Principles of probabilistic query optimization
Waas, F.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Model for the Problem

As we have seen in Figure 1.1, we distinguish different phases in the processing of a query. The first step, the rewriting and simplification of the original, declarative query is, technically speaking, already part of the process of improving the query. At this stage, the query is typically specified in a declarative language like SQL. The term query optimization, however, usually refers to the subsequent process of converting the result of the rewriting phase into a cost-effective, if possible cost optimal, procedural query execution plan which can be evaluated by the query engine. The resulting plan consists of operators of the relational algebra which implement set theoretic operations but also contains extensions to facilitate non-algebraic operations like sort orders or the selection of the first $n$ elements of a sorted set. For a detailed discussion of aspects of relational algebra operators see e.g. [AHV95]. The number of extensions varies from one database product to another as does the expressiveness these operators.

The question what an "optimal" execution plan is cannot possibly be answered in simple terms, but will rather accompany us throughout the whole work. We will see different facets of this question in different contexts. However, for the moment an intuitive notion of the optimization goal is satisfactory. Clearly, the effort it takes the engine to evaluate the plan should be as little as possible.\(^1\) On the other hand, we do not want to invest too much time in the optimization. The case that optimization time dominates the query processing costs, which may occur especially for small queries, must be avoided. Thus a trade-off is sought, which takes all costs arising in the whole process into account.

In today's commercial database systems different techniques are deployed to assure a certain balance between the components. IBM's DB2 for example gives the user the possibility to switch off the standard adjustments and control the effort put in the optimization himself. This can be beneficial if the user is a seasoned database expert, but may be diffic-

\(^1\)Though optimization goals other than response time have been proposed in the literature, the response time is clearly the predominant objective.
cult to manage by the lesser experienced user leading to poor results. In
the Microsoft SQL Server, developers pursued a different approach using
several timeout controlled levels of optimization that produce simple and
usually sub-optimal fall back solutions before investigating further optim-
ization possibilities with more and more sophisticated implementations.
As opposed to the previous case, no user adjustment is permitted and—in
theory—not needed either.

Keeping those practical details in mind we now focus on the optimi-
zation process itself. Relentless efforts in this area provided a wealth
of research papers addressing all kinds of relational algebra operators
and their use as well as specific optimization techniques. It turned out
early, that joins, the algebraic operator which combines two database
tables—essentially building the cross product and simultaneously applying
a filter—, is one of the operators most crucial to the optimization process.
Not only does it rank among the most expensive operations in terms of ex-
ecution costs, regarding both time and resource consumption, it is also one
of the most frequently used operations. Every non-trivial query uses more
than one table, i.e., requires a join operator in general. Other expensive op-
erations include sorting and aggregation operators, however, joins usually
outnumber them by far. Therefore, focusing on the join ordering problem is
a restriction not uncommon in this field of research [IK84, IK90, GLPK94].
In the further course of this work, we will address the limited problem as
well as the unrestricted, general case. Specifically, we will first address the
simplified model and try a step-by-step transfer of the knowledge obtained
to establish an understanding of the complex case. The following example
illustrates the basic concepts.

2.1 Example

Consider a database used to maintain a college's organization consisting
of tables that store information about professors, students, and courses
(see Fig. 2.1). We omitted further attributes and show only the information
used later on. Underscored attributes are keys, e.g. title of a course. For
professors and students we store an id and the name. The courses table
comprises title and the id of the lecturer who is giving this course. The last
table, contains the information what student is enrolled in what course.

To find out what professors the student "Sam White" meets the next
term, we pose the following query:

```sql
SELECT *
FROM  Professors P, Students S, Enrolled E, Courses C
WHERE S.Name = "Sam White" AND
      S.SID = E.SID AND
      E.Title = C.Title AND
      C.By = P.PID
```
2.1. EXAMPLE

We translate this query into a procedural execution plan using binary join operators that combine tables pairwise applying a predicate as a filter. This corresponds theoretically to building the Cartesian product of both tables and filtering the result afterwards. Practically, however, a join can be implemented in more sophisticated ways using indexes, sort orders or hash tables. For a detailed discussion of possible implementation techniques see for instance [Gra93]. Figures 2.2a and 2.2b show two possible execution plans for this query. The plans are to be read bottom-up. The leaves of the trees are the base tables as shown in Figure 2.1; the numbers on the edges denote the sizes—i.e., the number of rows—of tables and intermediate results, respectively. That is, in Figure 2.2a, Enrolled and Courses are joined first under the condition E.Title=C.Title. The resulting intermediate table has 131739 rows. This is combined with the table Professors and finally joined with the result of filtering the student table. For simplicity we assume there is only one single student named Sam White. The result table, at the root of the tree, contains only 8 rows. Plan 2.2b differs only in so far as the tables are joined in an alternative order.

Although we focus on join ordering, we usually need additional operators for instance to restrict Students to “Sam White”, in our example. Such restrictions are applied as early as possible, thus not interfering with the join ordering problem. That is, restrictions as well as other operators do not need to be taken care of when determining a join order. This proceeding is not only a common assumption in previous work, but also practice in commercial systems.

While semantically equivalent, the plans differ in sizes of intermediate results as indicated. The sizes relate to the costs of a plan since they reflect the amount of work that has to be done at each operator as well as total resource allocation, e.g. main memory, etc. Accordingly, plan 2.2b is preferable.


2.2 Formal Model

For a formal description of the problem, we decompose it into three separate components, *join graphs, execution plans, and cost computation*.

2.2.1 Join Graphs

Given a query in a declarative query language like SQL, we identify a *join* or *query graph* $G(V, E)$. The nodes $V$ represent the tables involved in the query. The edges $E$ denote which tables are to be combined by a join predicate [Ull89]. Figure 2.3 shows the join graph for the previous example. Every query corresponds to exactly one join graph.

In the literature, certain shapes of join graphs received special attention for two reasons. Firstly, certain shapes are characteristic for certain applications. Secondly, for database evaluation, synthetic workloads are often used where the shape of the query graph is used as parameter for the random generation of queries.

A common restriction is to consider tree shaped graphs only, i.e., graphs without cycles, since the majority of queries in practice are of this kind. Among the tree shaped queries, further differentiation is applied. Star graphs where all nodes are to be joined with one single, distinguished node occur in data warehousing applications when one, usually very large, fact table is combined with additional information. Another frequently encountered shape is the chain graph in which all nodes have degree less or equal
to 2. Chain graphs are typical for path expressions for instance in object oriented database systems (see e.g. [KM94]), but also database systems that use vertical fragmentation [Bat79, CK85, BK99]. The combination of star and chain graph is known as snowflake graph where chains are adjacent to one center node. The chains themselves may also contain further stars.

In the field of cyclic query graphs, further differentiation is not common, though certain patterns can be identified from a geometric point of view, such as grids etc. However, they do not correspond to frequently encountered queries in practice. In practical examples of cyclic queries, the graphs can be easily decomposed to a tree shaped one with 1 or 2 additional edges. The graph theoretic notion of connectivity is not very helpful either, since connectivity larger than 1 means every node is part of at least one cycle. Besides, for queries of trivial sizes such a scenario is rather unrealistic. Particularly, the pattern of a clique graph where every node is connected with every other node is—again trivial size excluded—very unlikely to correspond to any real-world query.

2.2.2 Execution Plans

Execution plans are also often referred to as execution tree, processing plans, or processing trees. We will use the term synonymously in the remainder of this work.

The following definition describes valid processing trees formally:

**Definition 2.2.1**

Let $G(V,E)$ be a query graph. A valid processing tree $t$ over $G$ is a binary tree with $|V|$ leaves such that:

a) the leaves of $t$ correspond to the nodes of $G$

b) the leaves of every subtree of $t$ form a connected component in $G$. $\diamond$

![Figure 2.3. Join graph](image-url)
The set of all possible execution plans spans the search space. The size of the search space grows exponentially in the size of $G$ due to combinatorial explosion. For a more detailed study of this issue in combination with tree shaped graphs see [GLPK94, GLPK95]. The problem of efficiently determining the search space size in case the underlying graph is cyclic, is unsolved yet, i.e., it is unknown whether this problem is $\#P$-complete.

We introduce execution plans as binary trees. Other models have been proposed and actual implementation may use $n$-ary trees. However, from the complexity point of view, $n$-ary joins can always be decomposed into a set of binary joins with the same complexity. The agglutination to $n$-ary joins is used in multi-stage optimization where for example at one stage the particular join order is irrelevant or not determined yet. Such proceeding is helpful in transformation-based optimizers, where all stages of the optimization use fully assembled execution plans—not yet optimized join clusters are carried on to the next stage as $n$-ary joins.

A further general restriction we seem to have applied in the above definition is the limitation to trees. This is due to the direct translation of relational algebra operators which, like any other functional expression, have a tree shaped evaluation graph. In the context of object-oriented databases, Kemper et al. suggested cyclic execution plans where data can bypass a number of operators reducing execution time [KMPS94]. The decision how to split data streams is made at run time. This technique can be particularly beneficial when other join operators are bypassed [SPMK95]. In practice, the cases where these methods apply are rare however. Furthermore, using these so called bypass plans poses a difficult problem for the execution engine: for tree shaped plans, very efficient evaluation techniques have been developed and refined. The assumption that plans are tree shaped is unfortunately a necessary prerequisite for their functioning. If the execution plan is a DAG, the execution paradigm needs major modifications. In [6], we proposed a general model for this problem together with an extension of common evaluation techniques that overcomes this problem. In the present work, however, we will not consider these cases since they are not yet of any practical relevance.

Another scenario exceeding the limitation to trees is encountered in parallel database systems. Data streams are partitioned for executing partial plans in parallel. Usually, the parallel parts are identical in their function so that the optimizer can handle them like one single plan executed on hardware with accordingly scaled parameters [14]. Such simplification can be simply taken care of by the costing techniques without influencing the actual optimization process. This notion of transparent parallelism can be extended to plan segments of arbitrary size [13, 11, 10, 9] as well as to arbitrary shapes of the plan [8]. In either case, the techniques proposed allow a reduction to a sequential, tree shaped plan during optimization.
2.2.3 Cost Functions

We indicated above that different execution plans require different amounts of effort to be evaluated. The objective function for the query optimization problem assigns every execution plan a single non-negative value. This value is commonly referred to as costs in the query optimization business, but also in the context of other combinatorial optimization problems where the objective function is to be minimized.

Cost functions belong to the core of proprietary code of a database vendor. Their accurate tuning and alignment with all other database components requires a high level of expertise and knowledge of both hardware and database components. Cost modeling has become a line of research in its own right over the last decade [Yao77, Bat86, AKK95, ZL96, LN96, BF97, BMK99, BMK00].

In this section, we briefly describe the basic components as far as necessary for the understanding of the concepts presented later. We will not go into the gory details, not only because it would lead too far away, but also because we want to identify common schemas on a level of abstraction which allows a generalization—one of the questions we will have to face later is of course as to whether our generalization is justified and particularly where its limits are.

Cost functions can be split into two parts: a logical component that deals with the analysis of the algorithms used, and a physical one that reflects characteristics of the hardware deployed.

Logical Cost Component

Due to the strong encapsulation offered by relation algebra operators, the particular algorithms that implement different operators can be analyzed largely independently. A first criterion in terms of costs is the operator's complexity—i.e., the algorithm that implements its functionality—in the classical sense of complexity theory. Most unary operators are in $O(n)$, like selections or $O(n \log n)$, like sorting; $n$ being the size of the input in the form of a table. Binary operators can be in $O(n)$ like the union of sets that does not eliminate duplicates, or, more often, in $O(n^2)$, as for instance join operators. The strong limitation to polynomials of low degree will be of importance when we discuss a generalization of the concepts in Section 4.

Though the relational operators input and output tuples, there is usually no direct break down of tuples to the storage schemes used in database systems. Thus, the granularity needs to be enlarged to cover pages of either the underlying operating system's memory management or, and this is the usual case in commercial database systems, the pages of the database's own buffer management. As a result, costs are computed in terms of page or buffer I/Os.

The standard database literature provides a large variety of costing formulas on the basis of buffer I/O for the most frequently used oper-
ators and their implementations. We refer the interested reader e.g. to [KS91, EN94, KE96]

The costs of all operators used in one execution plan are computed in a bottom-up fashion, and summed up to obtain a total cost value. This schema may be slightly violated if sideway information passing, for instance in the form of bit filters, is used, where costs may also be propagated top-down within usually small parts of the plan (see e.g. [CHY93]). Such situations need a special treatment. In all other cases, an operator's cost represent the total cost of the partial execution plan rooted in it leading to a hierarchical decomposition of the plan. We will discuss this property with respect to the optimization process below.

Physical Cost Component

Obviously, the time needed for writing or reading a given set of pages may take different times in different hardware environments. The resulting time is the sum of the times the different subsystems needed—from buses to disk latencies etc. Those figures do not scale easily from one system to another since bottlenecks shift too.

It seems tempting to separate all sensitive hardware parameters and use them for an instantiation of a general costing schema. That is, in order to transfer the costing technique to a new environment, all we need is the set of hardware specific figures. After instantiating the optimization framework with this set, the optimizer generated should need no further adaption to the new environment. This approach has been pursued in Exodus and its successor Volcano [GD87, GM93].

2.2.4 Complexity

Join ordering as a sub-problem of query optimization has been proven NP-hard under the restriction of using only linear execution plans by Ibaraki and Kameda [IK84]. Cluet and Moerkotte have proven a more general case, where also cartesian products are considered to be NP-hard too [CM95]. Finally, Scheufele and Moerkotte established the proof for the general case where cartesian products are allowed and no limitations are imposed on the shape of the execution plan [SM97]. This includes also trees that consist of cartesian products only—a problem which may seem easier than join ordering at first sight.

2.2.5 Some Practical Considerations

In order to achieve a reasonable trade-off between the time spent on the optimization and the quality of the result, we have to relax the optimization goal and try to find a solution of acceptable quality rather than the absolute optimum [Swa89a].
2.3. QUERY OPTIMIZATION TECHNIQUES

However, another point, often neglected, is the fact that costing a query plan involves uncertainty caused by the cost model which is based on estimates in two ways:

1. In the example above, we pointed out that the number of qualifying tuples has direct impact on the costs. Cost functions in real database systems, however, have to consider far more parameters than only the table and intermediate result sizes, since the costs aim to give as accurate as possible an assessment how long it takes to execute the query on a certain hardware configuration. Those parameters reflect the way data is stored on disk (e.g. sorted, clustered, or indexed), different kinds of join implementations (e.g. nested loop, merge sort or hash joins) as well as hardware parameter (e.g. I/O bandwidth, CPU clock speed and type). All those values are obtained by calibration, i.e., they are means determined experimentally. A number of important parameters like cache misses and alike are difficult if not impossible to incorporate into a model [BMK99]. On the other hand, modeling too many details, or details at too fine a granularity may contribute to larger errors.

2. In the example, we gave the exact sizes of the intermediate results, tacitly omitting the fact that this information is not available during the optimization phase but only after the execution of the query. What is, however, available at the time of optimization are statistics about the data. Based on statistics about the base tables, the selectivities of the joins—i.e., the ratio of qualifying to non-qualifying rows—are estimated. Clearly, joins which operate not only on base tables but the output of preceding joins will be affected also by estimation errors, which occurred earlier. Those estimation errors compound exponentially throughout the execution plan [IC91].

The consequence is a limited resolution of the cost function. That is, cost differences of a few per cent only are insignificant, since the numerical error may outweigh them—we cannot determine which of them will actually be executed quicker. Consequently, the tree with the least estimated costs is not necessarily the optimal tree. To allow for this deficiency we consider two plans \( t_1 \) and \( t_2 \) of similar quality if their costs \( c(t_1) \) and \( c(t_2) \) differ less than the resolution \( \rho \), i.e., \( |c(t_1) - c(t_2)| \leq \rho \). Practically, the optimization goal shifts to find a plan with costs less than \( c_{min} + \rho \). We will discuss the role of \( \rho \) and suitable values for it later.

2.3 Query Optimization Techniques

Standard techniques in today’s commercial database systems are based on dynamic programming and heuristics. Stochastic techniques are still confined to research prototypes. In this section we give a brief overview on
the major representatives in the field. We will extend and analyze some of them later in more detail.

2.3.1 Exhaustive Optimization

Exhaustive methods, often also referred to as exact methods, are based on a partial enumeration of the search space. The techniques we describe here are based on dynamic programming. Though the term 'dynamic programming' is often used to reference the bottom-up enumeration schema used in IBM DB2, we will use this term in its general meaning [Ber87].

MEMO-based Enumeration

Using the hierarchical decomposition into subtrees as sketched above, the optimal plan consists of optimal subtrees. By optimal subtrees we mean the tree with least cost of all equivalent trees. We sketch the approach pursued in Microsoft SQL Server and Tandem Non-Stop SQL. Both descended from the research prototype Cascades which in turn is an enhancement of the Volcano optimizer [McK93, GM93, BMG93, GCD '94, Gra94].
2.3. QUERY OPTIMIZATION TECHNIQUES

Since this type of optimizer is transformation based, a first initial plan must be provided by the preprocessing phase. Such an initial plan can be generated as a canonical form where all tables referenced in the From clause are joined (cf. n-ary joins above) without specifying physical join operators, together with a subsequent application of select and aggregate operators. Subqueries if they have not been un-nested in the preprocessing are separately transformed and become a subtree of the initial plan. Consider a join query that joins three tables A, B, and C. A simple initial plan is given in Figure 2.4. The initial plan consists of logical operators that describe only the algebraic properties of the operator but do not contain any implementation details.

Every operator belongs to a group of equivalent subtrees, the root nodes of equivalent subtrees. During a copy-in phase, the operators of the initial tree are assigned to the respective groups. The original links between operators are substituted by references to groups. The system of groups is called MEMO Structure in the following [GCD*94]. Groups are identified by their group number. The group containing the root operator of the initial tree is referred to as root group. To facilitate the description of further operation we label operators in the MEMO with an id tag of the form (groupno.id) relative to the group. References to the groups that implement possible subtrees of an operator are given by the number in the lower right corner of the operator. We anticipate a reorganization of the system of groups in our example and arrange the group numbers in a way that allows for additional groups in a graphic way.

Once the initial plan is copied into the MEMO, the actual optimization process commences. According to a control strategy that includes the aforementioned timeout and fall-back mechanisms, different sets of rules are applied to the operators in the MEMO. The concept of rules is based on the work by Freytag [Fre87] and has been advanced and refined McKenna et al. [McK93, BMG93]. Often, rules are also referred to as transformations. We will use both terms in this work.

A rule, when applied to a logical operator verifies a set of conditions that must be fulfilled for a successful transformation. Conditions include type and algebraic properties of the operator—but may also include specifications concerning its children—, physical properties like sort orders on certain attributes, cost bounds, etc. Provided all conditions of a rule are fulfilled an alternative for the original operator is generated. The result of a rule application can be:

- a logical operator in the same group, e.g. \( \text{join}(A,B) \rightarrow \text{join}(B,A) \);
- a physical operator in the same group, e.g. \( \text{join} \rightarrow \text{hash join} \);
- a set of logical operators that form a connected sub-plan; the root goes to the original group, other operators may go to any group, including the creation of new groups as necessary, e.g. \( \text{join}(A,\text{join}(B,C)) \rightarrow \text{join(\text{join}(A,B),C)} \).
The resulting operators may reference any available group in the MEMO as new children. The MEMO framework provides scheduler primitives for the rule application as well as mechanisms to detect and eliminate multiple entries. Moreover, in every group the currently cheapest operator—and thus the subtree rooted in it—is marked up.

Figure 2.5 shows the MEMO structure form the previous example after applying several transformations, now filled with both logical and physical operators. An execution plan must consist of physical operators only. The optimization ends as soon as no new operators can be derived or secondary stopping criteria like timeouts are fulfilled.

The root group is distinguished as all operators in it are possible root operators for the final execution plan—i.e., they encode an alternative execution plan. Conversely no operator of any other group can be root of the final execution plan.

Thus, the least costly physical operator in the root group is the root of the final execution plan. Its children are recursively determined by selecting the least costly physical operator of its child groups, observing the physical properties such as sort orders. In Figure 2.5 a possible final execution plan is indicated by darkened arrows.

<table>
<thead>
<tr>
<th>Group</th>
<th>Operation</th>
<th>Group</th>
<th>Operation</th>
<th>Group</th>
<th>Operation</th>
<th>Group</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Join 7.1</td>
<td>3</td>
<td>Join 2.2</td>
<td>8</td>
<td>Join 4.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Join 2.2</td>
<td>2</td>
<td>Join 6.3</td>
<td>5</td>
<td>Join 5.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Join 6.3</td>
<td>5</td>
<td>Join 5.3</td>
<td>4</td>
<td>Join 5.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Join 5.2</td>
<td>4</td>
<td>Join 5.2</td>
<td>3</td>
<td>Join 3.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Scan C</td>
<td>3</td>
<td>HashJoin 3.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TableScan 2.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TableScan 1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 2.5. MEMO structure representing alternative solutions](image)
System R

In System R a similar approach is implemented. However, instead of using an initial plan and applying transformations, the plans are generated bottom-up from scratch. The resulting sub-plans are stored in a lookup table comparable to the MEMO. The optimizer of System R and its successors, Starburst and DB2 have been scrutinized concerning their complexity and both Pellenkof et al. as well as Vance and Maier showed that it actually does not meet its theoretically lower bounds. Vance and Maier proposed an enhanced version to overcome these drawbacks [VM96]. The improvements by Pellenkof et al. solve the problem in the context of a transformation based framework as sketched above [PGLK97a, PGLK97b].

2.3.2 Non-exhaustive Optimization

Ibaraki and Kameda suggested an algorithm that computes the optimal join order with nested-loop joins for acyclic query graphs under certain assumption concerning the cost function [IK94]. These restriction are usually too strong for practical application. However, the basic elements of this technique can be transferred to the more general case on the expense of optimality. Such a transfer can be of interest for generating initial solutions for transformation based optimizers.

Similar to the previous approach, Krishnamurthy et al. proposed a ranking technique, named KBZ, that computes the optimal join order under a set of conditions different from the one above [KBZ86]. Again, the conditions are not fulfilled in practical cases in general but as mentioned before, a relaxation of some of the restrictions provides a heuristic. Steinbrunn et al. established a quantitative assessment using this heuristic for optimizing large join queries. Their results suggest decreasing performance with increasing query size [SMK97].

Swami and Lyer modified the previous algorithms and introduced a random component altering the nature of the algorithm substantially as we will see later [SI93]. Their new algorithm, called AB outperforms the original KBZ in almost every case.

2.3.3 Probabilistic Optimization

During the last decade probabilistic algorithms gained popularity in many fields of combinatorial optimization. Especially in fields like VLSI layout, where they are part of a user feedback controlled refinement process, such techniques belong to the standard repertoire. The basic setup in query optimization differs in two vital details from the one in VLSI layout: Queries have to be optimized in (a) on-line fashion and (b) without user feedback. As opposed to off-line applications a reliable self tuning mechanism is required.
Most probabilistic algorithms can be proven to converge to the optimum, an infinite amount of running time permitted, but the conditions ensuring success are not useful in practical applications; in a user feedback controlled process, much more aggressive tuning can be applied leading to enormous performance improvements, in terms of running time.

One of the main advantages of probabilistic algorithms, however, is their black box or blind search approach: The algorithm alters query plans only by means of a set of transformations and checks the resulting quality by calling the cost function. It does not need any further knowledge of the semantic of the changes. That way, the optimization strategy is highly extensible and can be adapted to any kind of set of relational operators.

This ground has been broken notably by Ioannidis et al. [IW87, IK90, IK91], and Swami et al. [SG88, Swa91, SI93]. A detailed discussion of their work is given in Chapter 7.

Also genetic algorithms have been applied though delivering mediocre results only. We will scrutinize the major approaches [BFI91, SS96, SMK97] in Section 6.

### 2.4 Summary

Query optimization is an NP-hard combinatorial problem, which has been the subject of numerous probes during the last two decades. As a result of this process, the view on it of theorists and practitioners differ substantially: Academic researchers regard it largely as solved, database implementors experience the opposite on an almost daily basis [Cha97].

The reason for this difference of opinion is the difference in the models for the problem. In academic research usually some interesting aspects of the problem are separated from a larger context and analyzed, tackled, and solved in isolation. However, the integration of the solution back into the original setup usually has to sacrifice efficiency and effectiveness of the new approach to a considerable degree.

Practitioners, on the other hand, often seem too much concerned about minor technical details loosing the view for the picture as a whole. Consequently, the state of the art in commercial products did not progress significantly over the last decade.

In this chapter, we outlined the basic ingredients of the problem. Especially the multitude of different implementations of relational operators together with the uncertainties involved by the cost computation make the transfer of theoretical results particularly difficult. Exhaustive algorithms based on dynamic programming are therefore the algorithms of choice in commercial database systems to date. Though randomized algorithms can exceed the strict limitations of dynamic programming and may be used to solve problem instances of way larger sizes, they have not been employed in comprehensive query optimization as many effects are not fully understood yet.