Query Optimization

Yannis E. Ioannidis*

Computer Sciences Department

University of Wisconsin

Madison, WI 53706

yannis@cs.wisc.edu

1 Introduction

Imagine yourself standing in front of an exquisite buffet filled with numerous delicacies. Your goal is to try them all out, but you need to decide in what order. What exchange of tastes will maximize the overall pleasure of your palate?

Although much less pleasurable and subjective, that is the type of problem that query optimizers are called to solve. Given a query, there are many plans that a database management system (DBMS) can follow to process it and produce its answer. All plans are equivalent in terms of their final output but vary in their cost, i.e., the amount of time that they need to run. What is the plan that needs the least amount of time?

Such *query optimization* is absolutely necessary in a DBMS. The cost difference between two alternatives can be enormous. For example, consider the following database schema, which will be

*Partially supported by the National Science Foundation under Grants IRI-9113736 and IRI-9157368 (PYI Award) and by grants from DEC, IBM, HP, AT&T, Informix, and Oracle.

used throughout this chapter:

emp(name,age,sal,dno)

dept(dno,dname,floor,budget,mgr,ano)

acnt(ano,type,balance,bno)

bank(bno,bname,address)

Further, consider the following very simple SQL query:

select name, floor

from emp, dept

where emp.dno=dept.dno **and** sal>100K.

Assume the characteristics below for the database contents, structure, and run-time environment:

Parameter Description	Parameter Value
Number of emp pages	20000
Number of emp tuples	100000
Number of emp tuples with sal>100K	10
Number of dept pages	10
Number of dept tuples	100
Indices of emp	Clustered B+-tree on emp.sal (3-levels deep)
Indices of dept	Clustered hashing on dept.dno (average bucket length of 1.2 pages)
Number of buffer pages	3
Cost of one disk page access	20ms

Consider the following three different plans:

P1 Through the B+-tree find all tuples of emp that satisfy the selection on emp.sal. For each one, use the hashing index to find the corresponding dept tuples. (Nested loops, using the index on both relations.)

P2 For each dept page, scan the entire emp relation. If an emp tuple agrees on the dno attribute with a tuple on the dept page and satisfies the selection on emp.sal, then the emp-dept tuple pair appears in the result. (Page-level nested loops, using no index.)

P3 For each dept tuple, scan the entire emp relation and store all emp-dept tuple pairs. Then, scan this set of pairs and, for each one, check if it has the same values in the two dno attributes and satisfies the selection on emp.sal. (Tuple-level formation of the cross product, with subsequent scan to test the join and the selection.)

Calculating the expected I/O costs of these three plans shows the tremendous difference in efficiency that equivalent plans may have. P1 needs 0.32 seconds, P2 needs a bit more than an hour, and P3 needs more than a whole day. Without query optimization, a system may choose plan P2 or P3 to execute this query with devastating results. Query optimizers, however, examine "all" alternatives, so they should have no trouble choosing P1 to process the query.

The path that a query traverses through a DBMS until its answer is generated is shown in Figure 1. The system modules through which it moves have the following functionality:

- The *Query Parser* checks the validity of the query and then translates it into an internal form, usually a relational calculus expression or something equivalent.
- The *Query Optimizer* examines all algebraic expressions that are equivalent to the given query and chooses the one that is estimated to be the cheapest.
- The *Code Generator* or the *Interpreter* transforms the access plan generated by the optimizer into calls to the query processor.
- The *Query Processor* actually executes the query.

Queries are posed to a DBMS by interactive users or by programs written in general-purpose programming languages (e.g., C/C++, Fortran, PL-1) that have queries embedded in them. An interactive (ad hoc) query goes through the entire path shown in Figure 1. On the other hand, an embedded query goes through the first three steps only once, when the program in which it is em-

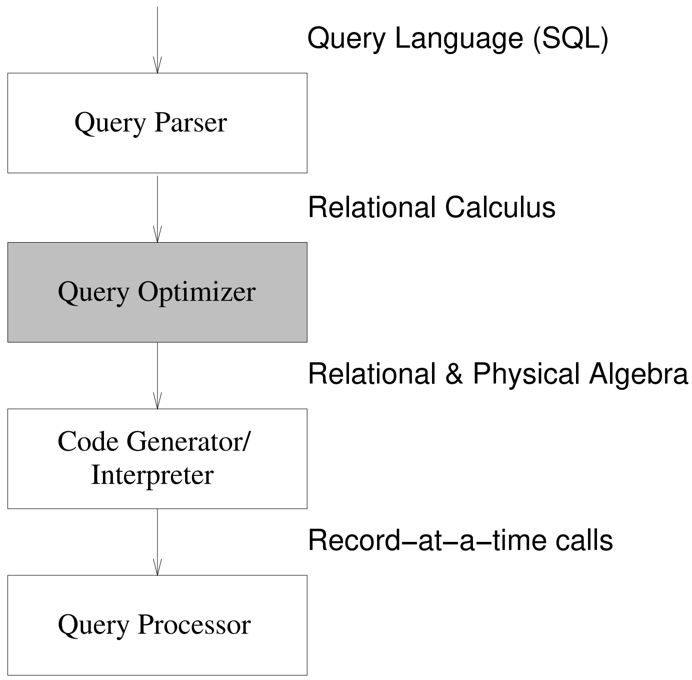

Figure 1: Query flow through a DBMS.

bedded is compiled (*compile time*). The code produced by the Code Generator is stored in the database and is simply invoked and executed by the Query Processor whenever control reaches that query during the program execution (*run time*). Thus, independent of the number of times an embedded query needs to be executed, optimization is not repeated until database updates make the access plan invalid (e.g., index deletion) or highly suboptimal (e.g., extensive changes in database contents). There is no real difference between optimizing interactive or embedded queries, so we make no distinction between the two in this chapter.

The area of query optimization is very large within the database field. It has been studied in a great variety of contexts and from many different angles, giving rise to several diverse solutions in each case. The purpose of this chapter is to primarily discuss the core problems in query optimization and their solutions, and only touch upon the wealth of results that exist beyond that. More specifically, we concentrate on optimizing a single *flat SQL query* with 'and' as the only boolean connective in its qualification (also known as *conjunctive query*, *select-project-join query*, or *nonrecursive Horn clause*) in a centralized relational DBMS, assuming that full knowledge of the run-time environment exists at compile time. Likewise, we make no attempt to provide a complete survey of the literature, in most cases providing only a few example references. More extensive surveys can be found elsewhere [JK84, MCS88].

The rest of the chapter is organized as follows. Section 2 presents a modular architecture for a query optimizer and describes the role of each module in it. Section 3 analyzes the choices that exist in the shapes of relational query access plans, and the restrictions usually imposed by current optimizers to make the whole process more manageable. Section 4 focuses on the dynamic programming search strategy used by commercial query optimizers and briefly describes alternative strategies that have been proposed. Section 5 defines the problem of estimating the sizes of query results and/or the frequency distributions of values in them, and describes in detail histograms, which represent the statistical information typically used by systems to derive such estimates. Section 6 discusses query optimization in non-centralized environments, i.e., parallel and distributed DBMSs. Section 7 briefly touches upon several advanced types of query optimization that have been proposed to solve some hard problems in the area. Finally, Section 8 summarizes the chapter and raises some questions related to query optimization that still have no good answer.

2 Query Optimizer Architecture

2.1 Overall Architecture

In this section, we provide an abstraction of the query optimization process in a DBMS. Given a database and a query on it, several execution plans exist that can be employed to answer the query. In principle, all the alternatives need to be considered so that the one with the best estimated performance is chosen. An abstraction of the process of generating and testing these alternatives is shown in Figure 2, which is essentially a modular architecture of a query optimizer. Although one could build an optimizer based on this architecture, in real systems, the modules shown do not always have so clear-cut boundaries as in Figure 2. Based on Figure 2, the entire query optimization

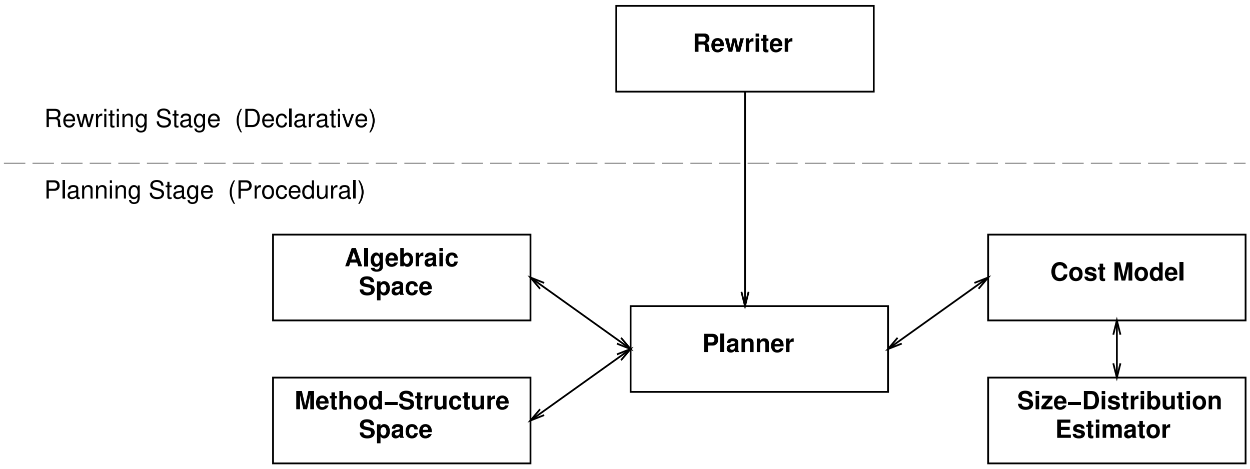

Figure 2: Query optimizer architecture.

process can be seen as having two stages: *rewriting* and *planning*. There is only one module in the first stage, the *Rewriter*, whereas all other modules are in the second stage. The functionality of each of the modules in Figure 2 is analyzed below.

2.2 Module Functionality

Rewriter: This module applies transformations to a given query and produces equivalent queries that are hopefully more efficient, e.g., replacement of views with their definition, flattening out of nested queries, etc. The transformations performed by the Rewriter depend only on the declarative, i.e., static, characteristics of queries and do not take into account the actual query costs for the specific DBMS and database concerned. If the rewriting is known or assumed to always be beneficial, the original query is discarded; otherwise, it is sent to the next stage as well. By the nature of the rewriting transformations, this stage operates at the *declarative* level.

Planner: This is the main module of the ordering stage. It examines all possible execution plans for each query produced in the previous stage and selects the overall cheapest one to be used to generate the answer of the original query. It employs a *search strategy*, which examines the space of execution plans in a particular fashion. This space is determined by two other modules of the optimizer, the *Algebraic Space* and the *Method-Structure Space*. For the most part, these two modules and the search strategy determine the cost, i.e., running time, of the optimizer itself, which should be as low as possible. The execution plans examined by the Planner are compared based on estimates of their cost so that the cheapest may be chosen. These costs are derived by the last two modules of the optimizer, the *Cost Model* and the *Size-Distribution Estimator*.

Algebraic Space: This module determines the action execution orders that are to be considered by the Planner for each query sent to it. All such series of actions produce the same query answer, but usually differ in performance. They are usually represented in relational algebra as formulas or in tree form. Because of the algorithmic nature of the objects generated by this module and sent to the Planner, the overall planning stage is characterized as operating at the *procedural* level.

Method-Structure Space: This module determines the implementation choices that exist for the execution of each ordered series of actions specified by the Algebraic Space. This choice is related to the available join methods for each join (e.g., nested loops, merge scan, and hash join), if supporting data structures are built on the fly, if/when duplicates are eliminated, and other implementation characteristics of this sort, which are predetermined by the DBMS implementation. This choice is also related to the available indices for accessing each relation, which is determined by the physical schema of each database stored in its catalogs. Given an algebraic formula or tree from the Algebraic Space, this module produces all corresponding complete execution plans, which specify the implementation of each algebraic operator and the use of any indices.

Cost Model: This module specifies the arithmetic formulas that are used to estimate the cost of execution plans. For every different join method, for every different index type access, and in general for every distinct kind of step that can be found in an execution plan, there is a formula that gives its cost. Given the complexity of many of these steps, most of these formulas are simple approximations of what the system actually does and are based on certain assumptions regarding issues like buffer management, disk-cpu overlap, sequential vs. random I/O, etc. The most important input parameters to a formula are the size of the buffer pool used by the corresponding step, the sizes of relations or indices accessed, and possibly various distributions of values in these relations. While the first one is determined by the DBMS for each query, the other two are estimated by the Size-Distribution Estimator.

Size-Distribution Estimator: This module specifies how the sizes (and possibly frequency distributions of attribute values) of database relations and indices as well as (sub)query results are estimated. As mentioned above, these estimates are needed by the Cost Model. The specific estimation approach adopted in this module also determines the form of statistics that need to be maintained in the catalogs of each database, if any.

2.3 Description Focus

Of the six modules of Figure 2, three are not discussed in any detail in this chapter: the Rewriter, the Method-Structure Space, and the Cost Model. The Rewriter is a module that exists in some commercial DBMSs (e.g., DB2-Client/Server and Illustra), although not in all of them. Most of the transformations normally performed by this module are considered an advanced form of query optimization, and not part of the core (planning) process. The Method-Structure Space specifies alternatives regarding join methods, indices, etc., which are based on decisions made outside the development of the query optimizer and do not really affect much of the rest of it. For the Cost

Model, for each alternative join method, index access, etc., offered by the Method-Structure Space, either there is a standard straightforward formula that people have devised by simple accounting of the corresponding actions (e.g., the formula for tuple-level nested loops join) or there are numerous variations of formulas that people have proposed and used to approximate these actions (e.g., formulas for finding the tuples in a relation having a random value in an attribute). In either case, the derivation of these formulas is not considered an intrinsic part of the query optimization field. For these reasons, we do not discuss these three modules any further until Section 7, where some Rewriter transformations are described. The following three sections provide a detailed description of the Algebraic Space, the Planner, and the Size-Distribution Estimator modules, respectively.

3 Algebraic Space

As mentioned above, a flat SQL query corresponds to a select-project-join query in relational algebra. Typically, such an algebraic query is represented by a *query tree* whose leaves are database relations and non-leaf nodes are algebraic operators like selections (denoted by σ), projections (denoted by π), and joins[1] (denoted by $\bowtie$). An intermediate node indicates the application of the corresponding operator on the relations generated by its children, the result of which is then sent further up. Thus, the edges of a tree represent data flow from bottom to top, i.e., from the leaves, which correspond to data in the database, to the root, which is the final operator producing the query answer. Figure 3 gives three examples of query trees for the query

> **select** name, floor
>
> **from** emp, dept
>
> **where** emp.dno=dept.dno **and** sal>100K .

[1] For simplicity, we think of the cross product operator as a special case of a join with no join qualification.

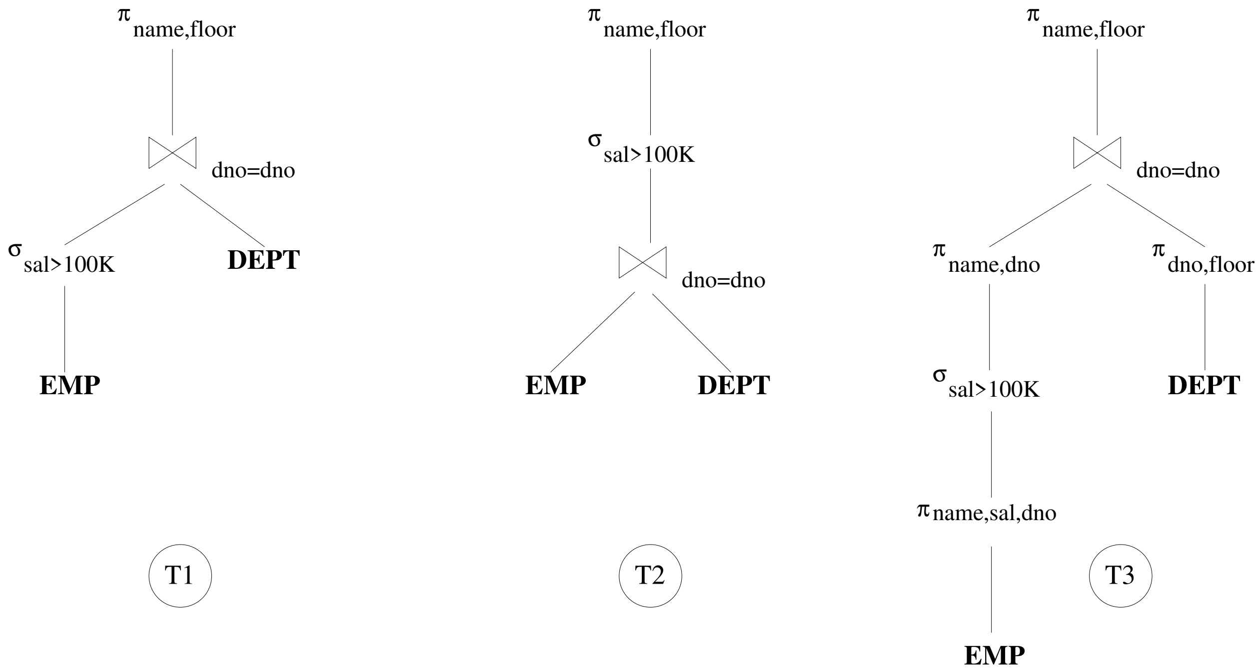

Figure 3: Examples of general query trees.

For a complicated query, the number of all query trees may be enormous. To reduce the size of the space that the search strategy has to explore, DBMSs usually restrict the space in several ways. The first typical restriction deals with selections and projections:

R1 Selections and projections are processed on the fly and almost never generate intermediate relations. Selections are processed as relations are accessed for the first time. Projections are processed as the results of other operators are generated.

For example, plan P1 of Section 1 satisfies restriction R1: the index scan of emp finds emp tuples that satisfy the selection on emp.sal on the fly and attempts to join only those; furthermore, the projection on the result attributes occurs as the join tuples are generated. For queries with no join, R1 is moot. For queries with joins, however, it implies that all operations are dealt with as part of join execution. Restriction R1 eliminates only suboptimal query trees, since separate processing of selections and projections incurs additional costs. Hence, the Algebraic Space module specifies alternative query trees with join operators only, selections and projections being implicit.

Given a set of relations to be combined in a query, the set of all alternative join trees is determined by two algebraic properties of join: commutativity ($R_1 \bowtie R_2 \equiv R_2 \bowtie R_1$) and associativity ($(R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3)$). The first determines which relation will be inner and which outer in the join execution. The second determines the order in which joins will be executed. Even with the R1 restriction, the alternative join trees that are generated by commutativity and associativity is very large, $\Omega(N!)$ for N relations. Thus, DBMSs usually further restrict the space that must be explored. In particular, the second typical restriction deals with cross products.

R2 Cross products are never formed, unless the query itself asks for them. Relations are combined always through joins in the query.

For example, consider the following query:

select name, floor, balance

from emp, dept, acnt

where emp.dno=dept.dno **and** dept.ano=acnt.ano

Figure 4 shows the three possible join trees (modulo join commutativity) that can be used to combine the emp, dept, and acnt relations to answer the query. Of the three trees in the figure,

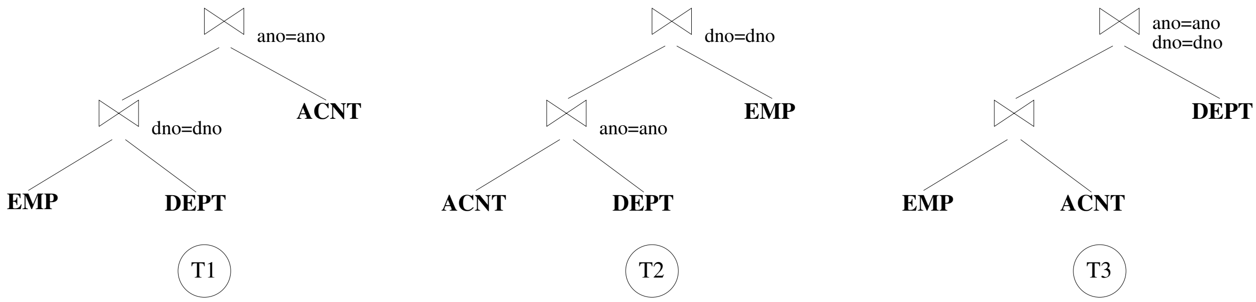

Figure 4: Examples of join trees; T3 has a cross product.

tree T3 has a cross product, since its lower join involves relations emp and acnt, which are not explicitly joined in the query. Restriction R2 almost always eliminates suboptimal join trees, due to the large size of the results typically generated by cross products. The exceptions are very few and are cases where the relations forming cross products are extremely small. Hence, the Algebraic Space module specifies alternative join trees that involve no cross product.

The exclusion of unnecessary cross products reduces the size of the space to be explored, but that still remains very large. Although some systems restrict the space no further (e.g., Ingres and DB2-Client/Server), others require an even smaller space (e.g., DB2/MVS). In particular, the third typical restriction deals with the shape of join trees.

R3 The inner operand of each join is a database relation, never an intermediate result.

For example, consider the following query:

> **select** name, floor, balance, address
>
> **from** emp, dept, acnt, bank
>
> **where** emp.dno=dept.dno **and** dept.ano=acnt.ano **and** acnt.bno=bank.bno

Figure 5 shows three possible cross-product-free join trees that can be used to combine the emp, dept, acnt, and bank relations to answer the query. Tree T1 satisfies restriction R3, whereas trees T2 and T3 do not, since they have at least one join with an intermediate result as the inner relation. Because of their shape (Figure 5) join trees that satisfy restriction R3, e.g., tree T1, are called *left-deep*. Trees that have their outer relation always being a database relation, e.g., tree T2, are called *right-deep*. Trees with at least one join between two intermediate results, e.g., tree T3, are called *bushy*. Restriction R3 is of a more heuristic nature than R1 and R2 and may well eliminate the optimal plan in several cases. It has been claimed that most often the optimal left-deep tree is not much more expensive than the optimal tree overall. The typical arguments used are two:

- Having original database relations as inners increases the use of any preexisting indices.

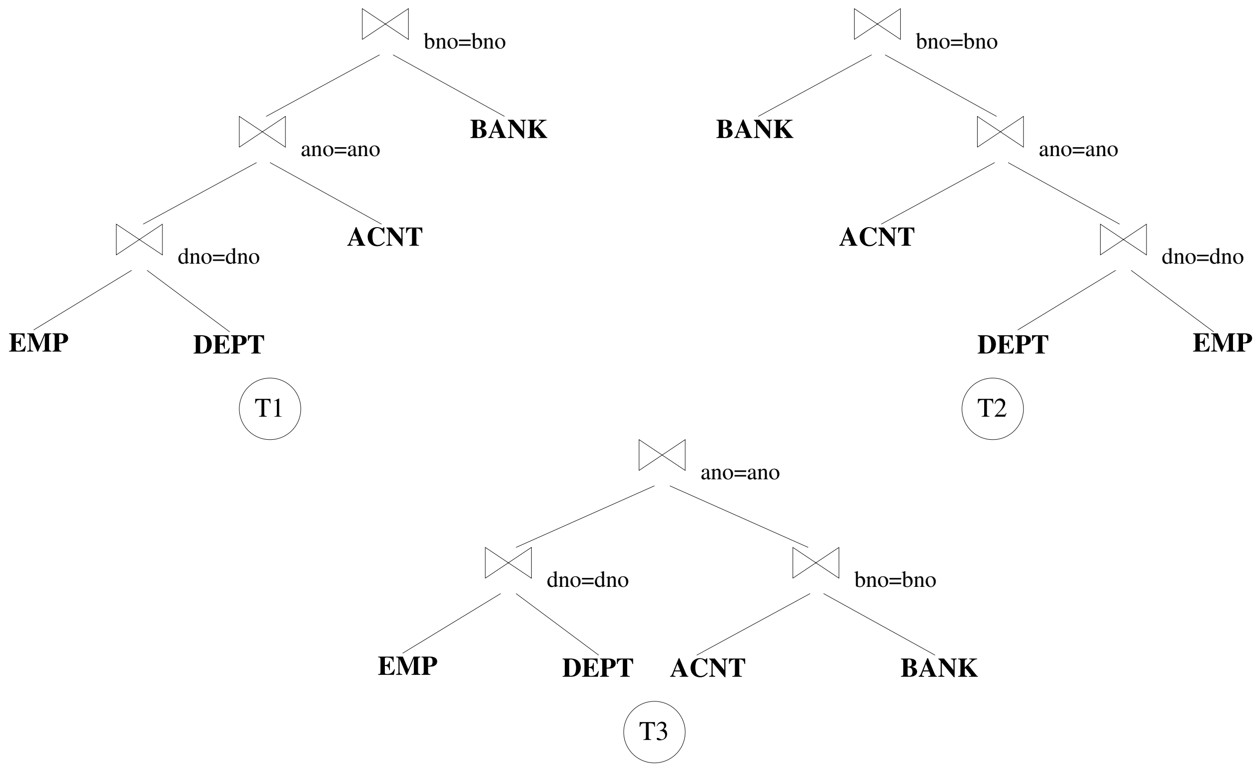

Figure 5: Examples of left-deep (T1), right-deep (T2), and bushy (T3) join trees.

- Having intermediate relations as outers allows sequences of nested loops joins to be executed in a pipelined fashion.[2]

Both index usage and pipelining reduce the cost of join trees. Moreover, restriction R3 significantly reduces the number of alternative join trees, to $O(2^N)$ for many queries with N relations. Hence, the Algebraic Space module of the typical query optimizer only specifies join trees that are left-deep.

In summary, typical query optimizers make restrictions R1, R2, and R3 to reduce the size of the space they explore. Hence, unless otherwise noted, our descriptions follow these restrictions as well.

[2] A similar argument can be made in favor of right-deep trees regarding sequences of hash joins.

4 Planner

The role of the Planner is to explore the set of alternative execution plans, as specified by the Algebraic Space and the Method-Structure space, and find the cheapest one, as determined by the Cost Model and the Size-Distribution Estimator. The following three subsections deal with different types of search strategies that the Planner may employ for its exploration. The first one focuses on the most important strategy, dynamic programming, which is the one used by essentially all commercial systems. The second one discusses a promising approach based on randomized algorithms, and the third one talks about other search strategies that have been proposed.

4.1 Dynamic Programming Algorithms

Dynamic programming was first proposed as a query optimization search strategy in the context of System R [A+76] by Selinger et al. [SAC+79]. Commercial systems have then used it in various forms and with various extensions. We present this algorithm pretty much in its original form [SAC+79], only ignoring details that do not arise in flat SQL queries, which are our focus.

The algorithm is essentially a dynamically pruning, exhaustive search algorithm. It constructs all alternative join trees (that satisfy restrictions R1-R3) by iterating on the number of relations joined so far, always pruning trees that are known to be suboptimal. Before we present the algorithm in detail, we need to discuss the issue of *interesting order*. One of the join methods that is usually specified by the Method-Structure Space module is *merge scan*. Merge scan first sorts the two input relations on the corresponding join attributes and then merges them with a synchronized scan. If any of the input relations, however, is already sorted on its join attribute (e.g., because of earlier use of a B+-tree index or sorting as part of an earlier merge-scan join), the sorting step can be skipped for the relation. Hence, given two partial plans during query optimization, one cannot compare them based on their cost only and prune the more expensive one; one has to also take into account the sorted order (if any) in which their result comes out. One of the plans may be more expensive but may generate its result sorted on an attribute that will save a sort in a subsequent merge-scan execution of a join. To take into account these possibilities, given a query, one defines its *interesting orders* to be orders of intermediate results on any relation attributes that participate in joins. (For more general SQL queries, attributes in order-by and group-by clauses give rise to interesting orders as well.) For example, in the query of Section 3, orders on the attributes emp.dno, dept.dno, dept.ano, acnt.ano, acnt.bno, and bank.bno are interesting. During optimization of this query, if any intermediate result comes out sorted on any of these attributes, then the partial plan that gave this result must be treated specially.

Using the above, we give below a detailed English description of the dynamic programming algorithm optimizing a query of N relations:

Step 1 For each relation in the query, all possible ways to access it, i.e., via all existing indices and including the simple sequential scan, are obtained. (Accessing an index takes into account any query selection on the index key attribute.) These partial (single-relation) plans are partitioned into equivalence classes based on any interesting order in which they produce their result. An additional equivalence class is formed by the partial plans whose results are in no interesting order. Estimates of the costs of all plans are obtained from the Cost Model module, and the cheapest plan in each equivalence class is retained for further consideration. However, the cheapest plan of the no-order equivalence class is not retained if it is not cheaper than all other plans.

Step 2 For each pair of relations joined in the query, all possible ways to evaluate their join using all relation access plans retained after Step 1 are obtained. Partitioning and pruning of these partial (two-relation) plans proceeds as above.

...

Step i For each set of $i-1$ relations joined in the query, the cheapest plans to join them for each interesting order are known from the previous step. In this step, for each such set, all possible ways to join one more relation with it without creating a cross product are evaluated. For each set of i relations, all generated (partial) plans are partitioned and pruned as before.

...

Step N All possible plans to answer the query (the unique set of N relations joined in the query) are generated from the plans retained in the previous step. The cheapest plan is the final output of the optimizer, to be used to process the query.

For a given query, the above algorithm is guaranteed to find the optimal plan among those satisfying restrictions R1-R3. It often avoids enumerating all plans in the space by being able to dynamically prune suboptimal parts of the space as partial plans are generated. In fact, although in general still exponential, there are query forms for which it only generates $O(N^3)$ plans [OL90].

An example that shows dynamic programming in its full detail takes too much space. We illustrate its basic mechanism by showing how it would proceed on the simple query below:

select name, mgr

from emp, dept

where emp.dno=dept.dno **and** sal>30K **and** floor=2

Assume that there is a B+-tree index on emp.sal, a B+-tree index on emp.dno, and a hashing index on dept.floor. Also assume that the DBMS supports two join methods, nested loops and merge scan. (Both types of information should be specified in the Method-Structure Space module.) Note that, based on the definition, potential interesting orders are those on emp.dno and dept.dno, since these are the only join attributes in the query. The algorithm proceeds as follows: *Step 1:* All possible ways to access emp and dept are found. The only interesting order arises from accessing emp via the B+-tree on emp.dno, which generates the emp tuples sorted and ready for the join with dept. The entire set of alternatives, appropriately partitioned are shown in the table below. Each

Relation	Interesting Order	Plan Description	Cost
emp	emp.dno	Access through B+-tree on emp.dno.	$\boxed{700}$
	–	Access through B+-tree on emp.sal.	$\boxed{200}$
		Sequential scan.	600
dept	–	Access through hashing on dept.floor.	$\boxed{50}$
		Sequential scan.	200

partial plan is associated with some hypothetical cost; in reality, these costs are obtained from the Cost Model module. Within each equivalence class, only the cheapest plan is retained for the next step, as indicated by the boxes surrounding the corresponding costs in the table. *Step 2:* Since the query has two relations, this is the last step of the algorithm. All possible ways to join emp and dept are found, using both supported join methods and all partial plans for individual relation access retained from Step 1. For the nested loops method, which relation is inner and which is outer is also specified. Since this is the last step of the algorithm, there is no issue of interesting orders. The entire set of alternatives is shown in the table below in a way similar to Step 1. Based on hypothetical costs for each of the plans, the optimizer produces as output the plan indicated by the box surrounding the corresponding cost in the table.

As the above example illustrates, the choices offered by the Method-Structure Space in addition to those of the Algebraic Space result in an extraordinary number of alternatives that the optimizer must search through. The memory requirements and running time of dynamic programming grow

Join Method	Outer/Inner	Plan Description	Cost
nested loops	emp/dept	• For each emp tuple obtained through the B+-tree on emp.sal, scan dept through the hashing index on dept.floor to find tuples matching on dno.	1800
		• For each emp tuple obtained through the B+-tree on emp.dno and satisfying the selection on emp.sal, scan dept through the hashing index on dept.floor to find tuples matching on dno.	3000
	dept/emp	• For each dept tuple obtained through the hashing index on dept.floor, scan emp through the B+-tree on emp.sal to find tuples matching on dno.	2500
		• For each dept tuple obtained through the hashing index on dept.floor, probe emp through the B+-tree on emp.dno using the value in dept.dno to find tuples satisfying the selection on emp.sal.	$\boxed{1500}$
merge scan	–	• Sort the emp tuples resulting from accessing the B+-tree on emp.sal into L_1. • Sort the dept tuples resulting from accessing the hashing index on dept.floor into L_2. • Merge L_1 and L_2.	2300
		• Sort the dept tuples resulting from accessing the hashing index on dept.floor into L_2. • Merge L_2 and the emp tuples resulting from accessing the B+-tree on emp.dno and satisfying the selection on emp.sal.	2000

exponentially with query size (i.e., number of joins) in the worst case since all viable partial plans generated in each step must be stored to be used in the next one. In fact, many modern systems place a limit on the size of queries that can be submitted (usually around fifteen joins), because for larger queries the optimizer crashes due to its very high memory requirements. Nevertheless, most queries seen in practice involve less than ten joins, and the algorithm has proved to be very effective in such contexts. It is considered the standard in query optimization search strategies.

4.2 Randomized Algorithms

To address the inability of dynamic programming to cope with really large queries, which appear in several novel application fields, several other algorithms have been proposed recently. Of these, randomized algorithms, i.e., algorithms that "flip coins" to make decisions, appear very promising.

The most important class of these optimization algorithms is based on plan transformations instead of the plan construction of dynamic programming, and includes algorithms like Simulated Annealing, Iterative Improvement, and Two-Phase Optimization. These are generic algorithms that can be applied to a variety of optimization problems and are briefly described below as adapted to query optimization. They operate by searching a graph whose nodes are all the alternative execution plans that can be used to answer a query. Each node has a cost associated with it, and the goal of the algorithm is to find a node with the globally minimum cost. Randomized algorithms perform *random walks* in the graph via a series of *moves*. The nodes that can be reached in one move from a node S are called the *neighbors* of S. A move is called *uphill* (resp. *downhill*) if the cost of the source node is lower (resp. higher) than the cost of the destination node. A node is a *global minimum* if it has the lowest cost among all nodes. It is a *local minimum* if, in all paths starting at that node, any downhill move comes after at least one uphill move.

4.2.1 Algorithm Description

Iterative Improvement (II) [NSS86, SG88, Swa89] performs a large number of *local optimizations*. Each one starts at a random node and repeatedly accepts random downhill moves until it reaches a local minimum. II returns the local minimum with the lowest cost found.

Simulated Annealing (SA) performs a continuous random walk accepting downhill moves always and uphill moves with some probability, trying to avoid being caught in a high cost local minimum [KGV83, IW87, IK90]. This probability decreases as time progresses and eventually becomes zero, at which point execution stops. Like II, SA returns the node with the lowest cost visited.

The Two Phase Optimization (2PO) algorithm is a combination of II and SA [IK90]. In phase 1, II is run for a small period of time, i.e., a few local optimizations are performed. The output of that phase, which is the best local minimum found, is the initial node of the next phase. In phase 2, SA is run starting from a low probability for uphill moves. Intuitively, the algorithm chooses a local minimum and then searches the area around it, still being able to move in and out of local minima, but practically unable to climb up very high hills.

4.2.2 Results

Given a finite amount of time, these randomized algorithms have performance that depends on the characteristics of the cost function over the graph and the connectivity of the latter as determined by the neighbors of each node. They have been studied extensively for query optimization, being mutually compared and also compared against dynamic programming [SG88, Swa89, IW87, IK90, Kan91]. The specific results of these comparisons vary depending on the choices made regarding issues of the algorithms' implementation and setup, but also choices made in other modules of the query optimizer, i.e., the Algebraic Space, the Method-Structure Space, and the Cost Model. In general, however, the conclusions are as follows. First, up to about ten joins, dynamic programming is preferred over the randomized algorithms because it is faster and it guarantees finding the optimal plan. For larger queries, the situation is reversed, and despite the probabilistic nature of the randomized algorithms, their efficiency makes them the algorithms of choice. Second, among randomized algorithms, II usually finds a reasonable plan very quickly, while given enough time, SA is able to find a better plan than II. 2PO gets the best of both worlds and is able to find plans that are as good as those of SA, if not better, in much shorter time.

4.3 Other Search Strategies

To complete the picture on search strategies we briefly describe several other algorithms that people have proposed in the past, deterministic, heuristic, or randomized. Ibaraki and Kameda were the ones that proved that query optimization is an NP-complete problem even if considering only the nested loops join method [IK84]. Given that result, there have been several efforts to obtain algorithms that solve important subcases of the query optimization problem and run in polynomial time. Ibaraki and Kameda themselves presented an algorithm (referred to as IK here) that takes advantage of the special form of the cost formula for nested loops and optimizes a tree query of N joins in $O(N^2 logN)$ time. They also presented an algorithm that is applicable to even cyclic queries and finds a good (but not always optimal) plan in $O(N^3)$ time.

The KBZ algorithm uses essentially the same techniques, but is more general and more sophisticated and runs in $O(N^2)$ time for tree queries [KBZ86]. As with IK, the applicability of KBZ depends on the cost formulas for joins to be of a specific form. Nested loops and hash join satisfy this requirement but, in general, merge scan does not.

The AB algorithm mixes deterministic and randomized techniques and runs in $O(N^4)$ time [SI93]. It uses KBZ as a subroutine, which needs $O(N^2)$ time, and essentially execute it $O(N^2)$ times on randomly selected spanning trees of the query graph. Through an interesting separation of the cost of merge scan into a part that affects optimization and a part that does not, AB is applicable to all join methods despite the dependence on KBZ.

In addition to SA, II, and 2PO, Genetic Algorithms [Gol89] form another class of generic randomized optimization algorithms that have been applied to query optimization. These algorithms simulate a biological phenomenon: a random set of solutions to the problem, each with its own cost, represent an initial population; pairs of solutions from that population are matched (*cross-over*) to generate offspring that obtain characteristics from both parents, and the new children may also be randomly changed in small ways (*mutation*); between the parents and the children, those with the least cost (*most fit*) survive in the next generation. The algorithm ends when the entire population consists of copies of the same solution, which is considered to be optimal. Genetic algorithms have been implemented for query optimization with promising results [BFI91].

Another interesting randomized approach to query optimization is pure, uniformly-random generation of access plans [GLPK94]. Trully uniform generation is a hard problem and has been solved for tree queries. With an efficient implementation of this step, experiments with the algorithm have shown good potential, since there is no dependence on plan transformations or random walks.

In the artificial intelligence community, the A* heuristic algorithm is extensively used for complex search problems. A* has been proposed for query optimization as well, and can be seen as a direct extension to the traditional dynamic programming algorithm [YL89]. Instead of proceeding in steps and using all plans with n relations to generate all plans with $n+1$ relations together, A* proceeds by expanding one of the generated plans at hand at a time, based on its expected proximity to the optimal plan. Thus, A* generates a full plan much earlier than dynamic programming and is able to prune more aggressively in a branch-and-bound mode. A* has been proposed for query optimization and has been shown quite successful for not very large queries.

Finally, in the context of extensible DBMSs, several unique search strategies have been proposed, which are all rule based. Rules are defined on how plans can be constructed or modified, and the Planner follows the rules to explore the specified plan space. The most representative of these efforts are those of Starburst [Loh88, H$^+$90] and Volcano/Exodus [GM93, GD87]. The Starburst optimizer employs constructive rules whereas the Volcano/Exodus optimizers employ transformation rules.

5 Size-Distribution Estimator

The final module of the query optimizer that we examine in detail is the Size-Distribution Estimator. Given a query, it estimates the sizes of the results of (sub)queries and the frequency distributions of values in attributes of these results.

Before we present specific techniques that have been proposed for estimation, we use an example to clarify the notion of frequency distribution. Consider the following simple relation *OLYMPIAN* on the left, with the frequency distribution of the values in its Department attribute on the right:

Name	Salary	Department
Zeus	100K	General Management
Poseidon	80K	Defense
Pluto	80K	Justice
Aris	50K	Defense
Ermis	60K	Commerce
Apollo	60K	Energy
Hefestus	50K	Energy
Hera	90K	General Management
Athena	70K	Education
Aphrodite	60K	Domestic Affairs
Demeter	60K	Agriculture
Hestia	50K	Domestic Affairs
Artemis	60K	Energy

Department	Frequency
General Management	2
Defense	2
Education	1
Domestic Affairs	2
Agriculture	1
Commerce	1
Justice	1
Energy	3

One can generalize the above and discuss distributions of frequencies of combinations of arbitrary numbers of attributes. In fact, to calculate/estimate the size of any query that involves multiple attributes from a single relation, multi-attribute joint frequency distributions or their approximations are required. Practical DBMSs, however, deal with frequency distributions of individual attributes only, because considering all possible combinations of attributes is very expensive. This essentially corresponds to what is known as the *attribute value independence assumption*, and although rarely true, it is adopted by all current DBMSs.

Several techniques have been proposed in the literature to estimate query result sizes and frequency distributions, most of them contained in the extensive survey by Mannino, Chu, and Sager [MCS88] and elsewhere [Chr89]. Most commercial DBMSs (e.g., DB2, Informix, Ingres, Sybase, Microsoft SQL server) base their estimation on *histograms*, so our description mostly focuses on those. We then briefly summarize other techniques that have been proposed.

5.1 Histograms

In a *histogram* on attribute a of relation R, the domain of a is partitioned into *buckets*, and a uniform distribution is assumed within each bucket. That is, for any bucket b in the histogram, if a value $v_i \in b$, then the frequency f_i of v_i is approximated by $\sum_{v_j \in b} f_j/|b|$. A histogram with a single bucket generates the same approximate frequency for all attribute values. Such a histogram is called *trivial* and corresponds to making the *uniform distribution assumption* over the entire attribute domain. Note that, in principle, any arbitrary subset of an attribute's domain may form a bucket and not necessarily consecutive ranges of its natural order.

	Histogram H1		Histogram H2	
Department	Frequency in Bucket	Approximate Frequency	Frequency in Bucket	Approximate Frequency
Agriculture	$\boxed{1}$	1.5	$\boxed{1}$	1.33
Commerce	$\boxed{1}$	1.5	$\boxed{1}$	1.33
Defense	$\boxed{2}$	1.5	$\boxed{2}$	1.33
Domestic Affairs	$\boxed{2}$	1.5	2	2.5
Education	1	1.75	$\boxed{1}$	1.33
Energy	3	1.75	3	2.5
General Management	2	1.75	$\boxed{2}$	1.33
Justice	1	1.75	$\boxed{1}$	1.33

Continuing on with the example of the *OLYMPIAN* relation, we present above two different histograms on the Department attribute, both with two buckets. For each histogram, we first show which frequencies are grouped in the same bucket by enclosing them in the same shape (box or circle), and then show the resulting approximate frequency, i.e., the average of all frequencies enclosed by identical shapes.

There are various classes of histograms that systems use or researchers have proposed for estimation. Most of the earlier prototypes, and still some of the commercial DBMSs, use trivial histograms, i.e., make the uniform distribution assumption [SAC+79]. That assumption, however, rarely holds in real data and estimates based on it usually have large errors [Chr84, IC91]. Excluding trivial ones, the histograms that are typically used belong to the class of *equi-width* histograms [Koo80]. In those, the number of consecutive attribute values or the size of the range of attribute values associated with each bucket is the same, independent of the frequency of each attribute value in the data. Since these histograms store a lot more information than trivial histograms (they typically have 10-20 buckets), their estimations are much better. Histogram H1 above is equi-width, since the first bucket contains four values starting from A-D and the second bucket contains also four values starting from E-Z.

Although we are not aware of any system that currently uses histograms in any other class than those mentioned above, several more advanced classes have been proposed and are worth discussing. *Equi-depth* (or *equi-height*) histograms are essentially duals of equi-width histograms [Koo80, PSC84]. In those, the sum of the frequencies of the attribute values associated with each bucket is the same, independent of the number of these attribute values. Equi-width histograms have a much higher worst-case and average error for a variety of selection queries than equi-depth histograms. Muralikrishna and DeWitt [MD88] extended the above work for multidimensional histograms that are appropriate for multi-attribute selection queries.

In *serial* histograms [IC93], the frequencies of the attribute values associated with each bucket are either all greater or all less than the frequencies of the attribute values associated with any other bucket. That is, the buckets of a serial histogram group frequencies that are close to each other with no interleaving. Histogram H1 in the earlier table is not serial as frequencies 1 and 3 appear in one bucket and frequency 2 appears in the other, while histogram H2 is. Under various optimality criteria, serial histograms have been shown to be optimal for reducing the worst-case and the average error in equality selection and join queries [IC93, Ioa93, IP95].

Identifying the optimal histogram among all serial ones takes exponential time in the number of buckets. Moreover, since there is usually no order-correlation between attribute values and their frequencies, storage of serial histograms essentially requires a regular index that will lead to the approximate frequency of every individual attribute value. Because of all these complexities, the class of *end-biased* histograms has been introduced. In those, some number of the highest frequencies and some number of the lowest frequencies in an attribute are explicitly and accurately maintained in separate individual buckets, and the remaining (middle) frequencies are all approximated together in a single bucket. End-biased histograms are serial since their buckets group frequencies with no interleaving. Identifying the optimal end-biased histogram, however, takes only slightly over linear time in the number of buckets. Moreover, end-biased histograms require little storage, since usually most of the attribute values belong in a single bucket and do not have to be stored explicitly. Finally, in several experiments it has been shown that most often the errors in the estimates based on end-biased histograms are not too far off from the corresponding (optimal) errors based on serial histograms. Thus, as a compromise between optimality and practicality, it has been suggested that the optimal end-biased histograms should be used in real systems.

5.2 Other Techniques

In addition to histograms, several other techniques have been proposed for query result size estimation [MCS88, Chr89]. Those that, like histograms, store information in the database typically approximate a frequency distribution by a parameterized mathematical distribution or a polynomial. Although requiring very little overhead, these approaches are typically inaccurate because most often real data does not follow any mathematical function. On the other hand, those based on *sampling* primarily operate at run time [OR86, LNS90, HS92, HS95] and compute their estimates by collecting and possibly processing random samples of the data. Although producing highly accurate estimates, sampling is quite expensive and, therefore, its practicality in query optimization is questionable, especially since optimizers need query result size estimations frequently.

6 Non-centralized Environments

The preceding discussion focuses on query optimization for sequential processing. This section touches upon issues and techniques related to optimizing queries in non-centralized environments. The focus is on the Method-Structure Space and the Planner modules of the optimizer, as the remaining ones are not significantly different from the centralized case.

6.1 Parallel Databases

Among all parallel architectures, the shared-nothing and the shared-memory paradigms have emerged as the most viable ones for database query processing. Thus, query optimization research has concentrated on these two. The processing choices that either of these paradigms offer represent a huge increase over the alternatives offered by the Method-Structure Space module in a sequential environment. In addition to the sources of alternatives that we discussed earlier, the Method-Structure Space module offers two more: the number of processors that should be given to each database operation (*intra-operator parallelism*) and placing operators into groups that should be executed simultaneously by the available processors (*inter-operator parallelism*, which can be further subdivided into *pipelining* and *independent parallelism*). The *scheduling* alternatives that arise from these two questions add at least another super-exponential factor to the total number of alternatives, and make searching an even more formidable task. Thus, most systems and research prototypes adopt various heuristics to avoid dealing with a very large search space. In the two-stage approach [HS91], given a query, one first identifies the optimal sequential plan for it using conventional techniques like those discussed in Section 4, and then identifies the optimal parallelization/scheduling of that plan. Various techniques have been proposed in the literature for the second stage, but none of them claims to provide a complete and optimal answer to the scheduling question, which remains an open research problem. In the segmented execution model, one considers only schedules that process memory-resident right-deep segments of (possibly bushy) query plans one-at-a-time (i.e., no independent inter-operator parallelism). Shekita et al. [SYT93] combined this model with a novel heuristic search strategy with good results for shared-memory. Finally, one may be restricted to deal with right-deep trees only [SD90].

In contrast to all the search-space reduction heuristics, Lanzelotte et al. [LVZ93] dealt with both deep and bushy trees, considering schedules with independent parallelism, where all the pipelines in an execution are divided into phases, pipelines in the same phase are executed in parallel, and each phase start only after the previous phase ended. The search strategy that they used was a randomized algorithm, similar to 2PO, and proved very effective in identifying efficient parallel plans for a shared-nothing architecture,

6.2 Distributed Databases

The difference between distributed and parallel DBMSs is that the former are formed by a collection of independent, semi-autonomous processing sites that are connected via a network that could be spread over a large geographic area, whereas the latter are individual systems controlling multiple processors that are in the same location, usually in the same machine room. Many prototypes of distributed DBMSs have been implemented [BGW+81, ML86] and several commercial systems are offering distributed versions of their products as well (e.g., DB2, Informix, Sybase, Oracle).

Other than the necessary extensions of the Cost Model module, the main differences between centralized and distributed query optimization are in the Method-Structure Space module, which offers additional processing strategies and opportunities for transmitting data for processing at multiple sites. In early distributed systems, where the network cost was dominating every other cost, a key idea has been using semijoins for processing in order to only transmit tuples that would certainly contribute to join results [BGW+81, ML86]. An extension of that idea is using Bloom filters, which are bit vectors that approximate join columns and are transferred across sites to determine which tuples *might* participate in a join so that only these may be transmitted [ML86].

7 Advanced Types of Optimization

In this section, we attempt to provide a brief glimpse of advanced types of optimization that researchers have proposed over the past few years. The descriptions are based on examples only; further details may be found in the references provided. Furthermore, there are several issues that are not discussed at all due to lack of space, although much interesting work has been done on them, e.g., nested query optimization, rule-based query optimization, query optimizer generators, object-oriented query optimization, optimization with materialized views, heterogeneous query optimization, recursive query optimization, aggregate query optimization, optimization with expensive selection predicates, and query optimizer validation.

7.1 Semantic Query Optimization

Semantic query optimization is a form of optimization mostly related to the Rewriter module. The basic idea lies in using integrity constraints defined in the database to rewrite a given query into *semantically equivalent* ones [Kin81]. These can then be optimized by the Planner as regular queries and the most efficient plan among all can be used to answer the original query. As a simple example, using a hypothetical SQL-like syntax, consider the following integrity constraint:

assert sal-constraint **on** emp:

sal>100K **where** job = "Sr. Programmer".

Also consider the following query:

select name, floor

from emp, dept

where emp.dno = dept.dno **and** job = "Sr. Programmer".

Using the above integrity constraint, the query can be rewritten into a semantically equivalent one to include a selection on sal:

select name, floor

from emp, dept

where emp.dno = dept.dno **and** job = "Sr. Programmer" **and** sal>100K.

Having the extra selection could help tremendously in finding a fast plan to answer the query if the only index in the database is a B+-tree on emp.sal. On the other hand, it would certainly be a waste if no such index exists. For such reasons, all proposals for semantic query optimization present various heuristics or rules on which rewritings have the potential of being beneficial and should be applied and which not.

7.2 Global Query Optimization

So far, we have focused our attention to optimizing individual queries. Quite often, however, multiple queries become available for optimization at the same time, e.g., queries with unions, queries from multiple concurrent users, queries embedded in a single program, or queries in a deductive system. Instead of optimizing each query separately, one may be able to obtain a global plan that, although possibly suboptimal for each individual query, is optimal for the execution of all of them as a group. Several techniques have been proposed for global query optimization [Sel88]. As a simple example of the problem of global optimization consider the following two queries:

select name, floor
from emp, dept
where emp.dno = dept.dno **and** job = "Sr. Programmer",

select name
from emp, dept
where emp.dno = dept.dno **and** budget > 1M.

Depending on the sizes of the emp and dept relations and the selectivities of the selections, it may well be that computing the entire join once and then applying separately the two selections to obtain the results of the two queries is more efficient than doing the join twice, each time taking into account the corresponding selection. Developing Planner modules that would examine all the available global plans and identify the optimal one is the goal of global/multiple query optimizers.

7.3 Parametric/Dynamic Query Optimization

As mentioned earlier, embedded queries are typically optimized once at compile time and are executed multiple times at run time. Because of this temporal separation between optimization and execution, the values of various parameters that are used during optimization may be very different during execution. This may make the chosen plan invalid (e.g., if indices used in the plan are no longer available) or simply not optimal (e.g., if the number of available buffer pages or operator selectivities have changed, or if new indices have become available). To address this issue, several techniques [GW89, INSS92, CG94] have been proposed that use various search strategies (e.g., randomized algorithms [INSS92] or the strategy of Volcano [CG94]) to optimize queries as much as possible at compile time taking into account all possible values that interesting parameters may have at run time. These techniques use the actual parameter values at run time, and simply pick the plan that was found optimal for them with little or no overhead. Of a drastically different flavor is the technique of Rdb/VMS [Ant93], where by dynamically monitoring how the probability distribution of plan costs changes, plan switching may actually occur during query execution.

8 Summary

To a large extent, the success of a DBMS lies in the quality, functionality, and sophistication of its query optimizer, since that determines much of the system's performance. In this chapter, we have given a bird's eye view of query optimization. We have presented an abstraction of the architecture of a query optimizer and focused on the techniques currently used by most commercial systems for its various modules. In addition, we have provided a glimpse of advanced issues in query optimization, whose solutions have not yet found their way into practical systems, but could certainly do so in the future.

Although query optimization exists as a field for more than twenty years, it is very surprising how fresh it remains in terms of being a source of research problems. In every single module of the architecture of Figure 2, there are many questions for which we do not have complete answers, even for the most simple, single-query, sequential, relational optimizations. When is it worth to consider bushy trees instead of just left-deep trees? How can one model buffering effectively in the system's cost formulas? What is the most effective means of estimating the cost of operators that involve random access to relations (e.g., nonclustered index selection)? Which search strategy can be used for complex queries with confidence, providing consistent plans for similar queries? Should optimization and execution be interleaved in complex queries so that estimate errors do not grow very large? Of course, we do not even attempt to mention the questions that arise in various advanced types of optimization.

We believe that the next twenty years will be as active as the previous twenty and will bring many advances to query optimization technology, changing many of the approaches currently used in practice. Despite its age, query optimization remains an exciting field.

Acknowledgements: I would like to thank Minos Garofalakis, Joe Hellerstein, Navin Kabra, and Vishy Poosala, for their many helpful comments.

References

[A+76] M. M. Astrahan et al. System R: A relational approach to data management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[Ant93] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proc. IEEE Int. Conference on Data Engineering*, pages 538–547, Vienna, Austria, March 1993.

[BFI91] K. Bennett, M. C. Ferris, and Y. Ioannidis. A genetic algorithm for database query optimization. In *Proc. 4th Int. Conference on Genetic Algorithms*, pages 400–407, San Diego, CA, July 1991.

[BGW+81] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM TODS*, 6(4):602–625, December 1981.

[CG94] R. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 150–160, Minneapolis, MN, June 1994.

[Chr84] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM TODS*, 9(2):163–186, June 1984.

[Chr89] S. Christodoulakis. On the estimation and use of selectivities in database performance evaluation. Research Report CS-89-24, Dept. of Computer Science, University of Waterloo, June 1989.

[GD87] G. Graefe and D. DeWitt. The exodus optimizer generator. In *Proc. ACM-SIGMOD Conf. on the Management of Data*, pages 160–172, San Francisco, CA, May 1987.

[GLPK94] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, randomized join-order selection - why use transformations? In *Proc. 20th Int. VLDB Conference*, pages 85–95, Santiago, Chile, September 1994. (Also available as CWI Tech. Report CS-R9416.).

[GM93] G. Graefe and B. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. IEEE Data Engineering Conf.*, Vienna, Austria, March 1993.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 358–366, Portland, OR, May 1989.

[H+90] L. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HS91] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Proc. 1st Int. PDIS Conference*, pages 218–225, Miami, FL, December 1991.

[HS92] P. Haas and A. Swami. Sequential sampling procedures for query size estimation. In *Proc. of the 1992 ACM-SIGMOD Conference on the Management of Data*, pages 341–350, San Diego, CA, June 1992.

[HS95] P. Haas and A. Swami. Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *Proc. of the 1995 IEEE Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[IC91] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the 1991 ACM-SIGMOD Conference on the Management of Data*, pages 268–277, Denver, CO, May 1991.

[IC93] Y. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM TODS*, 18(4):709–748, December 1993.

[IK84] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM-TODS*, 9(3):482–502, September 1984.

[IK90] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 312–321, Atlantic City, NJ, May 1990.

[INSS92] Y. Ioannidis, R. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *Proc. 18th Int. VLDB Conference*, pages 103–114, Vancouver, BC, August 1992.

[Ioa93] Y. Ioannidis. Universality of serial histograms. In *Proc. 19th Int. VLDB Conference*, pages 256–267, Dublin, Ireland, August 1993.

[IP95] Y. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proc. of the 1995 ACM-SIGMOD Conference on the Management of Data*, pages 233–244, San Jose, CA, May 1995.

[IW87] Y. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 9–22, San Francisco, CA, May 1987.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[Kan91] Y. Kang. *Randomized Algorithms for Query Optimization.* PhD thesis, University of Wisconsin, Madison, May 1991.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proceedings 12th Int. VLDB Conference*, pages 128–137, Kyoto, Japan, August 1986.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Kin81] J. J. King. Quist: A system for semantic query optimization in relational databases. In *Proc. of the 7th Int. VLDB Conference*, pages 510–517, Cannes, France, August 1981.

[Koo80] R. P. Kooi. *The Optimization of Queries in Relational Databases.* PhD thesis, Case Western Reserve University, September 1980.

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

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 18–27, Chicago, IL, June 1988.

[LVZ93] R. Lanzelotte, P. Valduriez, and M. Zait. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proc. of the 19th Int. VLDB Conference*, pages 493–504, Dublin, Ireland, August 1993.

[MCS88] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):192–221, September 1988.

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

[ML86] L. F. Mackert and G. M. Lohman. R^* validation and performance evaluation for distributed queries. In *Proc. 12th Int. VLDB Conf.*, pages 149–159, Kyoto, Japan, Aug 1986.

[NSS86] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. In *Proc. 23rd Design Automation Conference*, pages 293–299, 1986.

[OL90] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th Int. VLDB Conference*, pages 314–325, Brisbane, Australia, August 1990.

[OR86] F. Olken and D. Rotem. Simple random sampling from relational databases. In *Proc. 12th Int. VLDB Conference*, pages 160–169, Kyoto, Japan, August 1986.

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

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM-SIGMOD Conf. on the Management of Data*, pages 23–34, Boston, MA, June 1979.

[SD90] D. Schneider and D. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th Int. VLDB Conference*, pages 469–480, Brisbane, Australia, August 1990.

[Sel88] T. Sellis. Multiple query optimization. *ACM-TODS*, 13(1):23–52, March 1988.

[SG88] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 8–17, Chicago, IL, June 1988.

[SI93] A. Swami and B. Iyer. A polynomial time algorithm for optimizing join queries. In *Proc. IEEE Int. Conference on Data Engineering*, Vienna, Austria, March 1993.

[Swa89] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 367–376, Portland, OR, June 1989.

[SYT93] E. Shekita, H. Young, and K.-L. Tan. Multi-join optimization for symmetric multiprocessors. In *Proc. 19th Int. VLDB Conf.*, pages 479–492, Dublin, Ireland, Aug 1993.

[YL89] H. Yoo and S. Lafortune. An intelligent search method for query optimization by semijoins. *IEEE Trans. on Knowledge and Data Engineering*, 1(2):226–237, June 1989.