Computing Handbook
Information Systems and Information Technology
Heikki Topi, Allen Tucker

Query Optimization

Publication details
Nicolas Bruno
Published online on: 14 May 2014

Accessed on: 12 Oct 2018

PLEASE SCROLL DOWN FOR DOCUMENT

Full terms and conditions of use: https://www.routledgehandbooks.com/legal-notices/terms.

This Document PDF may be used for research, teaching and private study purposes. Any substantial or systematic reproductions, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The publisher shall not be liable for an loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.
11
Query Optimization

11.1 Introduction

Database management systems, or DBMSs for short, let users specify queries using high-level declarative languages such as SQL. In such languages, we write what we want but not how to obtain the desired results. The latter task is delegated to the query optimizer, a component in the DBMS responsible for finding an efficient execution plan to evaluate the input query. The query optimizer searches a large space of alternatives and chooses the one that is expected to be least expensive to evaluate. Query optimization is absolutely necessary in a DBMS because the difference in runtime among alternatives and thus the overhead of a bad choice can be arbitrarily large. In fact, for queries of medium complexity, execution plans producing the same result can run from milliseconds to several hours. To answer a SQL query, a typical DBMS goes through a series of steps, illustrated in Figure 11.1*:

1. The input query is parsed and transformed into an algebraic tree. This step performs both syntactic and semantic checks over the input query, rejecting all invalid requests.
2. The algebraic tree is optimized and turned into an execution plan. A query execution plan indicates not only the operations required to evaluate the input query but also the order in which they are performed and the specific algorithms used at each step.
3. The execution plan is either compiled into machine code or interpreted by the execution engine, and results are sent back to the user.

* Note that DBMSs can cache execution plans for repeating input queries and skip the first two steps until the cached plan becomes invalid due to changes in the database design (e.g., an index deletion) or in the data distribution itself.
Although implementation details vary among specific DBMSs, virtually all optimizers share the same high-level conceptual structure and conduct the search of execution plans in a cost-based manner. More specifically, optimizers assign each candidate plan its estimated cost and choose the plan with the least expected cost for execution. Alternative formulations attempt to minimize other related metrics, such as the time to obtain the first tuple. Query optimization can therefore be viewed as a complex search problem defined by three interacting components, illustrated in Figure 11.2:

1. Search space: The search space characterizes the set of execution plans that can be considered by the optimizer. For scalability, some optimizers restrict the set of alternatives that can be used to evaluate queries.
2. Cost model: The cost model assigns a cost value to every plan in the search space by estimating the resources consumed by the plan. The quality of the resulting plan is only as good as the underlying cost model.
3. Enumeration strategy: The search space can be characterized without specifying how to generate all candidate plans. The enumeration strategy serves this purpose and dictates how to traverse all interesting plans in the search space efficiently.

The task of an optimizer is nontrivial given the large number of execution plans for an input query, the large variance in response time of the plans in the search space, and the difficulty of accurately estimating costs. A desirable optimizer is one for which (i) the search space includes low-cost plans, (ii) the cost model is accurate, and (iii) the enumeration strategy is efficient.

**FIGURE 11.1** Executing SQL queries in a relational database system.

**FIGURE 11.2** Architecture of a query optimizer in a DBMS.
The area of query optimization is very rich within the database field. It has been studied in a great variety of contexts and from many different angles, giving rise to diverse solutions in each case. The purpose of this chapter is to primarily discuss the core architecture of modern query optimizers and only touch upon the wealth of results that exist beyond that. 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 [1,2]. The rest of the chapter is organized as follows. Section 11.2 discusses some choices that exist in the space of execution plans and the restrictions usually imposed by current optimizers to make the whole process more manageable. Section 11.3 explains the problem of estimating the cost of execution plans and describes cardinality estimation in more detail, since it is a crucial component in any optimizer cost model. Section 11.4 focuses on enumeration strategies that have been proposed in the context of commercial query optimizers. Section 11.5 touches upon several advanced types of query optimization that have been proposed to address hard problems in the area and raises some open questions in the field. Finally, Section 11.6 concludes the chapter.

11.2 Search Space

The search space of an optimizer depends on the set of physical operators supported by the execution engine and the set of algebraic equivalences among execution subplans. The main source of diversity within an optimizer arises from the available equivalence-preserving transformations.

11.2.1 Operator Reordering

An important class of transformations exploits the commutativity and associativity properties among operators. We next describe three examples of such transformations.

11.2.1.1 Join Reordering

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 \((R_1 \bowtie R_2 \equiv R_2 \bowtie R_1)\), which determines which relations will be inner and outer in the join execution, and associativity \[((R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3))\], which determines the order in which joins will be executed. The alternative join trees that are generated by commutativity and associativity are very large, up to \(O(N!)\) for \(N\) relations.

Thus, optimizers generally restrict the allowed sequences of join operations to limit the search space and therefore improve optimization time. For example, some optimizers only allow linear sequences of join operations. Of course, the sequence of joins in an operator tree does not need to be linear. For example, a query joining relations \(R_1, R_2, R_3, \) and \(R_4\) can be algebraically represented and evaluated as \((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)\). Such query trees are called bushy and typically require materializing intermediate relations. While bushy trees may result in a cheaper query plan, they considerably increase the cost of enumerating the search space. In general, most optimizers focus on linear join sequences and at most some restricted subsets of bushy join trees.

Another common restriction is to defer Cartesian products until after all joins are processed. This restriction speeds up optimization but may also miss opportunities in some decision-support queries where a single big table joins with multiple small tables. Performing some Cartesian products among these small tables before joining them with the big table can result in a significant reduction in cost.

11.2.1.2 Group-by and Join Clauses

In a conceptual evaluation of a SQL query block, the processing of the join precedes that of the group-by. Similarly, a group-by clause in a sub-query in the FROM clause is conceptually evaluated before the joins in the outer query block. Some algebraic transformations enable commuting group-by and join

* In a linear sequence of joins, at least one operand in each join is a base relation.
operators [3,4]. The evaluation of a group-by operator can potentially result in a significant reduction in the number of tuples, since only one tuple is generated for every partition of the relation induced by the group-by. At the same time, joins might eliminate large portions of an input relation, resulting in fewer tuples to subsequently group. Therefore, we can significantly reduce the cost of a query by reordering join and group-by clauses.

As an example of such transformations, a group-by clause can be pulled up above a join, as long as (i) we add a key of the other join relational input to the set of grouping columns and (ii) the join predicate is not defined over an aggregated column, or else the resulting join is not well-formed. Figure 11.3 illustrates this transformation, assuming that $b_2$ includes $b_1$ so that the original join is well-formed. Conversely, a group-by clause can be pushed below a join whenever certain conditions are satisfied.

11.2.1.3 Outer Joins

One-sided outer joins are asymmetric operators in SQL that preserve all of the tuples of one relation. For instance, a left outer join between tables $R$ and $S$, denoted $R \rightarrow S$, is similar to the join $R \bowtie S$, but additionally returns all rows in $R$ that do not match any row in $S$ filling with NULL values all columns in $S$. Symmetric outer joins preserve both the input relations. Unlike regular joins, a sequence of outer joins and joins do not freely commute. However, the following algebraic equivalence allows to pull outer joins above a block of joins:

$$R \bowtie (S \rightarrow T) \equiv (R \bowtie S) \rightarrow T$$

After outer joins have been delayed in this manner, joins may be freely reordered. There is no guarantee as to which alternative is better in all cases, and thus the use of this transformation needs to be cost-based [5].

11.2.2 Access Path Selection

Earlier query optimizers allowed at most one index for accessing tuples of each table in a given query. For queries with complex single-table predicates, more advanced strategies can manipulate multiple indexes simultaneously to efficiently obtain the qualifying tuples. Consider, as an example, a SQL query that returns information on engineers that earn less than $50,000

```sql
SELECT *
FROM Emp
WHERE Title = 'Engineer' AND Salary < 50000
```

and suppose that we have single-column indexes $E_1 = Emp(Title)$ and $E_2 = Emp(Salary)$. Traditionally, the alternative plans that the optimizer considers for the query are either (i) a clustered index scan filtering each tuple by both predicates or (ii) an index seek using either $E_1$ or $E_2$, followed by lookups to the clustered index to obtain the remaining columns, ended by a filter on the predicate that was not evaluated by the index. By using both indexes simultaneously, we can sometimes obtain
better execution plans. In this case, we could perform what is called an *index intersection* by (i) using index $E_1$ to obtain all record identifiers (RIDs) of tuples satisfying $Title = 'Engineer'$, (ii) using index $E_2$ to obtain all RIDs of tuples that satisfy $Salary < 50000$, (iii) intersecting both lists of RIDs, and (iv) fetching from the clustered index the resulting records, which satisfy both predicates.

This alternative can greatly reduce execution times because the individual indexes $E_1$ and $E_2$ are in general smaller than the clustered index, and the intersected list of RIDs can be much smaller than either of the original lists, resulting in much fewer RID lookups.

In general, we can intersect several indexes to answer complex single-table predicates, although at some point the benefit from additional index intersections is outweighed by its cost [6]. Additionally, there are analogous strategies to handle disjunctive predicates using RID unions and negation predicates using RID differences or two seek operations (e.g., employees that are not engineers). Considering these alternatives certainly increases the search space considerably, so some optimizers use heuristics similar to those used for join reordering to limit the number of such index-based alternative plans.

### 11.2.3 Other Algebraic Transformations

There are several additional transformations that exploit special cases or common patterns, some of which we briefly discuss in the following text:

*Query decorrelation*: The traditional strategy to process nested queries evaluates the inner sub-query for each tuple of the outer sub-query, in what sometimes is called *tuple iteration semantics*. Much work in the literature has identified techniques to decorrelate or *unnest* such queries by “flattening” them into a single block that can be executed more efficiently [7,8]. The complexity of the unnesting problem depends on the structure of the query. The simplest scenarios can be directly unnested using semijoins. The problem is more complex when aggregates are present in the nested sub-query, since unnesting requires pulling up the aggregation without violating the semantics of the nested query, which can be especially tricky in presence of duplicates and NULL values.

*Partial preaggregation*: In some scenarios, rather than fully pushing a group-by below a join, we can unfold a group-by clause into a local, more fine-grained group-by clause, which is pushed below the join, and a global group-by clause, which performs the final aggregation [9]. Such staged computation may still be useful in reducing the join costs because of the data reduction effect of the local aggregate, but requires the aggregate function to satisfy some additional algebraic properties.

*Materialized views*: Materialized views are results of queries that are stored and used by the optimizer transparently to answer other queries. Given a set of materialized views and a query, the problem is to optimize the query while leveraging available materialized views as much as possible. This problem introduces the challenge of identifying potential reformulations of the query so that it uses one or more of the materialized views [10,11].

*Common subexpressions*: When intermediate results are defined multiple times in a query, there are opportunities for avoiding multiple identical computations. Several specialized techniques leverage this fact to produce efficient plans that spool and reuse common subexpressions [12].

### 11.2.4 Implementation Choices

In addition to all the algebraic equivalences discussed so far, the search space also contains alternative implementations for each operator. For instance, joins can be implemented by different algorithms (e.g., hash-based, merge-based, and nested-loop-based), aggregates can be hash- or stream-based, supporting data structures can be built on the fly, and other implementation characteristics of this sort. The optimizer needs to consider different implementations for the same operator because depending on data and environment characteristics, any such alternative can be the most efficient one.
11.3 Cost Model

As we discussed so far, there are many logically equivalent algebraic expressions associated to a given query and multiple ways to implement each of these expressions. Even if we ignore the computational complexity of enumerating the space of possibilities, there remains the question of deciding which of the operator trees consumes the least resources. Resources may be CPU time, I/O cost, memory, or a combination of these. Therefore, given a partial or full operator tree, being able to accurately and efficiently evaluate its cost is of fundamental importance. Cost estimation must be accurate, since optimization is only as good as its cost estimates, and efficient, since it is in the inner loop of query optimization. The basic framework for estimating costs is based on the following recursive approach:

1. Collect statistical summaries of stored data.
2. Given an operator in the execution plan and statistical summaries for each of its subplans, determine (i) statistical summaries of the output and (ii) the estimated cost of executing the operator, which involves formulas that account the corresponding actions in the operator.

The procedure can be applied to an arbitrary tree to derive the costs of each operator. The estimated cost of a plan is then obtained by combining the costs of each of its operators. The resources needed to execute a query plan, and therefore its cost, is a function of the sizes of the intermediate query results. For that reason, cost estimation heavily depends on cardinality estimates of subplans generated during optimization. The following example illustrates how sizes of intermediate results may significantly change the chosen plan. Consider the following query template, where @C is a numeric parameter:

![Query Template](image)

Figure 11.4 shows optimal execution plans when @C is 20, 200, and 2000. Although the three query instances are syntactically very similar, the resulting plans are rather different. In Figure 11.4a, the number of tuples in R satisfying R.a < 20 is very small. Consequently, using a non-clustered index over R.a, the resulting plan first retrieves the RIDs of all tuples in R that satisfy R.a < 20. Then, using lookups against table R, it lookups the actual tuples that correspond to those RIDs. Finally, it performs a nested-loop join between the subset of tuples of R calculated before, and table S, which is sequentially scanned. For the case @C = 2000 in Figure 11.4c, the number of tuples of R satisfying R.a < 2000 is rather large. The resulting plan therefore scans both tables sequentially, discards on the fly tuples from R that do not satisfy the condition R.a < 2000, and then performs a hash join to obtain the result. (Note that in this scenario, the lookups of the previous plan would have been too numerous and therefore too expensive.) Finally, Figure 11.4b shows the execution plan when the number of tuples of R satisfying the predicate is neither too small nor too large. In this case, table S is scanned in increasing order of S.y using a

![Figure 11.4](image)
clustered index, and table $R$ is first scanned sequentially discarding invalid tuples as before and then sorted by $R.x$. Finally, a merge join is performed on the two intermediate results.

### 11.3.1 Statistical Summaries of Data

For every table, statistical information usually includes the number of tuples, the average size of each tuple, the number of physical pages used by the table, and the number of distinct tuples in the table, among others. Statistical information on table columns, if available, is helpful for estimating the cardinality of range or join predicates. A large body of work studies the representation of statistics on a given column or combination of columns. Such alternatives include histograms [13], wavelets [14], sketches [15], sampling [16,17], curve-fitting/parametric information [18], and many others [13]. In most commercial systems, information on the data distribution on a column is provided by histograms, since they can be built and maintained efficiently.

Histograms divide the values of a column or set of columns into a number of buckets, and associate with each bucket some aggregated information. The number of buckets not only influences the accuracy of the histogram, but also affects memory usage, since relevant histograms are loaded into memory during optimization. Both the particular procedure used to select bucket boundaries and the aggregated information associated with each bucket lead to different families of histograms [13]. A popular example is equi-depth histograms, which divide the domain of a column in a table into ranges so that each range has approximately the same number of rows. Other histogram variants, illustrated in Figure 11.5, are equi-width (where the data domain is divided into ranges of equal size), max-diff (where bucket boundaries are placed between pairs of consecutive values that have the largest frequency gap), and end-biased (where separate counts are maintained for a small number of very frequent values, and the remaining values are modeled using an equi-depth histogram). Although single-dimensional histograms are widely used, multidimensional histograms are relatively rare in commercial DBMSs.

### 11.3.2 Cardinality Estimation

Histograms and other statistical structures provide a compressed representation of the underlying column distribution. When manipulating such compact data structures, we generally make a number of simplifying assumptions. For instance, we assume that the tuples in a histogram bucket are distributed uniformly in the bucket domain. In absence of multidimensional statistics, we also assume that predicates in different columns are independent. Using these assumptions, there are simple procedures

![Histograms approximate value distributions of table columns.](11-5)
to estimate the number of output tuples for each operator. For instance, to estimate the cardinality of single-table range predicates we use interpolation over the histogram buckets. Join estimation requires that we combine information of two histograms, and involves aligning buckets under certain simplifying assumptions on data distributions. Group-by queries leverage the number of distinct values, stored either globally at the column level, or inside each histogram bucket. Of course, these assumptions only approximate the true cardinality values, and the aforementioned techniques are prone to very large errors in the worst case [19]. To mitigate these problems, some approaches directly store statistics on intermediate expressions and are able to estimate cardinality values without simplifying assumptions whenever the input subexpressions match such intermediate statistics [20].

11.4 Enumeration Strategy

An enumeration algorithm must pick an efficient execution plan for an input query by effectively exploring the search space. An important consideration while designing an enumeration algorithm is to allow it to gracefully adapt to changes in the search space or cost model. Optimization architectures built with this paradigm are called extensible optimizers. Building an extensible optimizer does not just involve designing a better enumeration algorithm but also providing an infrastructure for evolution of the optimizer design. In this section, we focus on three representative examples of such extensible optimizers. Rather than providing a detailed description of all the features in these enumeration architectures, we give instead a high-level overview of the frameworks and focus on the aspects that are relevant to our discussion.

11.4.1 Dynamic Programming

System-R was a seminal project that influenced much of the subsequent work in query optimization [21]. The join enumeration algorithm in System-R uses dynamic programming, and it assumes that the cost model satisfies the principle of optimality. Specifically, we assume that in order to obtain an optimal plan for a query consisting of \( k \) joins, it suffices to consider only the optimal plans for query subexpressions that consist of fewer joins and extend those plans with additional joins. In other words, the optimal plan is obtained by combining optimal subplans.

The dynamic programming algorithm, illustrated in Figure 11.6, views the input as a set of relations \( \{R_1, \ldots, R_n\} \) to be joined and works in a bottom-up manner. We assume there is an associative array \( \text{bestPlan} \), which, or each subset of tables \( R \), returns the best calculated plan so far for \( R \). First, we generate plans for every table in the query (lines 1 and 2). Such access path selection is encapsulated in function \( \text{bestAccessPath} \) and consists of scans, seeks, or optionally, more complex plans using index intersections and unions. Then, the dynamic programming algorithm performs \( n - 1 \) iterations in lines 3–8. At the end of the \( i \)-th iteration, we produce the optimal plans for all subexpressions that join \( i \) tables. Line 4 considers each subset \( R \) of \( i \) tables, and line 5 attempts to further partition each such subset into \( R' \) and \( R - R' \). For each pair of table subsets, line 6 obtains the best plan that joins tables in \( R' \)
Query Optimization

and tables in $\mathcal{R} - \mathcal{R}'$. For that purpose, the procedure $\text{bestJoin}$ tries different join implementations (e.g., hash-, merge-, and index-based joins). Note that the input to $\text{bestJoin}$ is $\text{bestPlan}(\mathcal{R}')$ and $\text{bestPlan}(\mathcal{R} - \mathcal{R}')$, therefore leveraging the principle of optimality. Lines 7 and 8 update $\text{bestPlan}$ for the current subset $\mathcal{R}$ based on the cost model. Suboptimal plans are discarded and never considered again. Finally, line 9 returns the best plan for the whole set of tables, which corresponds to the optimal join reordering for the input query. We now supplement the description of the dynamic programming algorithm for join reordering with a couple of important refinements.

First, consider a query that represents the join among $\{R_1, R_2, R_3\}$ with the predicates $R_1.a = R_2.a$ and $R_2.a = R_3.a$. Let us also assume that the cost of a hash-based join plan for $R_1 \bowtie R_2$ is smaller than that of a merge-based join alternative. In such a case, $\text{bestPlan}([R_1, R_2])$ would only keep the hash-based join alternative discarding the suboptimal merge-based join. However, note that if merge-based join alternative is used in $R_1 \bowtie R_2$, the result of the join is sorted on columns $R_1.a$ and $R_2.a$. The sorted order may significantly reduce the cost of the subsequent join with $R_3$. Thus, pruning the merge join alternative for $R_1 \bowtie R_2$ can result in suboptimality of the global plan. The problem arises because the result of the merge join between $R_1$ and $R_2$ has an ordering of tuples in the output stream that is useful in a subsequent join. However, the hash-based join alternative, while cheaper locally, does not have such ordering. To address this problem, the dynamic programming algorithm is extended to keep in $\text{bestPlan}$ an optimal plan for every choice of an interesting sort order. Two plans are then compared if they represent the same expression and have the same interesting order.

Second, note that the algorithm in Figure 11.6 enumerates all bushy plans, including those that perform early Cartesian products. The algorithm can easily be adapted to enumerate through different search spaces by suitably modifying line 5. To consider only (left) linear join trees, line 5 is changed to:

5 $\text{foreach } \mathcal{R}' \subset \mathcal{R}$ such that $|\mathcal{R}'| = 1$

The strategy discussed in this section was extended several times, culminating in a technique in [22] that efficiently enumerates all bushy trees with no cross products and additionally considers outerjoins.

11.4.2 Rule-Based Systems

In the context of extensible DBMSs, several rule-based strategies have been proposed. Rules are defined on how plans can be constructed or modified, and the enumeration strategy follows the rules to explore the search space. The most representative efforts are those of Starburst [23] and Volcano/Cascades [24,25]. In this section, we briefly describe the Cascades framework, used as the foundation for both academic and industrial query optimizers. This framework works by manipulating operators, which are the building blocks of operator trees and are used to describe both the input declarative queries and the output execution plans. Consider the simple SQL query in the following text:

```
SELECT *
FROM R, S, T
WHERE R.x = S.x AND S.y = T.y
```

Figure 11.7a shows a tree of logical operators that specify, in an almost one-to-one correspondence, the relational algebra representation of the query in the preceding text. In turn, Figure 11.7c shows a tree of physical operators that corresponds to an execution plan for the aforementioned query. The goal of the optimization process is to transform the original logical operator tree into an efficient physical operator tree. For that purpose, Cascades-based optimizers rely on two components: the $\text{MEMO}$ data structure, which keeps track of the explored search space, and optimization tasks, which guide the enumeration strategy.
The **MEMO** data structure provides a compact representation of the search space of plans by relying on memoization, which is a variant of dynamic programming. A **MEMO** consists of two key data structures, called **groups** and **groupExpressions**. A **group** represents all equivalent operator trees producing the same output. To reduce memory requirements, a **group** does not explicitly enumerate all its operator trees. Instead, it represents all the operator trees by using **groupExpressions**. A **groupExpression** is an operator having other **groups** as children (rather than other operators). As an example, consider Figure 11.7b, which shows a **MEMO** for the simple query in the previous section, where logical operators are shaded and physical operators use white background. In the figure, **group 1** represents all equivalent expressions that return the contents of table **R**. Some operators in **group 1** are logical (e.g., **Get R**) and some are physical (e.g., **Table Scan** and **Sorted Index Scan**). Likewise, **group 3** contains all equivalent expressions for **R ⊙ S**. Note that **groupExpression 3.1**, (i.e., **Join(1,2)**), represents all operator trees whose root is **Join**, first child is in **group 1**, and second child is in **group 2**. In this way, a **MEMO** compactly represents a very large number of operator trees.

The enumeration algorithm in Cascades is divided into several **tasks**. Initially, the optimizer schedules the optimization of the **group** corresponding to the root of the original query tree (**group 5** in the figure). This task in turn triggers the optimization of smaller and smaller operator subtrees and eventually returns the most efficient execution plan for the input query. At the heart of the optimization tasks there are transformation rules. A transformation rule is a pair of (i) an antecedent expression to match in the **MEMO** and (ii) a consequent expression to generate and introduce back in the **MEMO**. On one side, **exploration** rules transform logical operator trees into equivalent logical operator trees, and can range from simple rules like join reordering to more complex ones like pushing aggregates below joins. On the other side, **implementation** rules transform logical operator trees into hybrid logical/physical trees by introducing physical operators into the **MEMO**. Implementation rules range from simple ones like transforming a logical join into a physical hash join to very complex ones that generate index strategies.

Applying a rule can be broken down into four steps. First, all bindings for the rule antecedent are identified and iterated over (note there can be different ways of matching the antecedent of the rule with operator trees in the current **group**). Second, the rule is applied to each binding generating one or more new expressions. Third, the resulting expressions are integrated back into the **MEMO** possibly creating new **groups** (e.g., applying join associativity to expression 5.1 in the figure results in **groupExpression 5.2**, which points to a newly created **group 7**). Finally, each new **groupExpression** triggers new tasks that continue exploring logical equivalences or implement physical plans.
11.4.3 Heuristic Enumeration Strategies

The enumeration strategies discussed so far exhaustively explore the plan search space. Their memory requirements and running time thus grow exponentially with query size in the worst case. For that reason, many modern systems have a limit on the size of queries that can be exhaustively optimized (usually around 15 joins). Beyond these limits, query optimizers apply heuristics to enumerate a fraction of the search space or use faster specialized strategies for special cases. As an example of the latter approach, it is known that for chain queries, the classical dynamic programming algorithm works in polynomial time [26]. Also, if the join graph is acyclic and the cost model satisfies the adjacent sequence interchange (ASI) property [27], the IKKBZ technique in [28] returns the optimal plan in $O(N^2)$ where $N$ is the number of tables.

To effectively address more complex scenarios, heuristic enumeration strategies are proposed, which explore only a fraction of the search space. One line of work adapts randomized techniques and combinatorial heuristics to address this problem, and considers the space of plans as points in a high-dimensional space, that can be traversed via transformations (e.g., join commutativity and associativity). Such strategies include iterative improvement, simulated annealing, and genetic algorithms [29] and can be seen as heuristic variations of transformation-based exhaustive enumeration algorithms. Another line of work implements heuristic variations of dynamic programming. Some notable approaches include iteratively performing dynamic programming on a subset of tables and replacing intermediate results with a new virtual table [30], simplifying the initial join graph by disallowing non-promising join edges and exhaustively searching the resulting simpler problem [31], and greedily building execution trees one operator at a time [32,33].

11.5 Advanced Optimization Topics

In this section, we discuss some advanced types of optimization that researchers have proposed over the past few years, and for which fundamental questions remain open. The descriptions are based on examples only; further details may be found in the references provided. Furthermore, there are several research challenges that we do not discuss at all due to space limitations, although much interesting work has been and remains to be done on them, such as query optimizer generators; query optimizer validation; and domain-specific query optimization, such as object-oriented, heterogeneous, recursive, XML-based, stream-based, and probabilistic-based, among others.

11.5.1 Parallel/Distributed Optimization

A parallel DBMS is typically an individual system controlling multiple processors that are in the same location. When multiple processors are available, each operator in the query tree can be parallelized by partitioning the input among available processors and later combining results using intraoperator parallelism. Additionally, there are multiple choices to place operators into groups that should be executed simultaneously by the available processors, called interoperator parallelism and further divided into pipelining and independent parallelism. Parallelism can reduce the overall time to process a query, but is associated with much larger search spaces over traditional sequential environments. For that reason, most systems and research prototypes adopt heuristics to avoid dealing with a very large search space. One popular approach first identifies the optimal sequential plan using conventional techniques and then identifies the optimal parallelization/scheduling of that plan [34]. Other approaches only consider schedules that process memory-resident right-deep segments of possibly bushy query plans one at a time with no independent interoperator parallelism [35,36]. More recent strategies integrate parallel plan optimization mode deeply into the query optimizer, instead of being done during postoptimization [37]. With recent growing interest in large-scale computation, parallel optimization remains an active research area.
Distributed Databases are composed of semiautonomous processing sites that are connected via a network that could be spread over a large geographic area. Many prototypes of distributed DBMSs have been implemented [38,39], and several commercial systems offer distributed versions of their products as well. The main differences between centralized and distributed query optimization is in the search space, which additionally contains various processing strategies and opportunities for transmitting data among sites. The most well-known opportunity is to preprocess joins use semijoins in order to only transmit tuples that would certainly contribute to join results [38]. 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 [39].

11.5.2 Parametric Query Optimization

Many commercial applications rely on precompiled parameterized queries to interact with a database. The DBMS optimizes and caches a parameterized query the first time it is submitted and then reuses the cached execution plan when the query is resubmitted with different parameter values. This approach is especially useful for relatively simple queries, as each optimization call may be a significant fraction of the total execution time. Unfortunately, executing a cached query with a set of parameters different from those used at compilation may be suboptimal. Parametric query optimization addresses this problem by exhaustively determining the optimal plans in each point of the parameter space at compile time. A parametric optimizer determines a set of plans such that, for each point in the parameter space, there is at least one plan in the set that is optimal. At runtime, these techniques use the actual parameter values and simply pick the plan that was found optimal for them with little or no overhead. Parametric optimization proposals often assume that the cost formulas of physical plans are linear or piecewise linear with respect to the cost parameters or that the regions of optimality are connected and convex. Various approaches to parametric query optimization include dynamic plans [40], randomized algorithms [41], and non-intrusive procedures on top of traditional query optimizers [42]. Recent work provides interesting and actionable diagrams of the plan choices of the optimizer over a space of input parameters [43]. Although much work has been done in the area of parametric optimization, there is still significant room for improvement, especially in the context of industry-strength database systems.

11.5.3 Adaptive Query Processing

Traditional approaches to query processing can be crisply divided into query optimization (which produces an execution plan) and execution proper (which evaluates the plan and obtains results). The quality of the resulting execution plan depends, to a large extent, on (i) the cost model and underlying cardinality estimations being accurate and (ii) the absence of variability in the environment variables used during optimization (e.g., available memory). In real scenarios, however, both these assumptions are not valid, and thus execution plans can be highly suboptimal. This problem motivates intra-query adaptivity, which essentially attempts to somehow interleave query optimization and execution, allowing the system to correct bad choices as soon as possible [44]. A simple approach uses execution feedback to progressively improve the statistical information used during optimization [18,45]. More dynamic approaches to address this problem insert checkpoints in the execution plan that monitor actual cardinality values during execution. If the actual intermediate results are very different from the estimated ones, the system suspends the execution of the query, reoptimizes it using the new information, and chooses whether to continue with the original plan or switch to a new one [46,47]. An extreme approach continuously reorders operators in a query plan as it runs, conceptually considering different plans for each routed tuple [48]. Adaptive query processing is a topic that received much attention lately, but has not yet fully found its way into commercial systems.
11.5.4 Multiquery Optimization

So far, this chapter focused on optimizing individual queries. Quite often, however, multiple queries become available for optimization at roughly the same time. Instead of optimizing each query separately, we may obtain a \textit{global} plan that, although suboptimal for each individual query, is optimal for the execution of all of them as a group [49]. Consider the following queries:

\begin{align*}
Q_1 &= \text{SELECT Emp.Name} \\
&\text{FROM Emp, Dept} \\
&\text{WHERE ON Emp.id = Dept.id} \\
&\text{AND Dept.Budget > 1M}
\end{align*}

\begin{align*}
Q_2 &= \text{SELECT Dept.floor, Emp.Name} \\
&\text{FROM Emp, Dept} \\
&\text{WHERE ON Emp.id = Dept.id} \\
&\text{AND Emp.Title = 'Programmer'}
\end{align*}

Depending on the sizes of the \texttt{Emp} and \texttt{Dept} relations and the selectivities of the predicates, it may well be that computing the entire join once and then applying the two predicates are more efficient than doing the join twice, each time taking into account the corresponding predicate.

11.5.5 Semantic Optimization

Semantic query optimization is a form of optimization that uses integrity constraints defined in the database to rewrite a given query into semantically equivalent ones [50,51]. These can then be optimized as regular queries, and the most efficient plan among all can be used to answer the original query. Examples of semantic optimization include introduction of inferred predicates, removal of joins under foreign key constraints when only columns from the primary table are used, and inference about distinct values for grouping queries. As a simple example, consider an integrity constraint on table \texttt{Emp} of the form \texttt{Title = "Programmer" \Rightarrow salary > 100K} and the following query:

\begin{align*}
\text{SELECT name, floor} \\
&\text{from Emp JOIN Dept ON Emp.id = Dept.id} \\
&\text{WHERE Title = "Programmer"}
\end{align*}

Using this constraint, the query can be rewritten into an equivalent one that includes an extra predicate \texttt{salary > 100 K}, which results in an efficient plan if the only available index is on \texttt{Emp.salary}.

\begin{align*}
\text{SELECT name, floor} \\
&\text{from Emp JOIN Dept ON Emp.id = Dept.id} \\
&\text{WHERE Title = "Programmer" AND salary > 100K}
\end{align*}

Semantic optimization is a relatively old topic that periodically finds renewed interest in the research community.

11.5.6 Physical Database Design

DBMSs provide physical data independence, so that queries return the same result independently of the set of available indexes. Of course, the performance of evaluating queries will significantly vary depending on index availability. Together with the capabilities of the execution engine and query optimizer, the physical design of a database determines how efficiently a query is ultimately executed. For this reason, there has been a great amount of interest in automatically deciding, for a given query workload, what would be a reasonably good index configuration to materialize and maximize query processing performance [52]. Physical database design is a complex search problem over the space of available indexes, and current solutions iterate over different index configurations evaluating the benefit of such
index subsets. These solutions require evaluating the expected cost of a query under a given candidate index configuration in the search space. To do so in a scalable manner, optimizers are typically extended with a new interface that simulates a hypothetical index configuration in the DBMS and optimize queries without actually materializing new indexes, a process called what-if optimization. What-if optimization is a cornerstone of automated physical database design, which enabled virtually all commercial tuners. It is also one of the first scenarios in which the query optimizer itself is made into the inner loop of an even larger application. Relying on a query optimizer to recommend physical design changes is a novel research direction that found significant interest in the industry, and is currently being actively refined and generalized in the research community.

## 11.6 Conclusion

To a large extent, the success of a DBMS lies in the quality, functionality, and sophistication of its query optimizer. In this chapter, we gave a bird’s eye view of query optimization, presented an abstraction of the architecture of a query optimizer, and focused on the techniques currently used by most commercial systems. In addition, we provided a glimpse of advanced issues in query optimization, whose solutions are still partial and have not completely found their way into commercial systems. Although query optimization has existed as a field for more than 30 years, it is very surprising how fresh it remains in terms of being a source of research problems, as illustrated in the previous section. At the same time, in virtually every component of the traditional optimization architecture of Figure 11.2, there are crucial questions for which we do not have complete answers. When is it worthwhile to consider bushy trees instead of just linear trees? How can buffering and other runtime variables be effectively modeled in the cost formulas? Which search strategy can be used for complex queries with reliability and confidence? We believe that the future in query optimization will be as active as the past and will bring many advances to optimization technology, changing many of the approaches currently used in practice. Despite its age, query optimization remains an exciting field.

## Acknowledgment

I thank Ravi Ramamurthy for many useful comments that improved the presentation and content of this chapter. Portions of this chapter also appeared in earlier papers and book chapters [52,53].

## References


