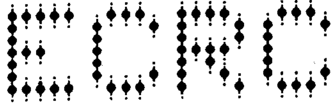

Internal Report IR-KB-59
20 March 1989

The Basic Principles of Query Optimization in in Relational Database Management Systems

Johann Christoph Freytag

European Computer-Industry
Research Centre GmbH
Arabellastr. 17
D-8000 Muenchen 81
West Germany

Abstract

The query optimizer is an important component in today's relational database management systems (DBMSs). This component is responsible for translating a user-submitted query – usually written in a non-procedural language – into an efficient query evaluation program that can be executed against the database. The research literature describes a wide variety of optimization algorithms for different query languages and implementation environments. The different ways these algorithms are presented often make it difficult to compare and analyze them without considerable effort. Therefore, the first goal of this paper is to develop a uniform understanding of (possibly) all query optimization algorithms by identifying three important principles that characterize all of them.

Based on our identifying principles we propose an abstract but simple operational model for query optimization in the second part of this paper. This model clearly reflects the three different principles of any optimization algorithm while showing their interaction at the same time. The model provides the basis for a modular architecture to implement an extensible query optimizer.

1 Introduction

An important component in today's relational DBMS is the query optimizer. A user request that is usually expressed in a high-level, non-procedural language describing the conditions that the result produced by the DBMS has to satisfy. It is the optimizer's responsibility to create a *query evaluation plan* (QEP), i.e. a procedural specification of the user-submitted query that computes the requested answer efficiently.

The research literature proposes a wide variety of query optimization algorithms. Jarke/Koch and Yu/Chang give comprehensive overviews on various query optimization techniques for centralized and distributed DBMSs, respectively [JK84] [YC84]. However, these overviews do not attempt to develop a model of query optimization that explains and presents the algorithms in a uniform way. Since many optimization algorithms differ in their computational behavior while reflecting aspects of the implementation environment at the same time, it is the purpose of this paper to understand all of them by few simple concepts. This understanding becomes even more important in case we want to change or extend existing algorithms to adapt them to new requirements.

To develop this fundamental understanding for existing optimization algorithms, Section 2 presents a framework by clearly identifying three major "principles" that are found in every optimization algorithm. We verify our framework by comparing and analyzing existing algorithms in Section 3.

Our work is motivated by the general need for an extensible query optimization component that can be customized for different application environments. Projects such as EXODUS [C+86b], PROBE [DS85], Genesis [B+86], Postgres [SR86], and Starburst [S+86] 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 on 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..

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) [A+76]. Most research work in the area of extensible DBMSs has focused on new architectures for a flexible PDBP that satisfies the increasingly complex requirements for storing and retrieving structured objects [C+86b] [DS85].

The research work on the LDBP has mostly concentrated on describing the extended requirements for the PDBP. To adjust the optimizer, a subcomponent of the LDBP, to these new requirements, recent work by Graefe and DeWitt, Lohman, and us describes alternative implementations of an optimizer component using transformation rules that, at least partially, accommodate the extensibility requirement [GD87] [Loh88] [Fre87]. In [GD87] Graefe and DeWitt focus on the description of an optimizer generator without discussing the requirements for a generalized optimizer architecture. In [Fre87], we outlined the basic structure of an optimizer

generator without giving more details on how to build such a system. What is missing is a comprehensive and uniform framework for query optimization that can serve as the basis for implementing the optimization component.

The framework for query optimization developed in Section 2 is more than an intellectual exercise since it provides such a basis for query optimization. This is necessary and important for adjusting existing optimization algorithms to new requirements and for building modular optimizers and optimizer generators. The analysis of Section 3 naturally leads to a simple operational model for query optimization which is presented in Section 4. By combining the different principles in a controlled manner the model demonstrates that it is possible to change one aspect of the optimization algorithm without affecting others. The model can serve as a general architecture for implementing a modular query optimizer. In Section 4 we also discuss current research work that supports and validates our operational model.

2 General Aspects of Optimization

To provide a better understanding of what we mean by the term query optimization, the first subsection briefly outlines the difference between query optimization and query modification. The second subsection the discusses the three major aspects of query optimization that can be found in (almost) all optimization algorithms described in the literature.

2.1 Two Kinds of Query Optimization

The term *query optimization* has been used in the literature to describe different aspects of query processing. In general, we can distinguish two major phases of optimization that can be performed during the translation of the user-submitted query into an executable program. The first one, *query modification*, intends to *rewrite* the initial query such that we can expect an improved efficiency during the evaluation of the query. For example, consider the query of Example 1.

Example 1:

We use the following database for the examples in this paper:

```
EMP (Emp#, Name, Salary, Dept#)
DEPT (Dept#, DeptName, Mgr)
```

Each tuple in the relation EMP describes an employee by his or her employee number, name, salary, and department number. Relation DEPT stores the department number, name, and manager. The query

```
SELECT Name, Mgr FROM EMP, DEPT
WHERE EMP.Dept# = DEPT.Dept#
      and EMP.Dept# < 1000
```

can be modified as

```
SELECT Name, Mgr FROM EMP, DEPT
WHERE EMP.Dept# = DEPT.Dept#
      and DEPT.Dept# < 1000
```

by restricting the Dept# of the DEPT relation instead in the EMP relation. □

By moving the restriction on the Dept# attribute from the EMP relation to the DEPT relation, the joining tuples of that relation are immediately restricted to those with Dept# < 1000. The research literature reports on a wide variety of query modification schemes that range from syntactically rewriting a query to including semantic knowledge to simplify the initial query.

It is important to notice that query modification does not change the non-procedurality of the query. It is the responsibility of the second phase to translate the non-procedural specification into a procedural one. We call the generated plan a *query evaluation plan (QEP)* [FG86]. Besides being procedural, the QEP also incorporates knowledge about the physical representation of relations in terms of base tables and indexes that can speed up the access to data, and might include operators such as sorting or creating temporary tables. Furthermore, for operations like join, the QEP specifies what methods to use among different alternatives. Example 2 shows one possible QEP for the modified query of Example 1 assuming that an index is present on the attribute DEPT.Dept#.

Example 2:

The following QEP is one possibility for the evaluation of the query in Example 1. We use algebraic (i.e. relational-algebra like) operators as introduced in [Fre87] to express the QEP as follows:

```
(PROJECT (EMP.Name, DEPT.Mgr)
  (NLJOIN
    (FILTER (DEPT.Dept# < 1000)
            (ACCESS DEPT))
    (GET EMP
         (ACCESS (Dept# = DEPT.Dept#)
                 D_INDEX))))
```

Table DEPT is accessed before filtering out all tuples with a department value more or equal to 1000. The resulting set of tuples is the outer input stream for the nested-loop join operator. The inner input stream is generated by first accessing the D_INDEX using the join predicate before retrieving the tuple from the EMP data table. Finally, the join result is projected onto the required attribute values. □

Throughout this paper we concentrate on the second kind of optimization and refer to it as *query optimization*, i.e. the translation of the non-procedural query specification into a procedural one. Furthermore it is important to notice that there exists a fundamental difference

on how both phases are usually performed. Query modification rewrites the initial query in a straightforward manner without considering alternatives. On the other hand, query optimization explores different alternative QEPs for the same query and chooses one of them as the best candidate for later execution. To create different alternatives, to compare them and to select one of them in an efficient way, makes query optimization especially complicated.

It is worth noting some additional differences between query modification which many researcher consider as "high-level optimization", and query translation which is often viewed as a kind of "low-level optmization". As the latter kind has been studied extensively since the time relational systems were built is is understood quite well. The former kind of optimization has been studied less extensively; despite many results there is no agreed way how to structure or to perform query modification. It is therefore impossible to come up with a "standard model" for this kind of optimization. On the other hand, because of it's well defined nature and its well understood structure we undertake this task for query translation in the rest of this paper.

2.2 The Principles of Optimization

To describe the basic principles, of any query optimization algorithm, we need to identify the "basic building blocks" that underly all algorithms. In [Fre87] we argue that query optimization is a specialized *program transformation* problem. Based on this view, we identified three major, independent aspects that, in our opinion, characterize any optimization algorithm:

- The QEP Generation
- The Search Strategy
- The Cost Function

The aspect of QEP creation describes the transformation, that is it determines the source language (specifying the initial query) and the target language (expressing QEPs), and defines how to create QEPs in the defined target language from the initial query specified in the source language. The target language usually reflects aspects of the execution (runtime) environment in which the QEP is evaluated. If we assume an algebraic form of QEPs as in [Fre87], the set of available operators determines the evaluation possible of a submitted query. For example, the physical representation of relations by base tables, indexes, hash tables, etc. determines the variety of *access operators* used in QEPs. Furthermore, operators implementing different join methods or allowing the creation of temporary tables and indexes, expand the possible repertoire of QEPs.

Since a user-submitted query can usually be evaluated by many different QEPs we need to describe in which order to create the different choices to find a good candidate. That is we must define how to *enumerate* or *search through* the set of possible QEPs. This aspect is described by the *search strategy*. Based on enumeration problems in other areas, the research literature describes a variety of search strategies which we shall survey in the next subsection.

Finally, cost functions provide the basis for comparing different QEPs and for choosing the best plan for execution. They try to estimate the "effort" that is necessary for the execution

of a given QEP by using information about the resources that are later consumed. Cost functions reflect various aspects of the execution environment, such as distribution, cpu consumption, sizes of tables, I/O costs etc. Again, in the next subsection we describe different cost functions used in different implementations of DBMSs.

3 Analysis of Optimization Algorithms

Based on the characterization presented in the previous section, this section analyzes different optimization algorithms that have been described in the literature. We discuss a large variety of algorithms to show that the three "fundamental" principles are the basic building block of all of them. Not all algorithms might fully describe all three principles, but they clearly support the validity of our model.

3.1 QEP Creation

As already outlined in the previous section there are two major aspects that characterize the QEP "creation"[1]: First, the kind of QEPs that the optimization algorithm creates and second the steps that are performed to obtain a QEP from an initial query specification.

In many ways the functional capabilities of the PDBP strongly determines the QEPs created. For example, the PDBP in SYSTEM R (called RSS) supports the access of at most one index during a relation scan [S+79]. For a more efficient evaluation of queries it would be advantageous to access indexes independent of the data tables thus creating more efficient QEPs[2]. For example, Rosenthal and Reiner describe an optimizer that creates more complex (and hopefully more efficient) QEPs than the optimizer for SYSTEM R [RR82].

On the other hand, SYSTEM R accepts *sargable* predicates (SARGs). That is, a predicate (conjunction of terms of the form "column comparison_op value") can be passed to the PDBP (function shipping) which then returns only those tuples that satisfy the predicate [S+79]. Other systems, such as INGRES [WY76], do not provide this capability; they have to evaluate predicates outside the PDBP.

Recent developments of DBMSs allow to retrieve more than one (flat) tuple at a time. For example, in EXODUS and DASDB, the PDBP returns one "complex object", i.e. a hierarchical composition of tuples, on each call [C+86a] [P+87]. Again, these more powerful interfaces also influence the operations in the QEPs created.

Besides the functional capabilities of the PDBP, the set of available operators that manipulate sets of tuples once they are retrieved from stored relations, greatly influences the efficiency of the created QEP. SYSTEM R and R*, for example, provide two join methods: nested-loop join and merge-join. In their evaluation of the R* query optimizer Mackert and Lohman point out that alternative join methods, such as semijoins, joins using hashing, and joins using temporary

[1] We purposely avoid the term "generation" since this term will be used later in this subsection to describe one way of obtaining a QEP from an initial query specification.

[2] This improvement is based on the manipulation of internal tuple identfiers (or TIDs). The interested reader is referred to [RR82].

indexes could be beneficial in R*'s execution environment [ML86]. Other join methods such as nested-block-join [Kim80] have been suggested in the literature. The semijoin operator is, for example, available in SDD-1 [B+81], DDM [C+83], and – less obvious – in University-INGRES [WY76]. Of course, for the evaluation of queries in a distributed environment, a "ship" operator is needed to move data from one site to another [L+84] [Fre87].

The second aspect of QEP creation describes the steps that lead from the initial query specification to a valid QEP. In SYSTEM R/R* the optimizer first considers all "reasonable" access paths for single relations before successively adding other relations to the (partial) QEP created so far, using different join methods [S+79] [SA80]. We described the most important steps of the QEP creation in SYSTEM R by transformation rules in [Fre87]. Notice, that the SYSTEM R/R* optimizer creates QEPs with no composite inners for joins and no temporary indexes.

On the other hand, University-INGRES transforms the initial query graph into a single node while creating the QEP [WY76]. In any final plan, semi-joins always precede regular joins. SDD-1 follows a two-phase processing strategy. The first phase, called *reduction phase* consists of zero, one, or more semijoins, before the second phase combines the intermediate results by several joins at one site to produce the final output [B+81].

There are other important aspects that influence the QEP creation process in a central and distributed execution environment, such as fragments, the presence of replicas, or the network topology. We do not discuss details of these aspects in this paper. More details for distributed DBMSs can be found in an overview by C. Mohan [Moh84].

In our opinion, Rosenthal and Reiner provide the most complete and detailed description of creating QEPs for a "database stored as flat files and Codasyl networks" [RR82]. They perform the creation by *refinement steps* that transform the initial query specification into the different choices for QEPs.

The latter work emphasizes another aspect of QEP creation that distinguishes the different known optimization algorithms. Algorithms such as the one for SYSTEM R/R* *generate* QEPs "bottom-up", that is, from a set of relations and a set of predicates the algorithm successively builds a QEP as described above. On the other hand, optimization algorithms in INGRES and SDD-1 *transform*, or *modify* an initial query graph (representing the structure of the query) until no further"graph" transformations are possible. The latter approach is favored by Rosenthal and Helman in [RH86]. They argue that it is easier to verify the correctness of those algorithms that transform query-graphs, than the ones based on the generation principle.

Finally, we mention that several researchers recently proposed to use *rewriting rules* to describe the aspects of query processing more precisely and implementation independent [Bat87] [GD87] [Loh88] [Fre87]. In particular, Batory describes the implementation of a LDBP, that is, different query processing algorithms, in a uniform way using rewriting rules [Bat87]. Graefe and DeWitt base their implementation of a *query optimizer generator* on rewriting rules thus making it possible to easily change the creation of QEPs [GD87].

3.2 Search Strategies

The search strategy is the "driving force" that determines how the QEP creation proceeds. Most search strategies fall into two main classes: exhaustive strategies and heuristic strategies. Search strategies in general have been well studied and applied to problems in operations research, game theory and in other AI areas in the context of learning and understanding.

Exhaustive strategies generate all plans possible by the creation procedure. Most commonly known are the *depth-first search* strategy and *breadth-first search* strategy. For example, SYSTEM R and R* implement a breadth-first search combined with a dynamic programming approach to avoid the redundant creation of (partial) QEPs [S+79].

The advantage of finding the globally optimal QEP is counterbalanced by the amount of time necessary to create the (possibly exponential number of) QEPs. Therefore, different optimization algorithms suggest different **heuristics** to shorten the search. The *greedy heuristic* is the most commonly known **domain-independent** heuristic that limits the creation of QEPs on each level to one choice. It is used, for example, by Chang [Cha83]. On the other hand, **domain-dependent** heuristics express "rules of thumb" in the application domain. In the context of query optimization an example of such rules are: "whenever possible perform selection and projection before any join" and "perform all joins before executing any cartesian product". Many other heuristics have been proposed all of which we cannot discuss in this paper. As another example we mention the heuristic proposed by Krishnamurthy, Boral, and Zaniolo [KBZ86]. They order the relations for joins according to some ranking function. For distributed query optimization Yu and Chang propose a heuristic that reduces the number of sites and fragments during QEP creation [YC83]. However, Mackert and Lohman argue strongly against the use of any heuristics [ML86]. Based on their performance study of the R* optimizer they increase their "commitment to modeling the various plans in sufficient detail to predict performance correctly, rather than assuming away complexities with simplifying heuristics" [ML86].

One can argue that restricting the set of QEPs that can possibly be created, embodies a heuristic strategy too. We already mentioned in the previous subsection that SYSTEM R and R* do not allow composite inners, and that SDD-1 and University-INGRES perform semi-joins before any joins. This restriction already reduces the number of possible choices significantly [S+79] [L+84] [B+81] [WY76].

Recently other search strategies have been proposed. Ioannidis and Wong investigate a probabilistic search strategy that is based on the principle of simulated annealing [IW87]. Their motivation for such a search strategy stems from their interest in optimizing logic queries whose expected large number of joins does not allow an exhaustive search for an efficient QEP. In [Sel86] Sellis investigates another alternative search strategy, the A* algorithm, a well-known search strategy from artificial intelligence, for optimizing multiple queries. Although these search strategies were used for specialized query processing problems, its seems likely that both search strategies can be applied to more conventional query optimization too as shown in [YL87] for the latter strategy.

Based on the above discussion of this subsection it is important to notice that there are

three major aspects common to (almost) all search strategies. First, any search strategy needs to determine when the search should terminate. Second, the search strategy has to decide where to continue the search when several choices are given. Finally, the search strategy might discard certain choices thus not considering them for further exploration. This latter step might be "empty", that is not present, in some of the search strategies.

3.3 Cost Functions

As already mention in Section 2 cost functions associate a "value" with a given QEP that reflects the expense at which the QEP can be executed. Based on the calculated cost values the search strategy performs various operations that finally leads to one QEP chosen to be the best for evaluation.

Since the optimizer bases its decision of the best QEP on the values computed by the cost functions, it is *crucial* that those closely approximate the real "cost" when actually executing the QEP. For example, SYSTEM R linearly combines the number of I/Os and the number of RSS calls to model I/O cost and CPU cost, respectively [S+79] [ML86]. Besides replacing the number of RSS calls by the actual number of instructions executed, R* also considers the number of messages to be sent (to account for the overhead of initiating a data transfer) and the amount of data transferred [L+84] [ML86]. Cost functions in several distributed DBMSs, such as SDD-1 [B+81], do not take into account the cpu and I/O cost assuming that a slow network dominates the overall cost anyway. On the other hand, the results of Selinger/Adiba and Mackert/Lohman show that the local processing cost contribute significantly to the overall execution costs in a distributed execution environment [SA80] [ML86]. The validation work by Mackert and Lohman also shows how well the cost functions reflect the cost of later execution [ML86]. We refer the interested reader to their paper for a more detailed discussion on cost functions.

4 An Operational Model

After the analysis of different optimization algorithms according to the three fundamental aspects of creation, search strategy, and cost function, it remains to show how we can combine them for the implementation of an optimizer component. Does there exist a "*practical*" and "*simple*" model of optimization that clearly maintains the separation between these three aspects for modularity reasons, but that also intertwines them and shows their interaction? We believe that such a model exists as we shall explain in the remainder of this section. The model is independent of the approach taken to optimize the query, that is, it can serve as a basis for either the generation approach or the transformation approach. To keep our explanations simple and clear, we prefer to describe the model informally. The model can immediately be used as a general architecture for implementing a modular and extensible query optimizer.

4.1 The Search Tree

We base our optimization model on a tree structure represented by a directed graph (V, E). The graph representation of the space searched so far, is by no means new. However, since it forms the basic "data structure" for the optimization model, we give the structure an interpretation suitable for our context.

The tree represents the "progress" made during the optimization process. Each vertex $s \in V$ represents a *state* during the creation of a QEP. Depending on the optimization algorithm the "contents of the state" (i.e. the representation of the state) might vary. For example, in SYSTEM R each state is characterized by a (possibly empty) set of relations and a (possibly empty) set of predicates (selection and join predicates) both of which still need to be included into the final (complete) QEP, and a (partial) QEP that has been created so far [S+79][3]. On the other hand, in University-INGRES the state information consists of a query graph that reflects the progress made during QEP creation, and the (partial) QEP created so far [WY76]. Of course, both systems also differ in their internal representation of QEPs.

Any two vertices (states) s_1, s_2 are connected in the graph by an *edge* $e \in E$ if there exists a "basic creation step" that derives s_2 from s_1. For example, using the transformation rules to describe the QEP creation, a basic creation step consists of applying *one* rule to the state s_1 creating s_2.

Typically, we distinguish between *initial states* that represent "starting points" for any QEP creation, *intermediate states*, i.e. states that are reached during the creation process and from which the creation still continues, and *final states*, i.e. states that represent "end points" for the QEP creation providing a (valid) QEP for the initial query specification. In the directed graph these states are characterized by only outgoing edges, by incoming and outgoing edges, and by only incoming edges, respectively.

4.2 The Optimization Model

Based on the discussion of Section 3, the optimization model consists of three major components: the *creation component*, the *costing component*, and the *search strategy component*. The latter consists of three subcomponents that implement the decision when to terminate, where to continue the search, and which states to prune from the tree and the newly created states. We call these three subcomponents the *Tester*, the *Decider*, and the *Updater*, respectively.

The creation component, also called the *Creator*, takes as its input the search tree created so far, and a state that is to be expanded next. It's output is a set of new states that form the input to the costing component, also called *Cost Evaluator*, that determines the cost of each new choice. During the next step the *Updater* decides to eliminate zero, one, or more states from the tree and/or from the set of newly created states before appending the remaining new states to the search tree. The decision for eliminating states depends on the algorithm implemented. It

[3] The state presentation might become more complicated in case of an SQL "HAVING" clause and an SQL "GROUP BY" clause. However, in principle the state presentation remains the same.

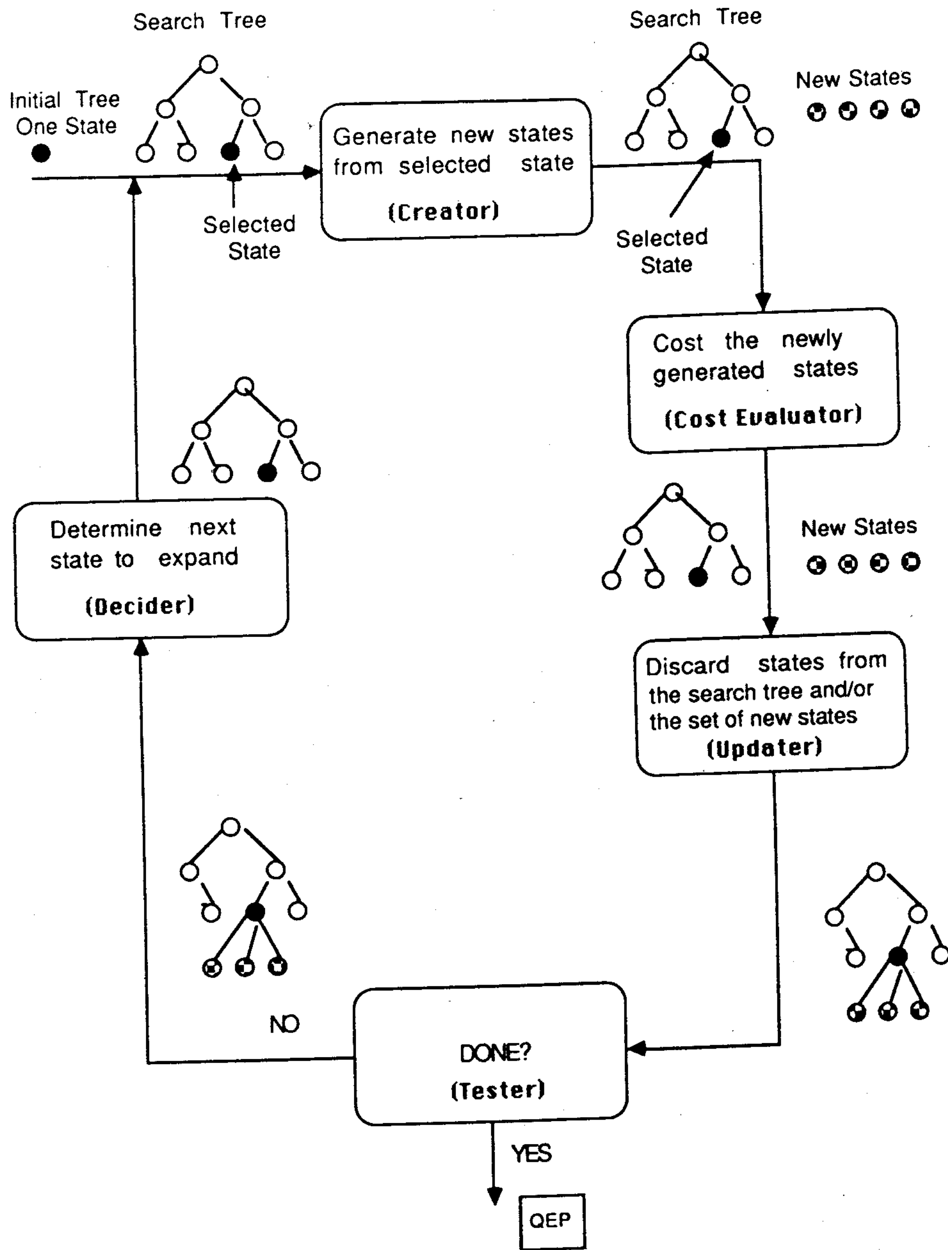

Figure 1: A Graphical View of the Optimization Model

can be based on either comparing costs of different QEPs created so far, on detecting duplicate QEPs (states), or on determining that a partial QEP will not create an efficient (complete) QEP.

Since the search tree has changed the *Tester* might decide to either terminate the search and output one QEP, or to continue the creation process. In case the search continues, it is the responsibility of the *Decider* to determine the next state that is expanded.

Our abstract description of the optimization process demonstrates that the five components work quite independently of each other. The knowledge about the representation of the states and of the search space has been "localized". That is, the Decider, the Tester, and the Updater only depend on the representation of the search tree, but they do not need to know the detailed representation of a state itself. The only relevant information needed from a state are the cost values of (partially or fully) generated QEPs.

On the other hand the Creator depends only on the representation of the states; no knowledge about the search tree representation or the representation of cost functions and values is necessary. This separation and independence is advantageous in case a new search strategy should be implemented, cost functions should be changed, or rewriting rules are changed, deleted, or added. All of these changes can be performed in the corresponding component without affecting others.

4.3 Two Examples

Although we discussed different optimization algorithms in Section 3, we would like to map two of them onto our general architecture to demonstrate its validity. In case of SYSTEM R/R* the Creator either generates different QEPs for accessing one relation considering the different access paths available and the predicates that are applicable, or adds to an existing (partial) QEP another relation by creating a join operator and possibly considering different join methods, such as a nested loop join, an index join, or a sort-merge join. The Cost Evaluator then determines the cost of each newly created (or extended) QEP using the cost functions as explained in [S+79], [L+84], and [ML86].

The Updater implements the dynamic programming aspect of the SYSTEM R/R* optimizer. That is, depending on properties of already existing or newly created plan, the optimizer stores only *one* plan, i.e. the cheapest, for each "property class". All other plans (in our model represented as states) are eliminated.

The Decider checks if all paths in the search tree have been explored. If so, the final plan is output, otherwise the search continues in a breadth-first manner. While SYSTEM R/R* used a generating optimization algorithm, University-INGRES implemented a transformation approach to query optimization using the Greedy search heuristic. Initially, University-INGRES builds a *query graph* from a user-submitted query consisting of nodes and labeled edges. Each node represents a relation in the query. Edges between different nodes are called *join edges* and are labeled with join predicates; edges that point whose origin and target is the same nodes represent *restriction edges* that are labeled with one-variable predicates.

During query optimization this graph is successively reduced to one node while creating

a complete QEP at the same time. By collapsing different pairs of nodes of the query graph into one, the Creator determines several alternative (partial) QEPs whose execution costs are estimated by the Cost Evaluator. After costing the different choices, the Updater eliminates all choices except the cheapest one. If the query graph has been reduced to one node, the Tester terminates the search and outputs the created QEP. Otherwise the creation continues with the one node left by the Updater, thus reducing, the Decider to an "empty component" in INGRES.

4.4 Implementation

We believe that our model provides a well founded basis for implementing an extensible optimizer. One can change the search strategy, the cost functions, or the QEP creation depending on new or changing requirements without affecting all subcomponents of the optimizer. The work by M. J. Massimilla [Mas83] who implemented a translator from relational calculus into relational algebra with an optimizing component strongly influenced our model. His translator incorporates three different search strategies: depth-first search, breadth-first search, and k-step-look-ahead. The latter search strategy allows the creation of all choices for k levels before choosing one state. If $k = 1$ then the k-step-look-ahead degenerates to a greedy algorithm. All three search strategies were implemented by a few lines of code. They could simply be exchanged for each other without requiring any additional changes.

The initial prototype of the rule-based query optimizer for the Starburst project at the IBM Almaden Research Center also reflects this general model of query optimization. The prototype which is discussed in more detail in [Loh88] and [LFL88], clearly demonstrates how the basic principles of query optimization can be implemented in (relatively) independent components thus leading to an **extensible** optimizer.

To explore other ways of implementing the above model we recently began to build an extensible, rule-based query optimizer in Prolog. This choice of implementation language allows us to implement this component rather quickly as a prototype. Currently the system consists of three search methods: The Greedy algorithm, the bread-first algorithm with dynamic programming, and the "k-step look-ahead, choose one alternative" algorithm. We are about to include other search methods such as simulated-annealing, and more sophisticated access-selection methods; the latter aspect is implemented by changing the set of rules that drive the QEP-creation process. Again, our effort shows that the above model represents a sound basis for implementing query optimizers in a flexible manner.

We also recognize that the optimizer – to some extend – resembles a specialized "expert system". Thus, one could have taken an"expert system" approach to develop an architecture that satisfies the extensibility requirements. We are about to study expert system methods and techniques to determine if they contribute to a better understanding (and better implementation) of an optimizer. However, we believe that the current model for query optimization and the derived architecture already represents a major step forward towards an extensible optimization component.

5 Conclusion

In this paper we discussed the fundamental aspects of query optimization that underly all optimization algorithms known in the research literature. We demonstrated that our analysis leads to a simple operational model for query optimization that incorporates the modularity and flexibility necessary to implement an extensible query optimizer as required for the new generation of DBMSs. We also point out several research efforts that validate our model in practice by implementing query optimizers for a new generation of DBMSs.

Although we did not discuss the requirements for the new application domains it should be clear that the adaptation of the existing algorithms and techniques in query optimization to other areas can only be successful if we understand the fundamental dimensions of today's query optimization algorithms. The results of this paper will help to extend the existing optimization techniques to the demanding requirements of new application areas, such as expert systems, deductive database systems, or CAD/CAM systems.

6 Acknowledgement

I would like to thank Laura Haas and Guy Lohman of the Starburst team at the IBM Almaden Research Center for many fruitful discussions that helped to form the initial ideas for this paper. Jean-Marie Nicolas at ECRC provided me with the time and resources to finish up this work.

References

[A+76] M. Astrahan et al. SYSTEM R: Relational Approach to Database Management. *ACM Transactions of Database Systems*, 1(2):97–137, June 1976.

[B+81] P.A. Bernstein et al. Query Processing in a System for Distributed Databases (SDD-1). *ACM Trans. on Database Systems*, 6(4):602–625, Dec. 1981.

[B+86] D.S. Batory et al. Genesis: A reconfigurable database management system. Technical Report 86-07, Department of Computer Sciences, The University of Texas, Austin, 1986.

[Bat87] D.S. Batory. A Molecular Database Systems Technology. Technical Report 87-23, Department of Computer Sciences, The University of Texas, Austin, 1987.

[C+83] A. Chan et al. Overview of an ADA Compatible Distributed Database Manager. In *Proceedings VLDB 1983, Florence, Italy*, pages 354–363, August 1983.

[C+86a] M. Carey et al. Object and File Management in the EXODUS Extensible DBMS. In *Proceedings VLDB 1986, Kyoto, Japan*, pages 91–100, August 1986.

[C+86b] M. Carey et al. The Architecture of the EXODUS Extensible DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, Asilomar, Pacific Grove, California, September 1986.

[Cha83] J.-M. Chang. A Heuristic Approach to Distributed Query Processing. In *Proceedings ACM SIGMOD 1983, San Jose, CA*, pages 54–61, May 1983.

[DS85] U. Dayal and J. Smith. Probe: A Knowledge Oriented Database Management System. In *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[FG86] J.C. Freytag and N. Goodman. Rule-Based Translation of Relational Queries into Iterative Programs. In *Proceedings ACM SIGMOD 1986, Washington, D.C.*, pages 206–214, May 1986.

[Fre87] J.C. Freytag. A Rule-Based View of Query Optimization. In *Proceedings ACM SIGMOD 1987, San Francisco, CA*, pages 173–180, May 1987.

[GD87] G. Graefe and D. DeWitt. The EXODUS Optimizer Generator. In *Proceedings ACM SIGMOD 1987, San Francisco, CA*, pages 160–172, May 1987.

[IW87] Y. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. In *Proceedings ACM SIGMOD 1987, San Francisco, CA*, pages 9–22, May 1987.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16,2, June 1984.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *Proceedings VLDB 1986, Kyoto, Japan*, pages 128–137, August 1986.

[Kim80] W. Kim. A New Way to compute the Product and Join of Relations. In *Proceedings ACM SIGMOD 1980, Santa Monica, CA*, May 1980.

[L+84] G. Lohman et al. Query Processing in R*. Technical Report RJ 4272, Almaden Research Center, San Jose, CA, April 1984.

[LFL88] M.K Lee, J.C. Freytag, and G. Lohman. Implementing an Interpreter for Functional Rules in a Query Optimizer. In *Proceedings VLDB 1988, Los Angeles, CA*, pages 218–229, August 1988.

[Loh88] G. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proceedings ACM SIGMOD 1988, Chicago, IL*, pages 18–27, June 1988.

[Mas83] M.J. Massimilla. A Modularized Translator and Optimizer. Master's thesis, Harvard College, Cambridge, MA, May 1983.

[ML86] L. Mackert and G. Lohman. R* Optimizer and Performance Evaluation for Distributed Queries. In *Proceedings VLDB 1986, Kyoto, Japan*, pages 149–159, August 1986.

[Moh84] C. Mohan. Recent and Future Trends in Distributed Data Base Management. In *Proceedings NYU Symposium on New Directions for Data Base Systems*, May 1984.

[P+87] H.-B. Paul et al. Architecture and Implementation of the Darmstadt Database Kernel System. In *Proceedings ACM SIGMOD 1987, San Francisco, CA*, pages 196–207, May 1987.

[RH86] A. Rosenthal and P. Helman. Understanding and Extending Transformation-Based Optimizers. *IEEE Bulletin on Database Engineering*, 9(4):44–51, December 1986.

[RR82] A. Rosenthal and D. Reiner. An Architecture For Query Optimization. In *Proceedings ACM SIGMOD 1982, Orlando, FL*, June 1982.

[S+79] P.G. Selinger et al. Access Path Selection in a Relational Database Management System. In *Proceedings ACM SIGMOD 1979, Boston, MA*, June 1979.

[S+86] P. Schwarz et al. Extensibility in the Starburst Database System. In *Proceedings of the Asilomar Workshop on Object-Oriented Database Systems*, September 1986.

[SA80] P.G. Selinger and M. Adiba. Access Path Selection in a Distributed Database Management System. In *Proceedings International Conference on Data Bases, Univ. of Aberdeen*, pages 204–215, July 1980.

[Sel86] T.K. Sellis. Global Query Optimization. In *Proceedings ACM SIGMOD 1986, Washington, D.C.*, pages 191–205, May 1986.

[SR86] M. Stonebraker and L. Rowe. The Design of Postgres. In *Proceedings ACM SIGMOD 1986, Washington, D.C.*, pages 340–355, May 1986.

[WY76] E. Wong and K. Youseffi. Decomposition - A Strategy for Query Processing. *ACM Trans. on Database Systems*, 1(3):223–241, Sept. 1976.

[YC83] C.T. Yu and C.C. Chang. On the Design of a Query Processing Strategy in a Distributed Database Environment. In *Proceedings ACM SIGMOD 1983, San Jose, CA*, May 1983.

[YC84] C.T. Yu and C.C. Chang. Distributed Query Processing. *ACM Computing Surveys*, 16,4, December 1984.

[YL87] H. Yoo and S. Lafortune. Distributed Query Processing as Problem Solving. Technical Report CRL-TR-07-87, Computing Research Lab., The University of Michgan, Ann Arbor, MI, 1987.