rep} = op(\mathfrak{E}_1,\dots,\mathfrak{E}_n)$, $n \geq 0$, corresponding to an expression $E=op(E_1, \dots, E_n) \in \mathfrak{E}$, where $E_1 \in \mathfrak{E}_1, \dots, E_n \in \mathfrak{E}_n$.

This compressed representation of expressions allows to store the generated expressions in an extremely space–efficient form. Of course, for the sake of pattern matching in rule applications, the arguments of expressions have to be temporarily replaced by all possible representatives up to the depth of the rule pattern expression. In order to apply this concept with Volcano we had to slightly adapt the internal representation of expressions. In the original Volcano system instead of replacing the top level arguments of an expression by equivalence classes they were replaced by canonical members of these equivalence classes. This leads to intricate reorganizations of the representations of internal expressions when equivalence classes which are identified to be identical as a result of rule applications have to be merged and thus the canonical members are changed. With our modified representation scheme these difficulties do not occur.

The following is a high level description of the transformation algorithm:

```
transform(expr) :={
for each argument a of expr
     transform(a)
if expr is not contained in an equivalence class
     { generate new equivalence class e(i);
     insert expr into e(i) }
for each rule r
     { if r applicable
     (by expanding arguments through representatives !)
          { generate new expression e by applying r
          insert e into equivalence class of expr
          if e found in a different equivalence class
               {merge equivalence classes}
          else transform(e); }
     }
}
```

4 Join Enumeration for Linear Queries

Linear queries are an important class of queries, for which a well known result on the complexity of join enumeration exists [Ono+90]. In a linear query classes are joined (linearly) by a sequence of join predicates. Given a sequence of *k* classes as in Figure 1 there exists (a conjunction of) *k–1* predicates $pred_i(C_i, C_{i+1})$.

Linear queries correspond closely to path expressions and are thus also particularly interesting in the context of object–oriented query optimization. We show that given an appropriate input representation of the linear query and an appropriate rule system, the

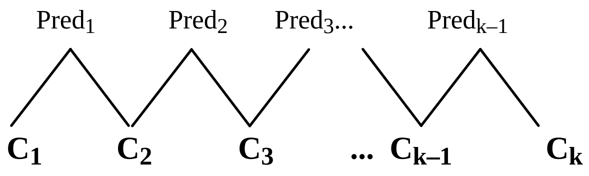


Figure 1. Classes and predicates in linear queries

enumeration of all join expressions can be performed with the theoretically lowest possible cost by exactly enumerating the $O(k^3)$ alternative join expressions. The underlying assumption is that inner and outer joins are not distinguished and Cartesian products are not allowed, see [Ono+90]. The Starburst optimizer uses an hard–coded join enumeration algorithm based on dynamic programming to achieve this complexity.

In the following we introduce an algebraic representation of linear queries that will be used as input representation for the transformation algorithm.

Let k classes be used in the query. Then we define the following expressions.

$$T_{ii} = \Gamma_{ai,Ci}$$

C_i is a class, a_i is a newly introduced reference to the object identifiers from class C_i, $i=1, ...,k$. For simplicity, we assume that selection predicates use only immediately materialized attributes. Using materialization operators does not principally change the results but complicates the presentation of the proof considerably. Next we introduce the representation of the single selection predicates.

$$T_{ii+1} = \sigma_{predi} \; (T_{ii} \; x \; T_{i+1i+1} \;),$$
$$S_{ii+1} = T_{ii} \; x \; T_{i+1i+1}, \; i = 1,...,k–1,$$

σ_{predi} is a selection based on predicate $pred_i$ which uses references in $ref(T_{ii})$ and $ref(T_{i+1i+1})$. Next we introduce expressions that extend linear queries.

$$T_{ij} = T_{ij–1} \; x \; T_{j–1j}, \; 0<i<j–1<k$$

Then we define the following equivalence classes:

$\mathcal{T}_{ij}$ is the class of all expressions equivalent to T_{ij}, $1 \leq i \leq j \leq k$

$\mathcal{S}_{ij}{}^n$ is the class of all expressions equivalent to $T_{in} \; x \; T_{n+1j}$, $1 \leq i \leq n<j \leq k$

Intuitively speaking the class $\mathcal{S}_{ij}{}^n$ corresponds to expressions where $pred_n$ is missing from an expression in class $\mathcal{T}_{ij.}$ For the equivalence classes we define the following representatives:

$T_{ij}{}^0$ is a representative of class $\mathcal{T}_{ij}$ of the form $\mathcal{T}_{ij–1} \; x \; \mathcal{T}_{j–1j}$, $i<j–1$

$S_{ij}{}^n$ is a representative of class $\mathcal{S}_{ij}{}^n$ of the form $\mathcal{T}_{in} \; x \; \mathcal{T}_{n+1j}$, $i \leq n< j$

$T_{ij}{}^n$ is a representative of class $\mathcal{T}_{ij}$ of the form $\sigma_n \; (\mathcal{S}_{ij}{}^n)$, $i \leq n< j$

$S_{ij}{}^{n0}$ is a representative of class $\mathcal{S}_{ij}{}^n$ of the form $\mathcal{S}_{ij–1}{}^n \; x \; \mathcal{T}_{j–1j}$, $i \leq n < j–1$,
respectively of the form $\mathcal{T}_{ij–1} \; x \; \mathcal{S}_{j–1j}{}^{j–1}$ for $i<n = j–1$

The goal of the expression transformation is to enumerate all possible join sequences. For our case this means all expressions of type $T_{ij}{}^n$ have to be generated, since they represent all possible join orders for the linear query. For illustration purposes, we give in Figure 2 all join orders for a linear query in the case k = 4 .

Next we introduce the rule set for efficient enumeration of all expressions of type $T_{ij}{}^n$. An interesting design criteria for rule systems is the question whether the rules are applied based on information on the expression itself only, or also require context information. It turns out that context information is required. In particular information on the history of how an expression has been generated by rules is required to avoid un-

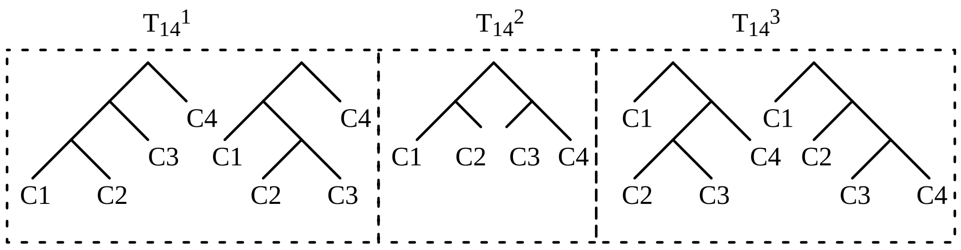


Figure 2. Join orders for linear queries for k=4.

productive rule applications. Note that the rules are all specializations of the general transformation rules from Section 3.2.

Rule R_1^l: $\sigma_1(E_1) \times \sigma_2(E_2) \Rightarrow \sigma_1(E_1 \times \sigma_2(E_2))$, if $ref(E_1) \cap ref(E_2) \neq \emptyset$ and the expression this rule is applied to has not been generated as the result of an application of this rule, including subexpressions. Analogously rule $\mathbf{R_1^r}$ for the right hand side is defined.

Rule R_2^r: $E_1 \times (E_2 \times E_3) \Rightarrow (E_1 \times E_2) \times E_3$, if $ref(E_1) \cap ref(E_2) \neq \emptyset$ and $ref(E_2) \cap ref(E_3) = \emptyset$. Analogously rule $\mathbf{R_2^l}$ is defined for $(E_1 \times E_2) \times E_3$.

Rule R_3^r: This rule comes in two versions: rule $\mathbf{R_3^{rl}}$ is given as $E_1 \times (E_2 \times E_3) \Rightarrow E_1 \times E_3$ if $ref(E_2) \subseteq ref(E_1)$ and E_2 is free of selection operators, and rule $\mathbf{R_3^{rr}}$ is given as $E_1 \times (E_2 \times E_3) \Rightarrow E_1 \times E_2$ if $ref(E_3) \subseteq ref(E_1)$ and E_3 is free of selection operators. Analogously rule $\mathbf{R_3^l}$ is defined for $(E_1 \times E_2) \times E_3$. These rules are normalization rules, i.e., if an expression is generated by means of these rules no further transformations are performed with this expression. In other words, for all other rules the additional condition holds that they are not applied if the expression has been generated by these normalization rules.

This rule system has the following property:

Lemma: For the following expressions no rules are applicable:

$E_1 \times E_2$, if $ref(E1) \cap ref(E_2) = \emptyset$

$\sigma(E)$, for any expression E

The lemma shows that only the representatives T_{ij}^n and S_{ij}^n can be used to enable the matching of rules. Using the other representatives never leads to possible rule applications. This property is useful in the proof of the following theorem.

Theorem: Under the assumption that the transformation algorithm is applied using the above rules to the representative expression T_{1k}^0, $k>1$, of class $\mathcal{T}_{1k}$ the following property holds:

For each equivalence class $\mathcal{T}_{ij}$, $1 \leq i<j-1<k$, the representatives T_{ij}^n and T_{ij}^0 are produced for $i \leq n < j$. For each equivalence class $\mathcal{S}_{ij}^n$, $1 \leq i<j-1<k$, the representatives S_{ij}^n and S_{ij}^{n0} are produced for $i \leq n < j$. No other expressions are generated, except of the trivial expressions T_{ii}, for $1 \leq i \leq k$, T_{ii+1}, S_{ii+1}, for $1 \leq i<k$.

This theorem leads immediately to the main result of this paper.

Corollary: All possible join sequences are generated for the linear query during rule transformation. The number of expressions generated during the transformation process is $O(k^3)$.

Proof: All expressions of type $T_{ij}{}^{n}$ are generated. Thus we have to analyze the complexity. We accomplish this straightforwardly by counting the total number of generated expressions. The number of expressions of form $T_{ij}{}^{0}$ is

$$\sum_{i=1}^{k-2}\sum_{j=i+2}^{k} 1 = \frac{1}{2}(k-2)(k-1)\,;$$

the number of expressions of form $T_{ij}{}^{n}$, $S_{ij}{}^{n}$ and $S_{ij}{}^{n0}$ is

$$3\times\sum_{i=1}^{k-1}\sum_{j=i+1}^{k}(j-i) = \frac{k^3}{2}-\frac{7k}{2}+3\,;$$

the number of expressions of form T_{ii}, T_{ii+1}, and S_{ii+1} is $3k-2$. Thus the total sum is $\frac{k^3}{2}+\frac{k^2}{2}-2k+2$. □

The proof of the theorem is given inductively by a detailed analysis of all possible transformation steps that can be performed at each level k starting from expression $T_{1k}{}^{0}$. An illustration of the transformations that are performed starting from $T_{1k}{}^{0}$ is given in Figure 3. For space limitations we omit the detailed analysis of all the possible steps. For ease of understanding, an illustration of sample representations and transformations of query representatives as used in the proof of the theorem is provided in the appendix of this paper.

We have simulated the transformation process using the computer algebra system Mathematica [Wolfram88] which is more flexible to handle for experimental purposes than the Volcano system. The simulations have verified experimentally the stated re-

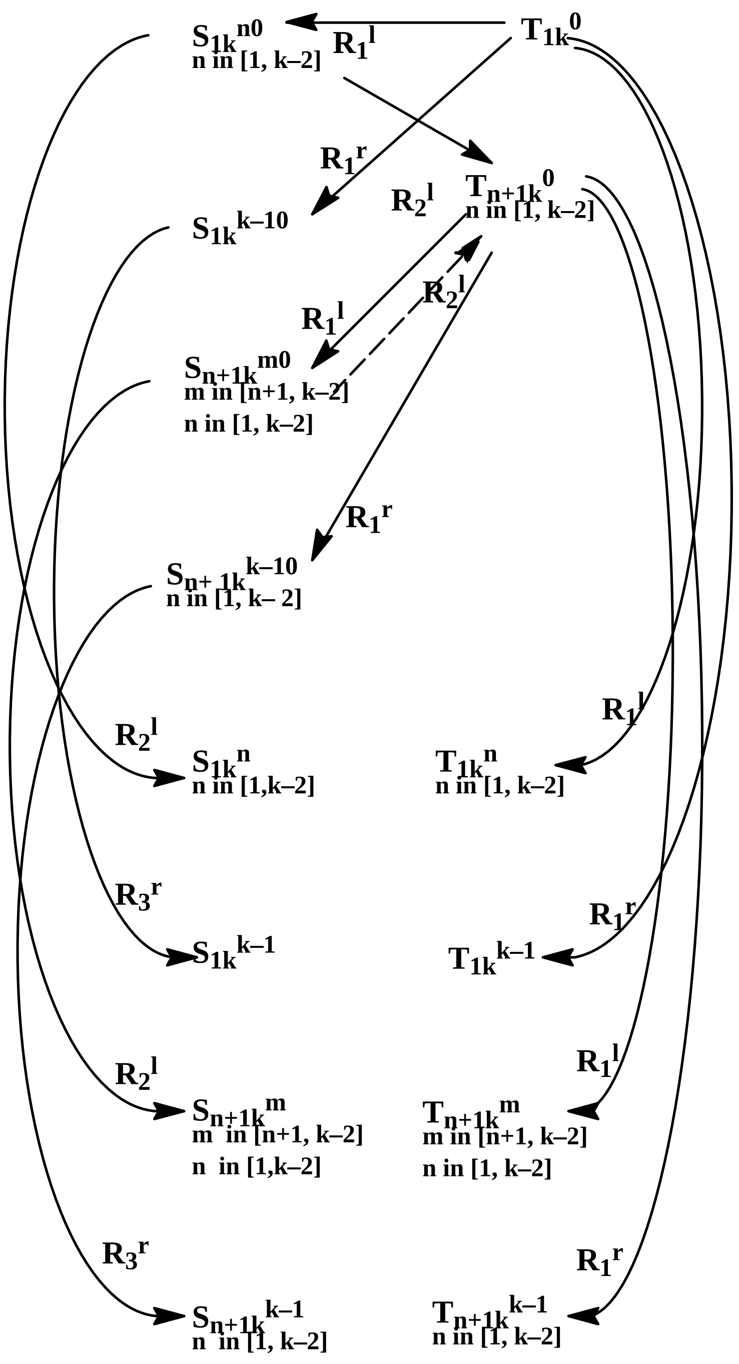

S_{1k}^{n0} n in [1, k–2]
T_{1k}^{0}
R_1^l
R_1^r
T_{n+1k}^{0} n in [1, k–2]
S_{1k}^{k-10}
R_2^l
R_1^l
R_2^l
S_{n+1k}^{m0} m in [n+1, k–2] n in [1, k–2]
R_1^r
S_{n+1k}^{k-10} n in [1, k– 2]
R_1^l
R_2^l
S_{1k}^{n} n in [1,k–2]
T_{1k}^{n} n in [1, k–2]
R_3^r
R_1^r
S_{1k}^{k-1}
T_{1k}^{k-1}
R_1^l
R_2^l
S_{n+1k}^{m} m in [n+1, k–2] n in [1,k–2]
T_{n+1k}^{m} m in [n+1, k–2] n in [1, k–2]
R_3^r
R_1^r
S_{n+1k}^{k-1} n in [1, k–2]
T_{n+1k}^{k-1} n in [1, k–2]

Figure 3. Illustration of different transformations

sults. We give the results in the following table. The number of expressions generated matches precisely the computed value of $\frac{k^3}{2} + \frac{k^2}{2} - 2k + 2$:

	expressions generated	transformations	equivalence classes generated
k=4	34	18	20
k=5	67	46	35
k=6	116	92	56
k=7	184	160	84
k=8	274	254	120
k=9	389	378	165
k=10	532	536	220
k=11	706	732	286
k=12	914	970	364
k=13	1159	1254	455

5 Related Work

To our knowledge this is among the first results on giving a strict complexity result for rule–based query optimizers. A work that has been published in parallel [Pellenkoft+ 97] gives comparable results in a relational context for duplication–free generation of join trees. It requires further investigation to see whether these results are applicable in a setting where materialization operators require a directed transformation approach. Also this work has been using the Volcano optimization framework.

There exists an abundance of results in the context of rule–based approaches to query optimization, which are related to this work. We review important features of several related and representative projects that adopt the rule–based approach to query optimization.

- The Volcano optimizer generator [Graefe+93] is a data model independent tool that provides a fairly complete framework for quickly developing a query optimizer with specific data model and for experimenting with alternatives in different directions of query processing, including logical operators, algorithms, cost models, and especially transformation and implementation rules.
- The Open OODB object query optimizer [Blakeley+93] is the first working object query optimizer based on a complete extensible optimization framework including logical algebra, execution algorithms, property enforcers, logical transformation rules, implementation rules, and selectivity and cost estimation. The two important contributions of it are the introduction of the "presence in

memory" physical property (materialization), and its first validation of the (Volcano) query optimizer generator paradigm for the generation of a working query optimizer.

- The MOOD [Dogac+95, Ozkan+93] object–oriented query optimizer is not only derived from the Volcano optimizer generator, but also is implemented on the storage manager of EXODUS [Graefe+87, Graefe93] (a predecessor of Volcano). The rule set of the MOOD query optimizer exploited only canonical well–known heuristics, e.g., "combining consecutive projections" and "combining nested selections into a single one".
- The Starburst [Hass+89, Hass+90] extensible database research project has explored extensibility of relational DBMS in "every" aspect of data management. Query processing is divided into two phases: Query rewriting and query planing where query rewrite rules (i.e., production rules, written in C) and plan generation rules (based on grammar like rules) are used, respectively. Lack of homogeneity in Starburst, e.g., the presence of two rule systems within the optimizer alone is particularly unsatisfactory [Finance+94]. The Starburst optimizer allows extensions to both kinds of rules, while the set of search algorithms, cost functions and the set of algebraic operators due to its underlying relational data model does not support extension [Ozsu+95].
- TIGUKAT [Ozsu+95] is an OBMS with an extensible object model characterized by purely behavioral semantics and a uniform modeling approach. It treats everything as first–class object. Consequently, every component of the optimizer, e.g., calculus, algebra, transformation rules, and the queries themselves, are modeled as first–class objects. The designers of the TIGUKAT optimizer put special emphasis on extensibility by pursuing a uniform object–oriented approach. Via the basic OO principle of subtyping, the type system in TIGUKAT incorporates all optimizer components, and it consequently allows extensibility in every direction of optimization.
- Using the EDS ESPRIT project as a testbed, Finance and Gardarin [Finance+94] have developed a rule–based query optimizer. They apply meta–rules to specify the search strategy of the optimizer. This makes their approach different and ambitious, and they see this feature as their main contribution. They proposed a high–level rule language based on extended term rewriting under constraints; it is uniform in the sense that it allows to express both transformation rules and optimization strategies. Their optimizer supports syntactic, semantic and cost–based optimization rules. The rule base is divided into modules each of which is associated with its own control strategy as meta–rules.
- Gral [Becker+92] is an extensible database system, based on the relational model. The key concept embodied in Gral is the formal concept of a many–sorted relational algebra. The query language, executable language, and rule language, as well as the whole system architecture [Becker+92, Guting89], are based on this formal concept. Consequently it provides a more uniform framework for extensibility and optimization; application developers can describe queries, execution plans and optimization rules all with the same kind of notation. Although three

different control mechanisms are offered, alternatives are only considered locally, i.e., no global comparison of query plans based on their complete cost estimates is performed. This may make optimization efficient, but runs the risk of producing bad query plans.

6 Conclusion

In this paper, after reviewing the VODAK query optimization framework we discussed the motivation of our approach and the underlying rational. We then elaborated on the approach to join enumeration and gave a strict complexity result.

In the sense of using the same query optimizer generator, our work is closely related with the Open OODB query optimizer [Blakeley+93]. The VODAK query optimizer, in which the approach is used, is a complete working object query optimizer, although not yet a full–fledged one at the moment with regard to the expressive power of OQL [Cattell+93]. We identified a set of highly efficient rules with practical heuristics which is good both for theoretical reflections and for experiments using a real system [Böhm+97].

The most distinguishable feature of our optimization approach are the various heuristics which we exploited for rule application, of which some are quite counterintuitive but practically effective. Our effort on developing an efficient rule–based optimizer for VODAK, thereby led us to investigate the efficiency problem of rule–based query optimization, and thus we originally proved that it is possible for a rule–based optimizer to achieve the same level of performance as a hard–coded optimizer by using an appropriate rule system with effective heuristics.

Some future work regarding rule–based query optimizers includes:

- Investigate systematically the effect of the rule system and the heuristics for other classes of queries, e.g. more general classes of join queries. E.g. we can show that a rule system compatible with the one used in this paper can perform efficient ordering of single–class–predicate selections (if an order relation is given on the predicates). Can this be extended for ordering of join predicates by using the Adjacent Sequence Interchange Property [Ibaraki+84] that is used in the Krishnamurthy–Boral–Zaniolo algorithm ?
- Perform quantitative performance analysis based on experiments by applying a rule system that has an optimality property for a certain class of queries to queries that are outside the class.
- Using the VQO framework on hand to incorporate semantic optimization rules which are application–dependent and studying the effect on the behavior of the rule–based optimizer when extending the rules system in this way. Semantic optimization is the main target of this investigation which tries to lay efficient foundations towards that goal.

Acknowledgement

We gratefully acknowledge the detailed remarks given by the anonymous reviewers which were very helpful to improve this paper.

References

Aberer, K., Fischer, G.. Semantic Query Optimization for Methods in Object–Oriented Database Systems. In Proc. of 11th IEEE International Conference on Data Engineering, pp. 70–79, Taipei, Taiwan, March 6–10, 1995.

Aberer, K., Hemm, K. Semantic Optimization of Biomolecular Queries in Object–oriented Database Systems, Meeting on the Interconnection of Molecular Biology Databases (MIMBD) '95, Cambridge, UK, 1995.

Batory, D.S., Barnett, J,R., Garza, J.F., Smith, K.P., Tsukuda, K., Twichell, B.,C., and Wise, T.E. GENESIS: An extensible database management system. IEEE Trans. Softw. Eng. 14, 11(Nov. 1988), 1711–1730.

Becker, L., Guting, R.H. Rule–based Optimization and Query Processing in an Extensible Geometric Database System. In ACM Transaction on Database Systems, Vol. 17, No. 2, June 1992, pp. 247–303.

Blakeley, J., McKenna, W., and Graefe, G. 1993. Experiences building the Open OODB query optimizer. In Proc. ACM SIGMOD Int. Conf. On Management of Data, 287–296.

Böhm, K., Aberer, K., Neuhold, E. J. and Yang, X. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. Accepted for publication in VLDB Journal, 1997.

Cattell, R.G.G. (ed). The Object Database Standard: ODMG–93, Release 1.2. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1993.

Derrett, N., and Shan, M.–C. Rule–based query optimization in IRIS. In Proc. of the 17th Annual ACM computer Science Conference (Louisville, Kentucky Feb.. 1989) pp. 78–86.

Dogac, A., Altinel, M., Ozkan, C., Durusoy I. Implementation Aspects of an Object–Oriented DBMS. In SIGMOD RECORD, Vol. 24, No.1, March 1995.

Dogac, A., Ozkan, C, Arpinar, B., Okay, T., and Evrendilek, C. 1994. METU object–oriented DBMS. In Advances in Object–Oriented Database Systems, A. Dogac, M.T. Ozsu, A. Biliris, T. Sellis, Eds. Springer–Verlag.

Finance, B., Gardarin, G. A rule–based query optimizer with multiple search strategies. In Data & Knowledge Engineering 13 (1994) 1–29.

Freytag, J. 1987. A rule–based view of query optimization. In Proc. ACM SIGMOD Int. Conf. On Management of Data, 173–180.

Graefe, G. and DeWitt, D. The EXODUS optimizer generator. In Proc. ACM SIGMOD Int. Conf. On Management of Data, pp. 160–172, May 1987.

Graefe, G. Query Evaluation Techniques for Large Databases, ACM Computing Survey,Vol. 25, No. 2, pp. 73–170, June 1993.

Graefe, G., McKenna, W.J. The Volcano Optimizer Generator: Extensibility and Efficient Search. Proc. 9th ICDE, pp. 209–218, Vienna, Austria, April 19–23, 1993.

Guting, R.H. Gral: An extensible relational database system for geometric applications. In proc. of the 15th International Conference on Very Large Databases (Amsterdam 1989), pp. 33–44.

Haas, L., Cody, W., Freytag, J., Lapis, G., Lindsay, B., Lohman, G., Ono, K., and Pirahesh, H. 1989. Extensible query processing in Starburst. In Proc. ACM SIGMOD Int. Conf. On Management of Data, 377–388.

Hass, L.M., Chang, W., Lohman, G.M., McPHERSON J., Wilms, P.F., Lapis, G., Lindsay, B., Pirahesh, H., Carey, M.J., Shekita, E. Starburst Mid–flight: As the Dust Clears. In IEEE Transaction on Knowledge and Data Engineering, Vol. 2, No. 1, March, 1990.

Ibaraki, T., Kameda, T. Optimal nesting for computing N–relational joins. ACM Trans. on Database systems, 9(3): 482–502. 1984.

Lanzelotte, R. and Valduriez, P. Extending the search strategy in a query optimizer. In Proc. of 17th Int. conf. on Very Large Databases, pp. 363–373, 1991.

Mitchell, G., Zdonik, S.B., and Dayal, U. An Architecture for Query Processing in Persistent Object Stores. In Proc. of the Hawaii International Conference on System Sciences, Vol. II, pp. 787–798, January 1992.

Ono, K., Lohman, G. Measuring the Complexity of Join Enumeration in Query Optimization. Proc. of the 16th VLDB Conference, Brisbane, Australia 1990.

Ozkan, C., Evrendilek, C., Dogac, A., Gesli. T. Design and Implementation of Object–Oriented SQL Query Processor with an Optimizer. Technical Report, 1993, Software Research and Develop Center, Scientific and Technical Research Council of Turkiye Middle East Technical University.

Ozsu, M.T., and Blakeley, J.A. Query Processing in Object–Oriented Database Systems. In Modern Database Management – Object–oriented and Multidatabase Technologies, W.Kim (ed.), Addison–Wesley/ACM Press, 1994, pp. 146–174.

Ozsu, M.T., Munoz, A., Szafron, D. An Extensible Query Optimizer for an Objectbase Management System. In Proc. of the 4th Int. Conf. on Information and Knowledge Management (CIKM'95), November 1995, pp. 188–196.

Pellenkoft, A., Galindo–Legaria, C.A., Kersten, M., Duplicate–free Generation of Alternatives in Transformation–based Optimizers, Proc. of the Fifth International Conference on Database Systems for Advanced Applications, Australia, April 1997.

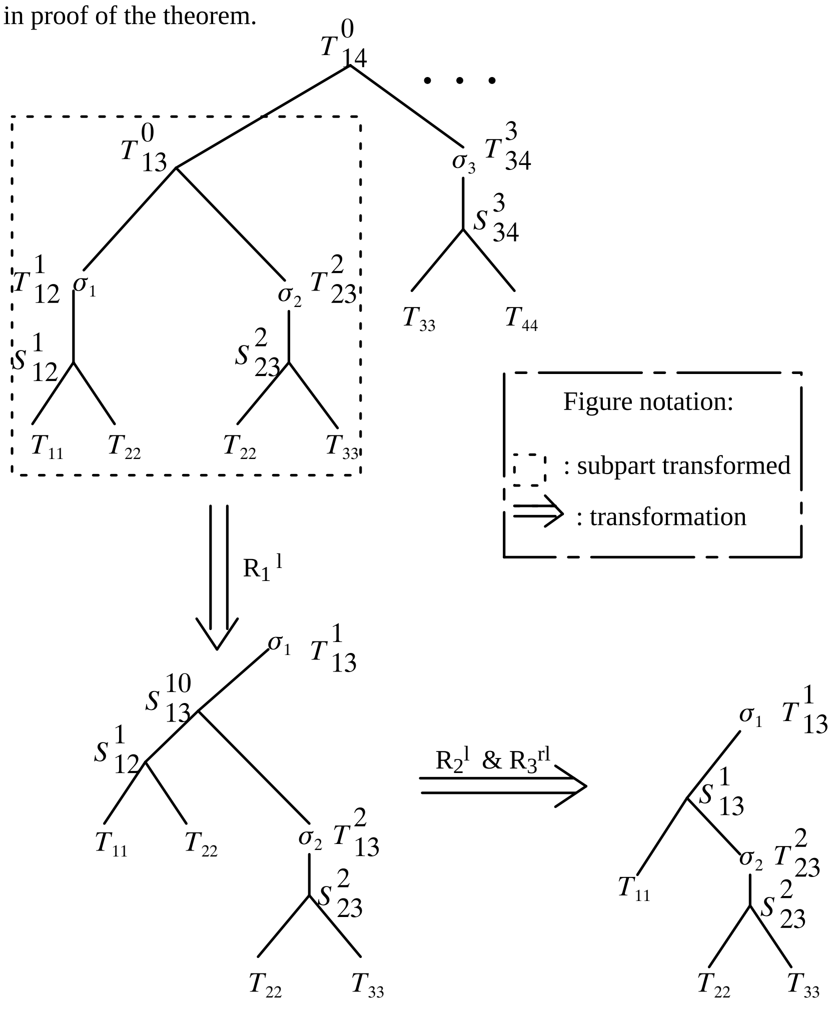
Stonebraker, M., and Rowe, L.A. The design of POSTGRES. In Proceedings of the ACM SIGMOD Conference (Washington, DC, May 1986), pp. 340–355.

VODAK V4.0 User Manual, GMD Technical Report No. 910, Sankt Augustin, April 1995.

Wolfram , S. Mathematica, A System for Doing Mathematics by Computer, Addison Wesley, 1988.

Appendix

Illustration of sample representation and transformation of query representatives as used in proof of the theorem.


T^{0}_{14}
T^{0}_{13}
σ_3 T^{3}_{34}
S^{3}_{34}
T^{1}_{12} σ_1
σ_2 T^{2}_{23}
S^{1}_{12}
S^{2}_{23}
T_{11} T_{22} T_{22} T_{33}
T_{33} T_{44}
Figure notation:
: subpart transformed
: transformation
$R_1{}^{1}$
σ_1 T^{1}_{13}
S^{10}_{13}
S^{1}_{12}
σ_2 T^{2}_{13}
S^{2}_{23}
$R_2{}^{l}$ & $R_3{}^{rl}$
S^{1}_{13}
σ_2 T^{2}_{23}