A Rule-Based View of Query Optimization

Johann Christoph Freytag
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120-6099

Abstract

The query optimizer is an important system component of a relational database management system (DBMS) It is the responsibility of this component to translate the user-submitted query - usually written in a non-procedural language - into an efficient query evaluation plan (QEP) which is then executed against the database The research literature describes a wide variety of optimization strategies for different query languages and implementation environments However, very little is known about how to design and structure the query optimization component to implement these strategies

This paper proposes a first step towards the design of a *modular query optimizer* We describe its operations by *transformation rules* which generate different QEPs from initial query specifications As we distinguish different aspects of the query optimization process, our hope is that the approach taken in this paper will contribute to the more general goal of a modular query optimizer as part of an extensible database management system

1. Introduction

An important component of today's relational database management systems (DBMSs) is the query optimizer Usually, the user's query, expressed in a non-procedural language, describes only the conditions that the final response must satisfy It is the optimizer's responsibility to generate a query evaluation plan (QEP) that computes the requested result efficiently Many different strategies for finding a QEP which evaluates a submitted query efficiently have been proposed Jarke/Koch and Yu/Chang give comprehensive overviews on various query optimization techniques for centralized and distributed DBMSs, respectively [JARK84] [YUCH84]

The difficult task of finding a good, and possibly the best, QEP has frequently led to the implementation of a highly sophisticated but complex optimizer for relational DBMSs The design and implementation of this system component often make changes or extensions in any of its parts very difficult, or even impossible

To overcome the above problems, this paper presents a first step towards a *modular query optimizer* The goal of such a component is to clearly separate the different aspects of query optimization in its design and implementation as much as possible, thus making it easier to make changes to one part of the optimizer without affecting others Instead of addressing the complex task of a modular query optimizer at once, we concentrate on one major aspect of the optimization process The paper describes the translation of a user-submitted query into an algebra-based QEP by *transformation rules* We demonstrate that transformation rules provide an adequate description of the optimization process in a high-level, implementation independent way Furthermore, we discuss how to use the set of transformation rules for the implementation of a query optimizer to efficiently generate QEPs from user-submitted queries

The design of a modular query optimizer is motivated by the general need for extensible DBMSs that can be customized for different application environments Projects such as EXODUS [CARE86], PROBE [DAYA85], Genesis [BATO86], Postgres [STON86], and Starburst [SCHW86] address the question of extensibility of relational DBMSs to support storage and retrieval requirements for applications such as VLSI design, expert systems, or CAD/CAM Depending upon the needs of these various applications, one would like the optimizer component to extend to new query language constructs, to include new access methods as operations into QEPs, to explore different optimization strategies, to use new cost models, etc We show that the rule-based description of the translation from a user-submitted query in QEPs makes it possible to easily change or extend the set of possible QEPs that the query optimizer can generate

The paper is organized as follows In the next section we discuss query optimization and query evaluation in some more detail and motivate our approach which is to apply program transformation techniques in this context In Section 3 we define the source language and the target language of the query optimization process before Section 4 presents the different sets of transformation rules necessary to translate non-procedural query specifications into algebraic QEPs Finally, Section 5 outlines possible extensions and discusses some implementation-related aspects of using the rule-based approach to query optimization

2. Query Optimization and Evaluation

Most relational DBMSs consist of two major components the logical database processor (LDBP) and the physical database processor (PDBP) For example, in System R, the LDBP is called the Relational Data System (RDS) and the PDBP is called the Relational Storage System (RSS) [ASTR76]

For the LDBP, we distinguish three different processing phases that translate a user query into a program which can then be executed by the PDBP The three phases are shown in Figure 1

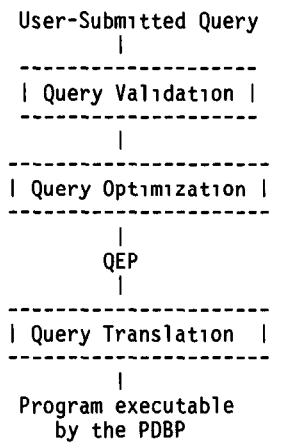

Figure 1 Processing Phases of the LDBP

The first phase, the *validation phase,* checks the query for syntactic and semantic correctness, performs view resolution, and possibly checks authorization before generating some internal representation of the query During the second phase, the *optimization phase*, the LDBP decides on a good, possibly best, evaluation strategy for the user-submitted query Based on the information about the representation of the data accessed, its location in the case of a distributed DBMS, and the available evaluation strategies, the optimizer generates a query evaluation plan (QEP) Finally, the third phase, called the *translation phase*, transforms the QEP into a representation which guarantees its fast execution by the PDBP

In many ways, the three-phase translation of a database query into a program executable by the PDBP resembles a specialized *program transformation problem* [BACK78, BELL84, BURST77, DARL76]

Generally speaking, program transformation promises to provide a comprehensive solution to the problem of producing programs which try to solve several incompatible goals simultaneously On the one hand, programs should be correct and clearly structured, thus allowing easy modification On the other hand, one expects them to be executed efficiently

The transformational approach tries to separate these two concerns by dividing the programming task into two steps The first step concentrates on producing programs that are written as clearly and understandably as possible without considering efficiency issues If initially the question of efficiency is completely ignored, the resulting program might be written very comprehensibly, but might be highly inefficient or even unexecutable The second step then successively transforms programs into more efficient ones - possibly for a particular machine environment - using methods that preserve the meaning of the original program The kind of improvements during the second step goes beyond those achievable during the optimization phase of compilers for conventional programming languages

In many ways, these intentions guided the design of query languages for relational database systems, such as QUEL [STON76] or SQL [ASTR76] Both languages permit the user to express database requests in a clear and understandable form that describes properties of the requested result without considering aspects of an efficient evaluation The DBMS is responsible for translating the query into an executable form while preserving its original meaning

In [FREY86a, FREY86b, FREY86c], we applied program transformation methods to the third phase of the LDBP, the translation phase There we showed how to successfully apply techniques developed in the area of program transformation to problems in database management systems The algorithms proposed there are based on sets of transformation rules that generate iterative programs from QEPs Their use allows the transformation algorithm to be easily extended to handle more complex queries It is only a natural step to view the second processing step, i e query optimization, as another specialized program transformation problem and to apply similar techniques to this processing phase of the LDBP

Transformation rules, however, do not completely describe all aspects of query optimization They only determine the source language and the target language of this processing phase and define how to derive different QEPs in the defined target language from an initial query specified in the source language In this paper we have chosen a general, non-procedural query representation to be the source language and an extended relational algebra to be our target language, both of which are defined in more detail in the following section The rule-based optimization describes the central part of the optimization algorithm for SQL [SELI79] The algorithm is complex enough to serve as a realistic test case to prove the power of the rule-based description

In our opinion, there are two other major elements which are necessary to completely determine all aspects of query optimization On the one hand, we need to describe in which order to generate different QEPs to find a good candidate for the evaluation of the submitted query, i e , we must define how to *search through* the set of all possible QEPs Search strategies, such as a greedy search, a breadth-first or depth-first search with or without dynamic programming, or a *k*-step look-ahead search are commonly used for this purpose On the other hand, we need to compare different QEPs deciding which ones are better than others Usually, this decision is based on the cost of using various resources, such as CPU, disk I/O, number of messages etc More details can be found in Jarke and Koch [JARK84], and Yu and Chu [YUCH84]

Using transformation rules for query optimization has attracted other researchers as well The EXODUS project especially investigates how to include rules into an architecture of an *optimizer generator* [GRAE87, CARE86] The work by Graefe and DeWitt focuses on the architectural aspects, the design, and the implementation of such a tool [GRAE87] They demonstrate to some extent that it is possible to separate the generation of QEPs from the cost function and the search strategy

Although our research goals are quite similar to theirs, we would like to investigate the fundamentals of query optimization and necessary concepts for a modular query optimizer first, before designing and implementing an optimizer generator It is the purpose of this paper to concentrate on the first aspect of query optimization, that is, the rule-based description of how to generate different QEPs from an initial query specification We demonstrate in the following sections that transformation rules are adequate to

express this aspect of optimization in a high-level, implementation-independent manner

3. The Source Language and the Target Language of Query Optimization

This section introduces the source language and the target language for query optimization As the source language we use *conjunctive queries* that exclude subqueries (in the SQL sense) and aggregate queries of any kind As we shall discuss later, our choice does not impose any restrictions on the rule-based transformation We shall demonstrate that our sets of rules can easily be extended to handle more complex queries

To describe the transformation uniformly and independent of any particular database query language, we assume that the initial query has the Lisp-like form

```
(SELECT
  <project_list> <select_pred_list>
  <join_pred_list> <table_list>)
```

which represents the source for the optimization phase The different parts of the list describe the projection of the resulting tuples, the predicates applicable to single tables, the join predicates, and the tables accessed, respectively

Example 1

We use the following database for the examples in this paper

EMP (Emp#, Name, Salary, Dept, Status)

PAPER (Emp#, Title, Subject, Year)

CONF (Emp#, CName, Year)

Each tuple in the relation EMP describes an employee by his or her employee number, name, salary, department, and status Relation PAPER stores the employees who wrote papers recording the title, the subject, and the year of the publication Relation CONF records the attendance of conferences by employees in a certain year For the query

Find the name of all professors who published a database paper in the same year as they attended the VLDB conference

the internal representation looks as follows

```
(SELECT
  (EMP Name)
  ((EMP Status=Prof)
   (PAPER Subject=DB) (CONF CName=VLDB))
  ((EMP Emp#=PAPER Emp#)
   (EMP Emp#=CONF Emp#)
   (PAPER Year=CONF Year))
  (EMP, PAPER, CONF))
```

□

As our target language we propose a small set of algebraic operators that are sufficient to express QEPs for the evaluation of those queries we consider We use an *extended relational algebra* since the kind of operators go beyond those of regular relational algebra

For example, we introduce operators to specify the use of a nested-loop join or a merge-join, the sort of a relation, an index access, or a relation access All operators manipulate some incoming list of tuples that are either derived from a relation referenced by name, or which are the output of some other operator, and generate a new list of tuples

(FSCAN <t_pred> rel)

scans a relation while applying the list of predicates in <t_pred>, which might be empty

(ISCAN <i_pred> index <t_pred> rel)

scans a relation using the *index* to apply the given index predicates in <i_pred> before scanning the table and applying predicates in <t_pred> Both predicate lists might be empty

(PROJECT <proj_list> list)

denotes the project operator which projects all incoming tuples onto those attributes specified by the projection list

(LJOIN <join_pred> list1 list2)

denotes a nested loop join with *list1* being the outer list and *list2* being the inner list of tuples

(MJOIN <join_pred> list1 list2)

denotes a merge join with *list1* being the *outer list* and *list2* being the *inner list* of tuples

(SORT <attr_list> list)

sorts and outputs a list of tuples which are ordered according to the attribute list

The current set of operators is not the most complete one For example, it does not include operations to select a list of tuples from an input list, to create a temporary relation, or to evaluate SQL subqueries We shall demonstrate later how to extend this set of operators to a more complete set

Example 2

Assuming that there exists an index I1 on EMP Status one possible QEP for the query in Example 1 might look as follows

```
(PROJECT (EMP Name)
  (MJOIN
    ((EMP Emp#=PAPER Emp#)
     (CONF Year=PAPER Year))
    (SORT (PAPER Emp#, PAPER Year)
      (FSCAN ((PAPER Subject=DB)) PAPER))
    (SORT (EMP Emp#, CONF Year)
     (LJOIN ((EMP Emp#=CONF Emp#))
      (ISCAN ((Status=Prof)) I1 () EMP)
      (FSCAN ((CONF CName=VLDB)) CONF)))))
```

□

During the transformation of the user-submitted query into a QEP, we need to generate some "intermediate" expressions that use the following three operators

(SCAN <sel_pred> rel)

denotes a relation scan including its selections without specifying the access path

(JOIN <join_pred> <scan_list>)

denotes the join of an arbitrary number of tables without specifying their order <scan_list> is a list of expressions, each of which is scanning a relation

(TJOIN <join_pred> list1 list2)

denotes a two-way join without specifying what kind of join to perform

In the next section we shall use all of the above operators in expressions generated by the different sets of transformation rules

4. The Rule-Based Generation of Query Evaluation Plans

This section defines different sets of transformation rules that successively translate a user-submitted query into different QEPs The transformation rules that we present in the next subsections reflect the important aspects of the optimization algorithm as described by Selinger et al [SELI79] Their algorithm restricts the kind of QEPs generated in the following way

1 To access a single relation, all available access paths are considered The choice of either a relation scan or a scan using an existing index generates all possible orders in which we can retrieve tuples from a relation

2 Either a nested-loop join or a merge-join implement the join operation

3 The inner tuple list of any join is generated by a *non-composite expression*, that is, only a single relation is accessed

The third restriction considerably limits the number of possible QEPs generated Notice that the QEP of Example 2 does not satisfy the last restriction, thus it will not be generated by the optimization algorithm of Selinger et al However, exchanging the inner relation with the outer relation of the MJOIN operator provides a QEP that satisfies the above restrictions

4.1 Basic Definitions

To describe the different translation steps of query optimization, we introduce *transformation rules* or *rewriting rules* [HUET80] For the purpose of this paper the transformation rule $(t_1 \rightarrow t_2)$ specifies to replace expression t_1 by expression t_2 For example, let t_1 and t_2 be arbitrary expressions, then the rule $((\mathit{if\ true}\ t_1\ t_2) \rightarrow t_1)$ means to reduce the conditional expression to the expression t_1 if the condition is known to be *true*

We extend the rule notation by introducing *restricted rules* A restricted rule $(t_1 \xrightarrow{C} t_2)$ specifies to replace t_1 by t_2 whenever condition C evaluates to *true* We do not restrict the form of condition C except that all variables referenced in C must be used in expression t_1 and t_2 For example, C might restrict t_1 to be a variable name denoting a relation or to be an expression referencing some relation names

To reference an arbitrary expression t_i in a list of expressions $(t_1 \ \ t_i \ \ t_n)$ we use the notation (t_i) where " " denotes zero or more subexpressions to the left or right of some t_i For instance, let $(*\ t_1 \ \ t_n)$ denote an expression which multiplies an arbitrary number of integers with * being the multiplication operator The result of the expression is 0 if any of the t_i's is 0 The transformation rule $((*\ \ 0\ \ ) \rightarrow 0)$ exactly describes this arithmetic property

For restricted rules, we often use functions in the conditions to determine properties of relations, predicates, or general expressions We introduce the function $Ind(I1, R)$ to determine if $I1$ is an index on relation R Furthermore, let p be a predicate then $T(p)$ denotes the set of relation names referenced by p For example, if p = (EMP Status = Prof), then $T(p)$ = {EMP} We also apply T to general expressions t to determine the set of relation names referenced in t Simularly, we use function $A(t, RS)$ to determine the set of attributes in any expression t for the set of relations in RS For example, let p be the list of predicates ((EMP Status = Prof) ∧ (EMP Emp# = PAPER Emp#) ∧ (PAPER Subject = DB)) then $A(p, \{EMP\})$ yields the attribute set {EMP Emp#, EMP Status} If no list of relations is present, function A returns all attributes referenced in an expression

To determine the *order* of tuples that are returned by a QEP, we introduce two more functions Let I be an index for some relation R, then $O(I)$ denotes the (ordered) attribute list which determines the (ascending) order in which tuples are retrieved from R using index I Based on the definition of O, we introduce the predicate < attr_set> ∈ $O(I)$ with <attr_set> being a set of attributes The predicate determines if any order of attributes in the set is a prefix of the ordered attribute list $O(I)$ For example, let $O(I)$ be the list <Name, Dept, Salary> then {Dept,Name} ∈ $O(I)$ is *true*, while {Salary,Dept} ∈ $O(I)$ is *false*

We generalize function O to function Ω which applies to QEPs For any QEP Q, $\Omega(Q)$ denotes the order imposed on the set of tuples created by Q We can define function Ω recursively based on the set of operators as follows

- Ω((FSCAN <p> rel)) = <>
- Ω((ISCAN <ip> ind <tp> rel)) = $O(ind)$
- Ω((PROJECT <pr> list)) = Ω(list)
- Ω((LJOIN <jp> list1 list2)) = Ω(list1)
- Ω((MJOIN <jp> list1 list2)) = Ω(list1)
- Ω((SORT <a_list> list)) = <a_list>

4.2. Generating an Algebraic Query Form

The first transformation step translates the initial form of the query into an algebraic form for further manipulation The following rules perform the desired transformation

- $((SELECT\ e_1\ e_2\ (\ t_1\ )) \rightarrow$
  $(SELECT\ e_1\ e_2\ (\ )\ ((SCAN\ ()\ t_1))))$
- $((SELECT\ e_1\ e_2\ (\ t_1\ )\ (\ )) \rightarrow$
  $(SELECT\ e_1\ e_2\ (\ )\ (\ (SCAN\ ()\ t_1)\ )))$
- $((SELECT\ e_1\ (\ p_1\ )\ e_2\ (\ (SCAN\ (\ )\ t_1)\ )) \xrightarrow{C1}$
  $(SELECT\ e_1\ (\ )\ e_2\ (\ (SCAN\ (\ p_1\ )\ t_1)\ )))$
- $((SELECT\ e_1\ ()\ e_2\ ()\ e_3) \rightarrow (PROJECT\ e_1\ (JOIN\ e_2\ e_3)))$

with $C1$ = $(T(p_1) = \{t_1\})$

The first rule initializes the transformation by taking any relation, removing it from the list of relations, and attaching the SCAN operator to it The generated expression is stored in a newly created list The second rule then removes one relation from the relation list, attaches the SCAN operator, and adds the new expression to the list of SCAN expressions generated so far The third rule distributes the selection predicates among the different relations depending on the relation names referenced in the predicates If the selection list and the relation list are empty, the last rule substitutes an n-way join followed by a projection for the selection expression Notice that we can apply the second and third rule in arbitrary order.

Example 3

For the query of Example 1, the above four rules generate the expression

```
(PROJECT (EMP Name)
  (JOIN
    ((EMP Emp#=PAPER Emp#)
     (EMP Emp#=CONF Emp#)
     (PAPER Year=CONF Year))
    ((SCAN ((EMP Status=Prof)) EMP)
     (SCAN ((PAPER Subject=DB)) PAPER)
     (SCAN ((CONF CName=VLDB)) CONF))))
```

□

4.3. Generating the Access Paths

Once we have generated an algebraic form of the query, we may refine the expression by using information about the internal storage structure of the relations In particular, predicates on relations might be evaluated by using existing indexes The following rules determine the possible choices for accessing an individual relation

- $((SCAN\, p_1\, t_1) \rightarrow (FSCAN\, p_1\, t_1))$
- $((SCAN\, p_1\, t_1) \overset{C1}{\rightarrow} (ISCAN\, p'_1\, I1\, p''_1\, t_1))$

with $C1 = (Ind(I1, t_1) \wedge (p_1 = (p'_1 \cup p''_1)) \wedge (p'_1 \cap p''_1 = \emptyset) \wedge (A(p'_1, t_1) \in O(I1)))$

The first rule converts the generic scan into a simple relation scan without using any indexes The second rule takes advantage of an existing index and generates different access plans depending on how the selection list is split between the index and the relation The condition $C1$ ensures that $I1$ is an index on the relation denoted by t_1, and that the predicate list is partitioned such that all attributes that appear in the terms of the index predicate form a prefix on the attribute list that determines the index order Notice that we also allow the selection list for the index to be empty, in which case all tuples of relation t_1 are retrieved in $O(I1)$ order

Condition $C1$ might be extended to further restrict the application of the second rule For example, an existing index might only be used if all predicate terms except the last one in attribute order test for equality, only the last term may involve an inequality test

Example 4

Based on the assumption of Example 2 that relation EMP has an index I1 on attribute Status we might generate three different expressions from the expression of Example 3 using the above transformation rules They differ in the access path used for relation EMP The SCAN operator translates either into a regular relation scan without using the index, an index scan without an index predicate, or an index scan using the predicate (EMP Status=Prof) The other two scans for relations PAPER and CONF translate into FSCAN operators, as no indexes exist for these two relations

□

4.4. Join Processing

Much of the processing of relational queries is concerned with exploring different join orders among the tables involved in the query, and with choosing a join method In this section we restrict ourselves to the join strategies proposed by Selinger et al [SELI79] as we already discussed at the beginning of Section 3 To implement these two aspects of join processing, we present two sets of rules The first one generates different join orders The second one chooses between the different evaluation strategies for a two-way join, in our case between a nested-loop join and a merge-join Notice that the order in which we present the rules does not imply an application order for the rules Most of the rules which generate the different join orders and the join methods can be applied before or after the rules which generate the access paths, or we can even mix both sets It is the responsibility of the search strategy to determine the order in which to apply the rules

4 4 1 Generating the Join Orders

The first step of join processing is implemented by a set of three transformation rules that generate two-way join expressions, for joining relations during query evaluation The first two rules are quite powerful as they generate many different join orders for the same initial expression

- $((JOIN\, t_1\, (\ \ t_2\ \ )) \rightarrow (JOIN\, t_1\, (\ \ )\, t_2))$
- $((JOIN\, (\ p_1\ )\, (\ t_1\ )\, t_2) \overset{C1}{\rightarrow} (JOIN\, (\ \ )\, (\ \ )\, (TJOIN\, (p_1)\, t_2\, t_1)))$
- $((JOIN\, ()\, ()\, t_1) \rightarrow t_1)$

with $C1 = (T(p_1) \in T(t_1) \cup T(t_2))$

The first rule initializes the transformation by choosing any relation to be the outermost relation of any join The second rule successively creates two-way joins by combining any relation in the relation list with the two-way join expression generated so far When the list of join predicates and the list of relations are empty, the last rule discards the n-way join operator

Unfortunately, this set of rules is not sufficient to completely reduce the list of join predicates, if there are cyclic queries or if two relations are related by more than one join predicate We need two additional rules which push predicates into a two-way join expression if the relations referenced in the predicate are already present in that expression

- $((JOIN\, (\ p_1\ )\, t_1\, (TJOIN\, (\ )\, t_2\, t_3)) \overset{C1}{\rightarrow} (JOIN\, (\ \ )\, t_1\, (TJOIN\, (\ p_1\ )\, t_2\, t_3)))$
- $((TJOIN\, (\ p_1\ )\, (TJOIN\, (\ )\, t_1\, t_2)\, t_3) \overset{C2}{\rightarrow} (TJOIN\, (\ \ )\, (TJOIN\, (p_1\ )\, t_1\, t_2)\, t_3))$

with $C1 = (T(p_1) \in T(t_2) \cup T(t_3))$ and $C2 = (T(p_1) \in T(t_1) \cup T(t_2))$

The first rule pushes a join predicate into the two-way join expression when it is applicable The second rule pushes the predicate even further down into the expression to ensure that it is applied as soon as possible

Example 5

One of the expressions generated in Example 4 is

```
(PROJECT (EMP Name)
  (JOIN
    ((EMP Emp#=PAPER Emp#)
     (EMP Emp#=CONF Emp#)
     (PAPER Year=CONF Year))
    ((ISCAN ((EMP Status= Prof)) I1 () EMP)
     (FSCAN ((PAPER Subject=DB)) PAPER)
     (FSCAN ((CONF CName=VLDB)) CONF))))
```

One possible order which we can generate using the first two rules of this subsection is by first joining relations EMP and PAPER before performing a join with relation CONF

```
(PROJECT (EMP Name)
  (JOIN ((PAPER YEAR=CONF Year)) ()
    (TJOIN ((EMP Emp#=CONF Emp#))
      (TJOIN ((EMP Emp#=PAPER Emp#))
        (ISCAN (EMP Status=Prof) I1 () EMP)
        (FSCAN ((PAPER Subject=DB)) PAPER))
      (FSCAN ((CONF CName=VLDB)) CONF))))
```

In order to completely reduce the list of join predicates, we need to apply the next to last rule of this subsection before discarding the n-way join with its empty predicate list, thus yielding the final expression of this transformation step

```
(PROJECT (EMP Name)
  (TJOIN
   ((PAPER YEAR=CONF Year)
    (EMP Emp#=CONF Emp#))
   (TJOIN ((EMP Emp#=PAPER Emp#))
     ((ISCAN (EMP Status=Prof) I1 () EMP)
     (FSCAN ((PAPER Subject=DB)) PAPER))
   (FSCAN ((CONF CName=VLDB)) CONF)))
```

□

4 4 2 Generating the Join Methods

Once we have generated the different orders in which to join all relations, we must choose a join method to implement the two-way join For the purpose of this paper, we restrict the possible join methods to either a nested-loop join or a merge-join, both of which are generated by the following three rules

- $((TJOIN\, p_1\, t_1\, t_2) \xrightarrow{C1} (MJOIN\, p_1\, t_1\, t_2))$
- $((TJOIN\, p_1\, t_1\, t_2) \xrightarrow{C2} (MJOIN\, p_1\, (SORT\, \mathrm{A}(p_1, \mathrm{T}(t_1))\, t_1)\, t_2))$
- $((TJOIN\, p_1\, t_1\, t_2) \rightarrow (LJOIN\, p_1\, t_1\, t_2))$

with $C1 = (\mathrm{A}(p_1, \mathrm{T}(t_1)) \in \Omega(t_1))$ and $C2 = (\mathrm{A}(p_1, \mathrm{T}(t_1)) \notin \Omega(t_1))$

The first two rules generate a merge-join If expression t_1 does not return the set of tuples in some order "compatible" with the join predicate, we need to sort the tuples The last rule translates the two-way join into a nested-loop join

In the case of a merge-join, we must retrieve the inner tuple list in the same order as the outer one Therefore, we introduce an additional rule which, if necessary, inserts the sort operator to satisfy this condition

- $((MJOIN\, p_1\, t_1\, t_2) \xrightarrow{C1} (MJOIN\, p_1\, t_1\, (SORT\, \mathrm{A}(p_1, \mathrm{T}(t_2))\, t_2)))$

with $C1 = ((\Omega(t_2) \notin \Omega(t_1)) \wedge (\Omega(\mathrm{A}(p_1, \mathrm{T}(t_2))) = \Omega(t_1)))$ Condition $C1$ ensures that the rule is applied only if the tuple order of the inner list is not the same as the tuple order of the outer list generated by t_1, and that the SORT operator returns tuples in the same order as expression t_1

To further improve join processing, System R takes advantage of existing indexes on the inner relations Instead of evaluating all terms of the join predicate by the join operator, those are "pushed down" into an index scan to use them as additional predicates on the index The following two rules implement this improvement which we apply to both join operators, the nested-loop join and the merge-join

- $((MJOIN\, p_1\, t_1\, (ISCAN(\ )\, t_2\, t_3\, t_4)) \xrightarrow{C1} (MJOIN\, p'_1\, t_1\, (ISCAN(p''_1\ )\, t_2\, t_3\, t_4)))$
- $((LJOIN\, p_1\, t_1\, (ISCAN(\ )\, t_2\, t_3\, t_4)) \xrightarrow{C1} (LJOIN\, p'_1\, t_1\, (ISCAN(p''_1\ )\, t_2\, t_3\, t_4)))$

with
$C1 = ((p_1 = p'_1 \cup p''_1) \wedge (p'_1 \cap p''_1 = \emptyset) \wedge (\mathrm{A}((p''_1\ )) \in \Omega(t_2)))$
For both rules, the predicate list p_1 is partitioned into two parts, one of which can be evaluated by the index, the other part being evaluated by the join operator Condition $C1$ ensures this restriction Similarly, we can define additional rules to push down the remaining condition into the inner relation scan, thus evaluating the join condition while retrieving the tuples by the RSS component [SELI79]

Example 6

We can transform the two-way joins of the final expression in Example 5 either into merge-join operations or nested-loop joins One possible expression generated by the first three rules of this section is

```
(PROJECT (EMP Name)
  (LJOIN
    ((PAPER YEAR=CONF Year)
     (EMP Emp#=CONF Emp#))
    (MJOIN
      ((EMP Emp#=PAPER Emp#))
      (SORT (EMP Emp#)
        (ISCAN ((EMP Status=Prof)) I1 () EMP)
      (SORT (PAPER Emp#)
        (FSCAN ((PAPER Subject=DB)) PAPER))
    (FSCAN ((CONF CName=VLDB)) CONF)))
```

□

5. Discussion

In this section we show one possible extension of the current rule set for distributed query processing and briefly discuss some implementation-related aspects

5.1. Distributed Queries

This section demonstrates how easy it is to extend the current set of rules by new ones to process distributed query processing We do not intend to implement the most sophisticated query evaluation strategy for distributed queries, our interest is focused more on the flexible use of rules for describing one aspect of query optimization

We introduce two rules which describe some of the shipping strategies for distributed queries as defined by Lohman et al [LOHM84] To move lists of tuples between sites and to store them if necessary, we define two new algebraic operators SHIP and STORE

(SHIP <site_name> list)

ships a list of tuples to the designated site

(STORE <rel_name> list)

stores a list of tuples in the relation <rel_name> [1]

Additionally, we introduce the function *Site*, which we need for conditional rules $Site(t_1)$ determines the site at which the tuples generated by t_1 reside We can define function *Site* recursively, similar to the function Ω

Based on these new definitions, we define the following two rules to generate expressions which either ship the outer tuple list to the site where the inner tuple list resides or vice versa

- $((TJOIN\, p_1\, t_1\, t_2) \overset{C1}{\rightarrow} (TJOIN\, p_1\, t_1\, (FSCAN\,()\ (STORE\, T\, (SHIP\, Site(t_1)\, t_2)))))$
- $((TJOIN\, p_1\, t_1\, t_2) \overset{C1}{\rightarrow} (TJOIN\, p_1\, (SHIP\, Site(t_2)\, t_1)\, t_2))$

with $C1 = (Site(t_1) \neq Site(t_2))$, and T being a new relation that stores the result of t_2 at t_1's site Notice, that the two rules are not symmetric When we ship tuples of the outer list to the inner list's site, we do not need to store them since each tuple is accessed only once However, if the tuples of the inner list are shipped to the outer list's site, we create a new relation to rescan the inner table locally, thus avoiding repetitive shipping of the inner tuple list

5 2. Implementation of the Rule-Based Optimization

After having defined the rules which generate different QEPs from a user-submitted query, the general question is how to build a query optimizer using the rule-based specification We show one possible approach in Figure 2 which describes the generation of a query optimizer in two steps The specification of the transformation rules as presented in this paper describes the generation of QEPs from initial queries in a high-level, implementation-independent way To guarantee their fast execution in any implementation, we propose to translate them into "transformation procedures" in programming languages such as C or Pascal Additionally, the translation into transformation procedures should consider the internal data structures which represent a query, thus making the transformation even more effective

Carey et al investigated a similar architecture for building an optimizer generator as part of the EXODUS project [GRAE87, CARE86] They especially suggested the translation of transformation rules into transformation procedures and proposed a "hill-climbing" algorithm to guide the selection of rules during the optimization

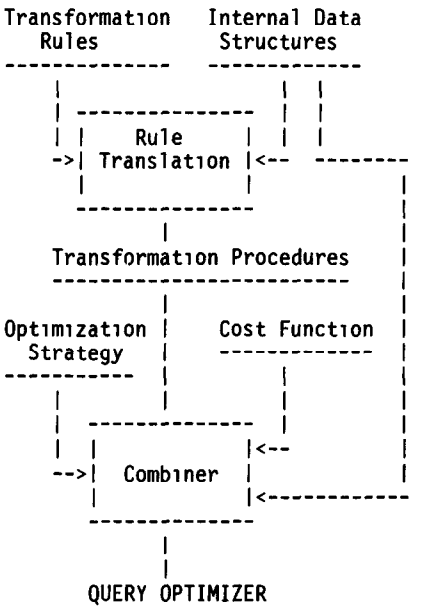

Figure 2 Generation of a Query Optimizer

As we pointed out in Section 2, the transformation rules describe only one aspect of query optimization We need to add information about the search strategy used in the optimization process, and to specify cost functions to compare different QEPs and to select the best one Therefore, in a second step all three aspects are combined to generate the query optimizer

Of course, this approach leads to many open questions Is it possible to show that a given set of rules generate only valid QEPs? How do we structure the rule translation process? Can we use already existing techniques, methods and tools? Is it possible to specify initially the three aspects of query optimization independently without losing the required efficiency of the optimization process? How can the optimizer avoid scanning all rules each time it has to find those rules that are applicable on the current query expression? The definition of rules in Section 4 already indicates a possible order for applying the rules during the transformation process These questions (and others) need to be answered by further research before we can achieve our goal of a modular query optimizer

6. Conclusion

We presented a rule-based description of how to generate different QEPs from a user-submitted query The rules do not completely cover all aspects of the optimization algorithms as described by Selinger et al [SELI79] However, we demonstrated that a rule-based description allows a high-level specification of one part of the query optimization process Additionally, we addressed the problem of how to extend the current rule set and discussed some initial ideas for implementing a rule-based optimizer Despite the many open problems, we are convinced that this kind of specification can support the implementation and - at least partially - the generation of a query optimizer, which will contribute to the more

1 Of course this operator is usually used for centralized queries too

general goal of building a modular query optimizer as part of an extensible database management system

7. Acknowledgement

We wish to thank Guy Lohman for several stimulating discussions which clarified many aspects of the query optimization algorithm in System R Laura Haas' and Guy Lohman's careful reading of an earlier draft of this paper improved the presentation in many ways

Bibliography

[ASTR76] Astrahan, M et al , *SYSTEM R Relational Approach to Database Management,* ACM Transactions of Database Systems **1,2** (June 1976) pp 97-137

[BACK78] Backus, J , *Can Programming be liberated from the von Neuman Style? A Functional Style and its Algebra of Programs,* Communications of the ACM **21,8** (August 1978) pp 613-641

[BATO86] Batory, D S et al , *GENESIS A Reconfigurable Database Management System,* Technical Report 86-07, Department of Computer Sciences, The University of Texas, Austin (1986)

[BELL84] Bellegarde, F , *Rewriting Systems on FP Expressions that Reduce the Number of Sequences they Yield,* ACM Symposium on LISP and Functional Programming (August 1984) pp 63-73

[BURS77] Burstall, R,M and Darlington, J , *A Transformation System for Developing Recursive Programs,* Journal of the ACM **24,1** (January 1977) pp 44-67

[CARE86] Carey, M et al , *The Architecture of the EXODUS Extensible DBMS,* Proceedings of the International Workshop on Object-Oriented Database Systems, Asilomar, Pacific Grove, California (September 1986)

[DARL76] Darlington, J and Burstall, R M , *A System which Automatically Improves Programs,* Acta Informatica **6,1** (January 1976) pp 41-60

[DAYA85] Dayal, U and Smith, J , *PROBE A Knowledge Oriented Database Management System,* Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems (February 1985)

[FREY85a] Freytag, J C and Goodman, N , *Rule-Based Translation of Relational Queries into Iterative Programs,* Proceedings ACM SIGMOD 1986, Washington, D C (May 1986) pp 206-214

[FREY85b] Freytag, J C and Goodman, N , *Translation Database Queries into Iterative Programs using a Program Transformation Approach,* IBM Research Report RJ 5092 (April 1986)

[FREY85c] Freytag, J C and Goodman, N , *Translating Aggregate Queries into Iterative Programs,* Proceedings VLDB 1986, Kyoto, Japan (August 1986)

[GRAE87] Graefe, G and DeWitt, D , *The EXODUS Optimizer Generator,* Proceedings ACM SIGMOD 1987, San Francisco, CA (May 1987)

[JARK84] Jarke, M and Koch, J , *Query Optimization in Database Systems,* ACM Computing Surveys **16,2** (June 1984)

[HUET80] Huet, G , *Confluent Reductions Abstract Properties and Applications of Term rewriting Systems,* Journal of the ACM **27,4** (October 1980) pp 797-821

[LOHM84] Lohman, G , et al , *Query Processing in R*,* IBM Research Report RJ 4272 (April 1984)

[SCHW86] Schwarz, P et al , *Extensibility in the Starburst Database System,* Proceedings of the Asilomar Workshop on Object-Oriented Database Systems (September 1986)

[SELI79] Selinger, P G et al , *Access Path Selection in a Relational Database Management System,* Proceedings ACM SIGMOD 1979, Boston, MA (June 1986)

[STON86] Stonebraker, M and Rowe, L , *The Design of Postgres,* Proceedings ACM SIGMOD 1986, Washington, D C (May 1986) pp 340-355

[YUCH84] Yu, C T and Chang, C C , *Distributed Query Processing,* ACM Computing Surveys **16,4** (December 1984)