Foundations and Trends® in Databases

Extensible Query Optimizers in Practice

Suggested Citation: Bailu Ding, Vivek Narasayya and Surajit Chaudhuri (2024), "Extensible Query Optimizers in Practice", Foundations and Trends® in Databases: Vol. 14, No. 3-4, pp 186–402. DOI: 10.1561/1900000077.

Bailu Ding
Microsoft Corporation
badin@microsoft.com

Vivek Narasayya
Microsoft Corporation
viveknar@microsoft.com

Surajit Chaudhuri
Microsoft Corporation
surajitc@microsoft.com

now
the essence of knowledge
Boston — Delft

Contents

Extensible Query Optimizers in Practice

Bailu Ding, Vivek Narasayya and Surajit Chaudhuri

Microsoft Corporation, USA; badin@microsoft.com, viveknar@microsoft.com, surajitc@microsoft.com

ABSTRACT

The performance of a query crucially depends on the ability of the query optimizer to choose a good execution plan from a large space of alternatives. With the discovery of algebraic transformation rules and the emergence of new application-specific contexts, extensibility has become a key requirement for query optimizers. This monograph describes extensible query optimizers in detail, focusing on the Volcano/Cascades framework used by several database systems including Microsoft SQL Server. We explain the need for extensible query optimizer architectures and how the optimizer navigates the search space efficiently. We then discuss several important transformations that are commonly used in practice. We describe cost estimation, an essential component that the optimizer relies upon to quantitatively compare alternative plans in the search space. We discuss how database systems manage plans over their lifetime as data and workloads change. We conclude with a few open challenges.

Bailu Ding, Vivek Narasayya and Surajit Chaudhuri (2024), "Extensible Query Optimizers in Practice", Foundations and Trends® in Databases: Vol. 14, No. 3-4, pp 186–402. DOI: 10.1561/1900000077.

1

Introduction

SQL [134] is a high-level declarative language for querying relational data. It is the de-facto standard query language for relational data and is supported by all major relational database management systems (RDBMSs) and increasingly also by the Big Data Systems. SQL allows declarative specification of queries over relational data involving selections, joins, group-by, aggregation, and nested sub-queries, which are important for a wide variety of decision support queries including business intelligence scenarios in enterprises [31].

Consider the example Query 1 shown below.

Query 1

```
SELECT *
FROM R, S, T
WHERE R.a = S.b AND S.c = T.d AND T.e = 10
```

Figure 1.1 shows the major steps in the workflow of processing a SQL query in a RDBMS. The three stages of query processing are explained below.

Parsing and validation The parsing and validation step converts the input SQL query into an internal representation. This step ensures that

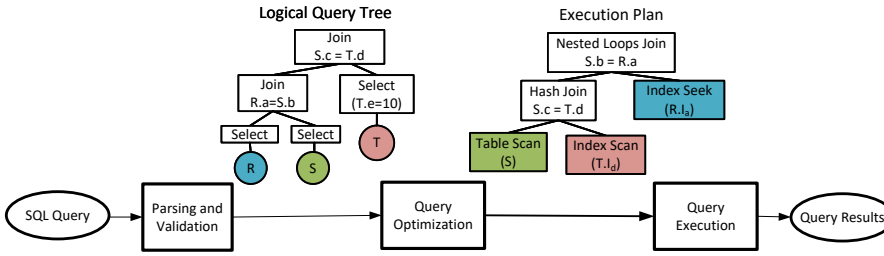


Figure 1.1: Workflow of query processing

the query adheres to the SQL syntax and only contains references to existing database objects, e.g., tables and columns. The output of this step is a logical query tree, an algebraic representation of the query in the form of a tree of logical relational operators (e.g,. Select, Join). For example, Figure 1.1 shows the output logical query tree of Query 1 after the parsing and validation step.

Query optimization The *query optimizer* takes a logical query tree as the input, and is responsible for generating an *efficient* execution plan that is either interpreted or compiled by the query execution engine. An *execution plan* (also referred to as *plan*) is a tree of physical operators, with edges representing the data flow between the operators. For example, Figure 1.1 shows the output execution plan of Query 1 after the query optimization step. For a given query, the number of different execution plans that may be used to answer the query may grow exponentially with the number of tables referenced in the query, and different execution plans can vary widely in terms of efficiency. Therefore, the performance of a query crucially depends on the ability of the optimizer to choose a good execution plan from a large space of alternatives. An overview of query optimization in RDBMSs is available in [28].

Query execution The query execution engine takes the plan from the query optimizer and executes the plan to produce the query results. The query execution engine implements a set of *physical operators*, which are building blocks for executing SQL query plans. A physical operator

takes one or more sets of data records as its input, referred to as *rows*, and outputs a set of rows. Examples of physical operators include Table Scan, Index Scan, Index Seek (see Appendix), Hash Join, Nested Loops Join, Merge Join, and Sort. For descriptions of algorithms used for various physical operators, we refer the reader to [77].

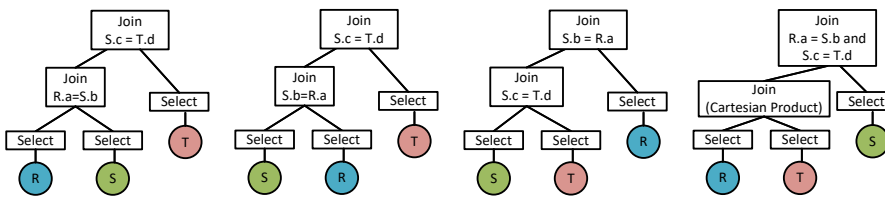
Query execution in a majority of relational database systems follows the *iterator* model, where each physical operator implements the *Open*, *GetNext*, and *Close* methods. Every iterator contains record of its state with information such as the size and the location of the hash table. In *Open*, the operator initializes its state and prepares for processing. When *GetNext* is called, the operator produces the next output row or indicates that there are no more rows, i.e., end of processing. We observe that to produce an output row a non-leaf operator in the plan needs to call *GetNext* on its child operator(s). For example, consider the execution plan shown in Figure 1.1. The Nested Loops Join operator calls *GetNext* on the Hash Join operator, which in turn calls *GetNext* on Table Scan(S) operator. When an operator completes producing its output rows (i.e., indicates that there are no more rows), the parent calls *Close* on it to allow the operator to clean up its state. The above approach of specifying operators through the iterator model makes it convenient to add new operators to the execution engine. Since each operator is an iterator from which rows are 'pulled', this model of execution is also referred to as a *pull model*. We refer the reader to [79] for a complete description of the pull model of query execution.

The iterator model as described above incurs high overhead of function invocations with each *GetNext* call processing a single row at a time, resulting in poor performance on modern CPUs. *Vectorization* enables batching so that a single *GetNext* call for a physical operator produces results for a batch of rows and leverages the SIMD instructions of modern CPUs [19]. Together with columnar representation [188], vectorization sharply increases the efficiency of query execution engines for decision support queries. In addition, *code generation* is a technique that generates efficient code from the query execution plan in a language such as C [152], which is then compiled and executed, or directly generates efficient machine code using a compiler framework such as LLVM [114]. The tradeoffs in vectorization and compilation are discussed in [107].

1.1 Key Challenges in Query Optimization

To choose an efficient plan among many alternative execution plans, a query optimizer must determine the *search space* of plans it will explore, compare the relative efficiency of the plans with *cost estimation*, and navigate the search space with an efficient *search algorithm* to find an execution plan that has very low (ideally lowest) cost of execution among its choices. We now briefly describe these facets of a query optimizer.

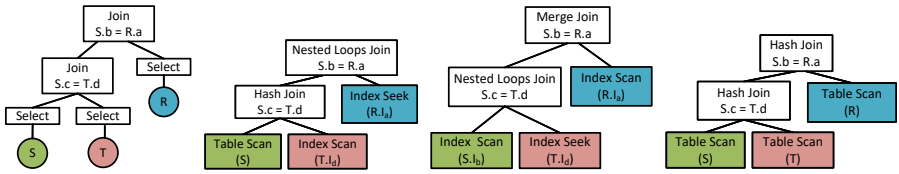
Search space The search space consists of alternative equivalent execution plans of the query, which can be large for complex queries. First, a given algebraic representation of a query can potentially be transformed into many other equivalent representations. These equivalences arise from properties of relational algebra, e.g., $Join(Join(R,S),T) \iff Join(Join(S,T),R)$ since the Join operator is commutative and associative [63]. Figure 1.2 shows four different but equivalent algebraic representations of the same query.



(a) Logical plan L_1 **(b)** Logical plan L_2 **(c)** Logical plan L_3 **(d)** Logical plan L_4

Figure 1.2: Semantically equivalent logical query trees

Second, for a given logical operator there are many different implementations of that logical operator. Hence, for a given logical query tree, there are potentially many different possible execution plans. For example, in Figure 1.3, for the logical query tree in Figure 1.3a, we show three out of many possible execution plans in Figure 1.3b-1.3d. Although the three plans have the same order in which joins are evaluated, they vary in the specific physical operators used to implement the logical operators. For example, the Select operator in Figure 1.3a can be implemented using Table Scan, Index Scan, or Index Seek; and the Join operator can be implemented using Nested Loops Join, Hash



(a) Logical plan L **(b)** Execution plan P_1 **(c)** Execution plan P_2 **(d)** Execution plan P_3

Figure 1.3: Different execution plans for a given logical query tree

Join, or Merge Join. The Nested Loops Join in Figure 1.3b may be the most efficient among the three when the *join size* (i.e., number of rows produced by the join) of the join between S and T is small and an index I_a is available on the join column $R.a$. The plan in Figure 1.3c with the Merge Join may be a good choice when an index I_b is available on $S.b$ and an index I_a is available on $R.a$, i.e., the indexes provide the sort order required by the Merge Join. In contrast, the plan in Figure 1.3d with the two Hash Join operators may be the plan of choice when the size of the join between S and T is large. Thus, unless the optimizer considers each of these plans in its search space and compares their resource usage and expected relative performance, it may not produce a good plan.

Cost estimation The efficiency of different execution plans for the same query, measured by their elapsed time or resources consumed (e.g., CPU, memory, I/O), can vary significantly, as the example in Figure 1.3 shows. The difference in elapsed time between a good and a poor execution plan for complex queries on large databases can be several orders of magnitudes. Therefore, to pick a good execution plan for a query from the space of execution plans as noted above, most query optimizers leverage a *cost model* that estimates the work done by query execution plans with sufficient fidelity so that relative comparisons of the execution plans are accurate. Specifically, a physical operator must estimate the work done by the algorithm used to implement that operator, and this estimation requires the sizes and other statistical characteristics of the input relation(s) to that operator as well as those of its output. Finally, even though the cost has at least three dimensions

(CPU, memory, I/O), the cost model combines these multi-dimensional costs in a single number for the convenience of comparing any two plans.

Search algorithm In principle, one could exhaustively enumerate every alternative execution plan in the search space and invoke cost estimation to determine the cost of each plan in order to find the plan with the lowest estimated cost. As some of the alternative execution plans can share common logical or physical operator trees, e.g., *Select(S)* in L_1-L_4 of Figure 1.2 or *Table Scan(S)* in P_1 and P_3 of Figure 1.3, the enumeration needs to be done carefully to avoid duplicate explorations. Even so, the exhaustive enumeration can still be too costly in practice. Thus, a good query optimizer will try to reduce the cost of enumeration without compromising significantly the quality of the chosen execution plan.

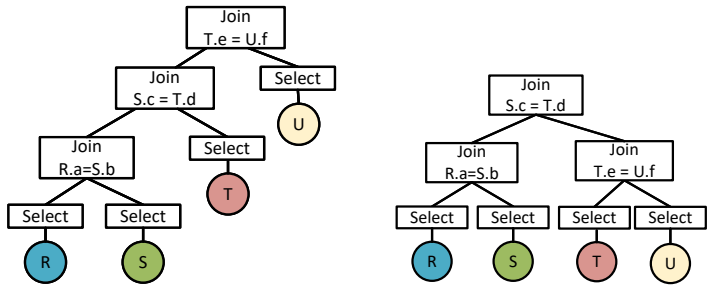
In summary, a good optimizer is one which: (a) considers a sufficiently large search space of promising plans, (b) models the cost of execution plans sufficiently accurately to distinguish between plans with significantly different costs, and (c) provides a search algorithm that efficiently finds a plan with low cost.

1.2 System R Query Optimizer

The System R project from IBM Research did pioneering work on query optimization [178]. We briefly review how the System R query optimizer addressed the key challenges mentioned in Section 1.1. The techniques developed in this project have had significant impact on all query optimizers that followed, including extensible query optimizers.

Search space The System R query optimizer's cost-based plan selection technique focused on the Select-Project-Join (SPJ) class of queries. The physical operators for implementing a Select operation included Table Scan and Index Scan. For Join, System R provided two physical operators, Nested Loops Join and Merge Join (which requires both inputs to be sorted on the respective join columns). In the example of Query 1, as Figure 1.2 and Figure 1.3 illustrate, there are several logical query

trees and execution plans for this SPJ queries. This arises because join is associative and commutative, and there are multiple options of physical operators for Scan and Join operations. The space of logical query trees explored by System R for SPJ queries included the space of *linear sequence* of binary Join operations, e.g., $Join(Join(Join(R, S), T), U)$. Figure 1.4a shows an example logical query tree of a linear sequence of Join operations whereas the logical query tree in Figure 1.4b, i.e., a bushy plan, is not in the search space of System R. The optimizer also offered techniques to improve the efficiency of nested queries based on program analysis but these optimizations were not cost-based.



(a) Linear plan **(b)** Bushy plan

Figure 1.4: Linear sequence of joins vs. bushy join

Cost model The cost model of System R used formulas to estimate the CPU and I/O costs for each operator in execution plans. Unlike today's optimizers, it did not incorporate the cost of memory. The System R optimizer maintained a set of statistics on base tables and indexes, e.g., number of rows (cardinality) and data pages in the table, number of pages in the index, number of distinct values in each column. System R provided a set of formulas to compute the selectivity of a single selection or join predicate. The selectivity of a WHERE clause containing a conjunction of selection predicates was computed by multiplying the selectivity of all predicates, i.e., assuming the predicates are independent. Thus, the cardinality of the output size of a join was estimated by taking the product of the cardinalities of the two input relations and multiplying it with the selectivity of all predicates. The cost model formulas, together with statistical information on base tables and indexes, enabled the

System R optimizer to perform estimation of CPU and I/O costs of execution plans.

Search algorithm The search algorithm of the System R optimizer used *dynamic programming* to find the "best" join order, and is based on the assumption that the cost model satisfies the *principle of optimality*. In other words, it assumes that, in the search space of linear sequence of joins, the optimal plan for a join of n relations can be found by extending the optimal plan of a sub-expression of $n-1$ joins with an additional join. For example, the optimal plan P_{RST} of joining relations R, S, and T can be found from joining R with P_{ST}, joining S with P_{RT}, and joining T with P_{RS}, where P_{ST}, P_{RT}, and P_{RS} are the optimal plans for joining S, T, joining R, T, and joining R, S respectively. In contrast to the naive approach that enumerates $O(n!)$ plans by enumerating all permutations of the join ordering, the dynamic programming approach enumerates $O(n2^{n-1})$ plans, and is therefore significantly faster, even though the time complexity is still exponential in the number of joins.

A second important aspect of System R's search algorithm was its consideration of *interesting orders*. Consider a query Q that joins three tables R, S, and T, with join predicates $R.a = S.a$ and $S.a = T.a$. Suppose the cost of joining R and S with Nested Loops Join using an Index Seek on S is smaller than the cost of using Merge Join. In this case, when considering plans for joining R, S, and T, the optimizer would prune out the plan where R and S are joined using Merge Join. However, if Merge Join is used to join R and S, then the result of the join is sorted on column a, which may significantly reduce the cost of the join with T if Merge Join is used. Therefore, pruning a plan that joins R and S with a Merge Join can result in a sub-optimal plan for the query. The fact that the output rows of an operator are ordered, i.e., the operator has an interesting order, may lower the cost of parent or ancestor operators in the plan. To accommodate this violation of the principle of optimality due to interesting orders while retaining the benefits of using dynamic programming, the search algorithm considered the interesting order for every expression it enumerates. For a join expression, plans were compared in cost if and only if they had the same interesting order, and an optimal plan was kept for each distinct interesting order.

1.3 Need for Extensible Query Optimizer Architecture

The important concepts introduced by System R, including the use of data statistics and a cost model to determine an execution plan, the dynamic programming based search for join ordering, and the need to consider interesting orders, have been adopted by virtually all widely used query optimizers. However, the framework could not be flexibly and efficiently extended to additional algebraic equivalences in relational algebra and new constructs in database systems in a cost-based manner, which can potentially miss out opportunities to find cheaper query plans. As relational databases and SQL became important for decision support queries, the transformations for these additional algebraic equivalences became valuable for generating an efficient execution plan. Examples of such transformations include pushing down a group-by below a join to reduce the cost of the join, optimization of outer joins that are not associative nor commutative, and decorrelation of nested queries. In addition, new constructs were introduced to database systems to improve query execution performance. For example, materialized views [43, 84], which precompute and store the results of a query subexpression, and thereby could dramatically reduce the cost of executing the query, became important for OLAP and other analytical workloads. Furthermore, the optimizer also needed to support new logical and physical operators that were introduced to efficiently execute SQL queries, e.g., Apply [69].

Fortunately, as the practical needs of a SQL query optimizer expanded, the research on extensible database systems that was ongoing at that time yielded architectural alternatives to extending the architecture of System R. Extensible database systems were envisioned as systems that can be used to customize a database system to the needs of an application. Specifically, Exodus [26] and later Volcano [79], which were designed to support user-specified operators for query execution, emerged in that context. Given the need to support custom operators, providing a framework for extensible query optimization became a necessity. Thus, extensibility of the optimizer was a design feature in Volcano from the very beginning as it was initially envisioned as an "experimental vehicle for multitude of purposes" [79]. They allowed

system designers to "plug-in" new *rules*, drawing inspiration from rules in expert systems (production systems), and thereby extend the capabilities of the optimizer. Later, the extensible optimizer frameworks of Volcano/Cascades [80, 82] and Starburst [123, 165] focused on SQL query optimization as a key application, which fulfilled a pressing need for a new architecture for SQL query optimization.

For most of this monograph, we will focus on extensible optimizers based on Volcano/Cascades. These extensible optimization frameworks center around the concept of rules. A *logical transformation rule* represents an equivalence in the SQL language (or its algebraic representation). For example, the equivalences implied by join commutativity and associativity noted earlier can be expressed using rules. Similarly, a rule may define the conditions under which pushing down a group-by operation below a join preserves equivalence. Applying logical transformation rules to a query tree results in an equivalent alternative query tree. An *implementation rule* defines the mapping from a logical operator (e.g., Join) to a physical operator (e.g., Hash Join). Implementation rules are needed to generate execution plans for the query. A judicious choice of a sequence of applications of rules can potentially transform the query tree into one that executes much faster. It should be noted that in this architecture, new operators, logical transformations, and implementation rules can be incorporated without having to modify the search algorithm of the optimizer each time. Last but not the least, it is important to note that transformations do not necessarily reduce cost, and therefore the search algorithm must choose among the alternatives in a cost-based manner.

SQL is a declarative query language. This allows the query optimizers to create efficient execution plans for SQL queries that leverage logical transformations that preserve semantic equivalence and also judiciously choose the most efficient implementation for the logical operators. The holy grail of query optimization is to produce the most efficient execution plan that preserves semantic equivalence but is independent of how the query is expressed syntactically by the users or the applications. The extensible query optimizers make this goal achievable by applying rules, chosen from a rich set of transformations, to the query tree successively in a judicious sequence driven by a cost-based search algorithm.

1.4 Outline

In this monograph, we focus on the technology of extensible query optimizers and use Microsoft SQL Server for illustration of the key concepts. In comparison to the overview article on query optimization by one of the authors [28], this monograph provides a detailed description of extensible optimizer frameworks as well as several additional transformation rules that are commonly used in practice. The extensibility framework and rules are explained in depth using pseudocode and examples.

The rest of the monograph is organized as follows:

Section 2: We review the extensible optimizer frameworks of Volcano [82] and its successor, the Cascades framework [80], that have been influential. We describe the search algorithms and key data structures needed in both frameworks, as well as additional techniques to improve the efficiency of query optimization. We illustrate how Microsoft SQL Server's query optimizer leverages the Cascades framework with a few examples. Finally, we describe how the optimizer handles parallel and distributed query processing.

Section 3: We present a brief review of other extensible query optimizers, including Starburst used in IBM DB2, Orca used in Greenplum DB, Calcite used in Apache Hive, and Catalyst used in Spark SQL. Although PostgreSQL's query optimizer does not possess the extensibility capabilities of frameworks such as Volcano and Cascades, given its popularity, we include a short overview of its query optimizer.

Section 4: An extensible optimizer draws its effectiveness from the rules it leverages. In this section, we review some of the key logical transformations and implementation rules relevant for access paths to base tables, inner and outer joins, group-by, aggregation, and decorrelation of nested queries. We touch upon a few selected "advanced" rules, e.g., for optimizing star and snowflake queries which are common in data warehouses, sideways information passing, user-defined functions (UDFs), and materialized views.

Section 5: An optimizer framework critically depends on the cost model and cardinality estimation. In this section, we provide an overview of cost modeling and cardinality estimation with a focus on industrial

practices. We discuss the statistical summaries used by the optimizer such as histograms and how they are used for complex queries. In addition, we discuss recent adoption of sampling and sketches in database systems. Finally, we illustrate these concepts and techniques using Microsoft SQL Server.

Section 6: Most articles on query optimization omit discussions on managing plans generated by the optimizer over the lifetime of the database. These aspects of plan management can critically impact overall workload performance. We discuss a few important challenges in this context: (a) plan caching and invalidation (b) improving sub-optimal plans with execution feedback (c) query hints, which allow users to influence the plan that is chosen by the optimizer (d) optimizing parameterized queries.

Section 7: While this monograph is centered on extensible query optimizers in practice, we use this section to mention some of the open problems and a few of research directions that are being pursued.

Errata and updates: We will provide corrections and updates to this monograph at the following URL [59]. We encourage readers who discover errors in this monograph to report them to the authors via email.

1.5 Suggested Reading

Citation numbers below correspond to numbers in the References section.

[178] P. G. Selinger *et al.*, "Access Path Selection in a Relational Database Management System," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79, pp. 23–34, Boston, Massachusetts: Association for Computing Machinery, 1979. DOI: 10.1145/582095.582099

[77] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys (CSUR)*, vol. 25, no. 2, 1993, pp. 73–169

[28] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems," in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 34–43, 1998

2

Extensible Optimizers

One of the ways extensible query optimizers are built is by having an extensible set of *rules* that defines the space of all equivalent plans. As noted in Section 1, this approach centers around the concepts of logical operators, physical operators, and a set of *transformation* and *implementation rules*. The optimizer navigates the space of equivalent plans by applying rules in an order guided by a *search strategy*, and chooses an efficient plan from the space of alternative plans it explores.

In this section, we first introduce the basic concepts of extensible optimizers (Section 2.1). We then discuss two extensible optimizer frameworks in depth: the Volcano and Cascades frameworks. We start by describing Volcano and its search (Section 2.2). Next, we review the limitations of Volcano, which motivated the development of its successor, the Cascades framework (Section 2.3). We present additional optimizations and heuristics used in practice for gaining efficiency in Volcano and Cascades (Section 2.4). To illustrate how extensible query optimizers ease the incorporation of new capabilities into query processing, we describe how the optimizer of Microsoft SQL Server leverages extensibility in Cascades to implement columnstores (Section 2.5). We conclude this section by describing how extensible query optimizers gen-

erate plans that take advantage of multi-core parallelism and distributed query processing (Section 2.6).

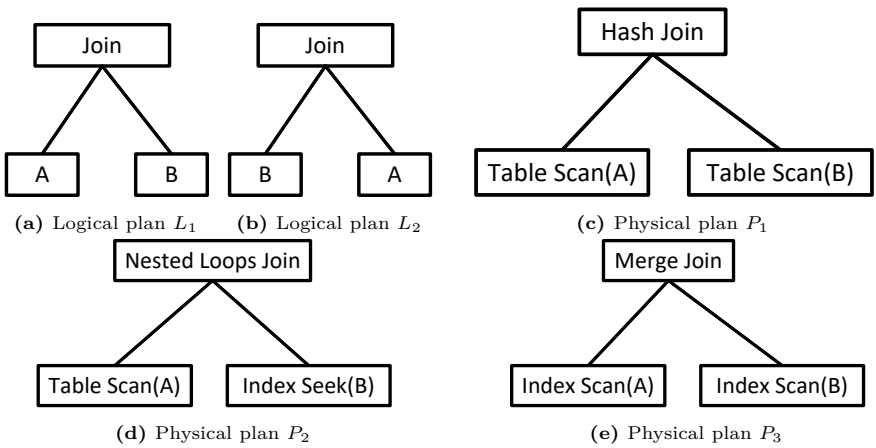
2.1 Basic Concepts

We introduce a few important concepts in rule-based extensible optimizers such as Volcano and Cascades. Note that some of these concepts such as operators and properties are not unique to Volcano/Cascades and were used in query optimizers of systems such as System R [178], Starburst [165] and EXODUS [26].

Consider the following example Query 2, where the corresponding logical and physical plans are shown in Figure 2.1.

Query 2

```
SELECT *
FROM A, B
WHERE A.k = B.k
```



(a) Logical plan L_1 **(b)** Logical plan L_2 **(c)** Physical plan P_1

(d) Physical plan P_2 **(e)** Physical plan P_3

Figure 2.1: Logical and physical plans of Query 2

Logical and physical operators A logical operator defines a relational operation on one or multiple relations, such as the Join operator between A and B in Figure 2.1a. Note that query optimizers may also introduce non-relational logical operators such as Apply for handling sub-queries

(Section 4.5.3) and Exchange for handling parallelism (Section 2.6.1). Thus, the set of logical operators used, and hence the search space of the optimizer, goes beyond those present in the SQL query. A physical operator is the implementation of an algorithm to perform an operation required for executing a query. An example of a physical operator is Hash Join (see Figure 2.1c). Note that a logical operator can be implemented by different choices of physical operators and vice versa. For example, the logical operator Join can be implemented by multiple physical operators such as Hash Join and Nested Loops Join. Similarly, the physical operator Hash Join can be used to implement multiple logical operators such as Join, Left Outer Join, and Union. In addition, a logical operator may require multiple physical operators to implement. For example, as we will introduce in Section 4.2, a logical Inner Join operator can be implemented using Sort and Merge Join operators.

Logical and physical expressions A logical expression is a tree of logical operators, and represents a relational algebraic expression. For example, the expression $L_1 : A \bowtie B$ of Query 2 is a logical expression as shown in Figure 2.1a. A physical expression is a tree of physical operators, which is also referred to as the *physical plan* or simply *plan*. For example, the expression $P_1 : HashJoin(TableScan(A), TableScan(B))$ is a physical expression that implements L_1 as shown in Figure 2.1c.

Logical and physical properties Examples of logical properties of an expression include its relational algebraic expression and the cardinality of the expression. For example, $A \bowtie B$ and $B \bowtie A$ both produce the join result of A and B, and they have the same logical properties (e.g., cardinality). Examples of physical properties of an expression include the sort order of the output of the expression and degree of parallelism, i.e., the number of threads used to execute the expression. In Figure 2.1e, the Merge Join introduces the physical property of the sort order, i.e., its output is sorted by column $A.k$.

The physical properties of an expression may be introduced due to the requirements in the original SQL query or because of the physical properties of one of its inputs. For example, consider Query 3 shown below. The query requests the top 3 tuples from table T ordered by

column $T.b$, with the constraint that column $T.a > 10$. In this case, the ORDER BY clause in the query introduces a required physical property of the sort order on $T.b$.

Query 3

```
SELECT TOP 3 *
FROM T
WHERE T.a > 10
ORDER BY T.b
```

Equivalence of expressions Two logical expressions are equivalent if the logical properties of the two expressions are the same. For example, the expression $L_1 : A \bowtie B$ in Figure 2.1a is equivalent to the expression $L_2 : B \bowtie A$ in Figure 2.1b. Two physical expressions are equivalent if their logical *and* physical properties are the same. For example, the expression $P_1 : HashJoin(TableScan(A), TableScan(B))$ in Figure 2.1c is equivalent to the expression of $P_2 : NestedLoopsJoin(TableScan(A), IndexSeek(B))$ in Figure 2.1d. However, since the expression $P_3 : MergeJoin(IndexScan(A), IndexScan(B))$ shown in Figure 2.1e delivers a sort order on $A.k$, P_3 is neither equivalent to P_1 nor P_2. Similarly, a logical expression with its required physical properties is equivalent to a physical expression if the logical properties of the physical expression are equivalent to that of the logical expression and the physical expression delivers the required physical properties.

Rules A rule rewrites an expression into another equivalent expression, and thereby enables the optimizer to explore alternative plans for the query. There are two kinds of equivalences that are of interest for the optimizer's exploration: the equivalence between two logical expressions and the equivalence between a logical expression with its required physical properties and a physical plan. A *transformation rule* rewrites a logical expression to another equivalent logical expression. For example, the join commutativity transforms $L_1 : A \bowtie B$ into $L_2 : B \bowtie A$. An *implementation rule* transforms a part of a logical expression to an equivalent physical expression with the associated physical properties. For example, the Merge Join implementation rule can transform the logical join operator in L_1 shown in Figure 2.1a into the Merge Join

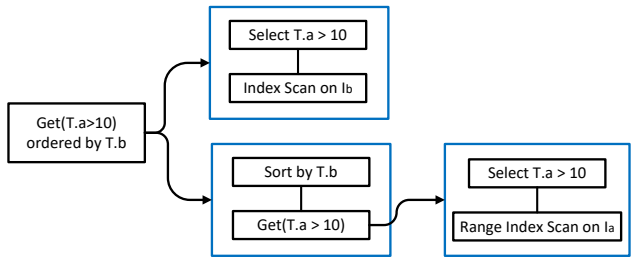


Figure 2.2: Example of enforcer during plan search

operator in P_3 shown in Figure 2.1e with the result sorted by the join column. As we will see in Section 2.2.2, the search in Volcano will apply the implementation rules recursively to transform an entire logical expression into a physical plan.

A rule is defined by two methods: $CheckPattern$ and $Transform$. $CheckPattern$ checks if the rule is applicable to the root node of the given expression and returns $True$ if applicable or $False$ otherwise. To determine if the rule is applicable, $CheckPattern$ may need to check properties of other nodes in the input expression, e.g., parent or children. If $CheckPattern$ returns $True$, then the $Transform$ method is invoked. Invoking $Transform$ produces a transformed expression as output that is semantically equivalent to the input expression. In Section 4, we will describe a selected subset of important rules that are widely used in practice.

Enforcers Enforcers are a class of physical operators that only enforce necessary physical properties of the output, such as sortedness or parallelism. The enforcer serves a similar purpose as interesting orders in System R (as mentioned in Section 1.2), but generalizes it to other physical properties beyond sortedness.

Consider the previous example Query 3. Assume the database has two indexes on T: a B+-tree index I_a with key column $T.a$, and a B+-tree index I_b with key column $T.b$. As shown in Figure 2.2, the optimizer searches for the best plan of $\sigma_{T.a>10}(T)$ with an interesting order on $T.b$ as the physical property. One possible implementation rule rewrites the expression into an Index Scan on I_b. Here, the index

provides the interesting order on $T.b$. A second transformation is via an enforcer, which adds a Sort operator on $T.b$ on top of $\sigma_{T.a>10}$ to derive the required interesting order. Because of the presence of the enforcer, its input logical expression $\sigma_{T.a>10}$ has no required physical property, which can be then transformed into a range-based Index Scan on I_a.

2.2 Volcano

2.2.1 Overview

Volcano, as described in [79, 132], is an extensible rule-based optimizer framework. It proposed several core concepts including the physical properties of expressions and enforcers (which generalizes the notion of interesting orders from System R), the memo, and the top-down dynamic programming based search algorithm that uses the notion of *promise* to determine the next move and *guidance* to control the search space explored. Section 2.1 has already introduced the basic concepts of extensible rule-based optimizers. Below, we describe the search and memo of Volcano in detail.

Search strategy Volcano separates its search into two phases: the *generation phase* and the *cost analysis phase*. In the generation phase, the optimizer generates all alternative equivalent logical expressions in the search space defined by the set of transformation rules. In the cost analysis phase, it generates the physical plans for logical expressions generated in the first phase, and returns the best plan for the original query, i.e., among the plans enumerated, the one with the lowest cost.

In contrast to the bottom-up search in System R (see Section 1.2), Volcano employs *top-down dynamic programming*, also known as *memoization*, to ensure that the search is both exhaustive and efficient. In the generation phase, the search recursively applies the set of transformation rules on the logical expressions and their inputs to generate alternative equivalent logical expressions. In the cost analysis phase, the optimizer recursively applies implementation rules to the logical expression and its inputs, and derives their cost in order to find the best physical plan of the logical expression. Throughout the search process, Volcano remembers the logical and physical expressions derived to avoid

redundant computations by caching them in a data structure called *memo*, which we describe next.

Memo The *memo* data structure compactly represents a large number of operator trees. It caches both logical and physical expressions optimized during the search. To avoid storing duplicate expressions, Volcano separates the logical expressions from the physical subplans, storing equivalent logical expressions and plans in separate types of objects. In the memo, each class of equivalent expressions is called an *equivalent class* or a *group*, and all equivalent expressions within the class are called *group expressions* or simply *expressions*. A group represents all equivalent operator trees producing the same output. For example, Figure 2.3 shows the memo for $A \bowtie B$, which has three groups: $g1 : Join(A, B), g2 : A$, and $g3 : B$. An expression is an operator that has *groups* (rather than operators) as children. Each expression can be either a logical or physical expression. A logical expression has a logical operator as its root and groups as its inputs. For example, the logical expression $e1 : Join(g2, g3)$ in group $g1$ of Figure 2.3 is a join with group $g2$ and $g3$ as inputs. A physical expression has a physical operator as its root and its inputs are groups. For example, as shown in Figure 2.3, expression $e6 : Table\,Scan(A)$ is a physical expression of group $g2$ and expression $e3 : Hash\,Join(g2, g3)$ is a physical expression of group $g1$.

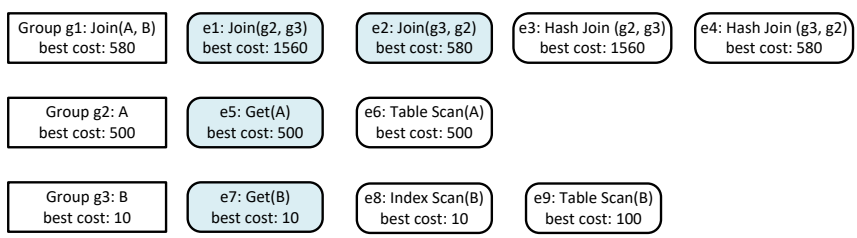


Figure 2.3: An example memo of $A \bowtie B$, where an index is available on table B. Logical expressions are colored in blue. Physical expressions and groups are annotated with the cost of the corresponding best plans.

As noted in Section 2.1, associated with each expression is a set of logical properties (e.g., cardinality) and physical properties (e.g., sort order), which are stored in the memo. As we will discuss later, Microsoft

SQL Server associates additional physical properties which it uses to enable optimization of parallel and distributed plans (Section 2.6) and for optimizing plans containing row mode and batch mode operators which are crucial for data organized as columnstores (Section 2.5).

In the search algorithm of Volcano, the memo serves multiple purposes. During the generation phase, the memo caches all the alternative logical expressions that have been generated. If a transformation leads to a logical expression of a new equivalent class, a new group is added to the memo. For example, group $g2 : A$ and $g3 : B$ in Figure 2.3 are created as the search recursively transforms the inputs of the expression $A \bowtie B$. If a transformation leads to a new expression of an already existing equivalent class, a new group expression is added to the corresponding group in the memo. For example, expression $e2 : Join(g3, g2)$ is created by transforming expression $e1 : Join(g2, g3)$ with the join commutativity rule in Figure 2.3. During the cost analysis phase, the memo caches the result of finding the best physical plan for an expression and its required physical properties to avoid duplicate computation. For the example shown in Figure 2.3, the best cost for group $g1$ is 580, and the corresponding physical plan is *Hash Join(Index Scan(B), Table Scan(A))*. We discuss the details of the search and memo in Section 2.2.2.

2.2.2 Search

As described in Section 2.2.1, Volcano's search is separated into the generation phase and the cost analysis phase. In both phases, the optimizer performs a top-down dynamic programming based, *depth-first* search. We first describe the algorithm of each phase and then illustrate the search with a concrete example.

Search algorithm Algorithm 1 shows the search algorithm of Volcano. We have taken the pseudocode for the algorithm from [79, 132] and made a few modifications for clarity. The search starts with the generation phase (line 1-10 in Algorithm 1), where the goal is to generate all the equivalent alternative logical expressions in the search space defined by the transformation rules. The optimizer performs a depth-first search to

Algorithm 1 Search in the Volcano optimizer. The groups and expressions in $Memo$ are updated in $GenerateLogicalExpr$, $MatchTransRule$, and $UpdatePlan$. The cost limit of an expression is updated as the search progresses. The cached result of the search is added to the $Memo$ at the end of $FindBestPlan$.

```
1: function GenerateLogicalExpr(LogExpr, Rules)          ▷ Generation phase
2:     for Child in inputs of LogExpr do
3:         if Group(Child) ∉ Memo then
4:             GenerateLogicalExpr(Child)                 ▷ Update memo
5:     MatchTransRule(LogExpr, Rules)
6: function MatchTransRule(LogExpr, Rules)   ▷ Apply transformation rules
7:     for rule in Rules do
8:         if rule matches LogExpr then
9:             NewLogExpr ← Transform(LogExpr, rule)   ▷ Update memo and
   record the neighbors
10:            GenerateLogicalExpr(NewLogExpr)          ▷ Only invoke if
   NewLogExpr is not previously in memo
11: function FindBestPlan(LogExpr, PhyProp, Limit)   ▷ Cost analysis phase
12:     if (LogExpr, PhyProp) in the Memo then          ▷ Check duplicate
13:         Plan, Cost ← LookUpBestPlan(LogExpr, PhyProp)
14:         if Cost ≠ null and Cost ≤ Limit then
15:             return Plan, Cost
16:         else
17:             return null, null   ▷ No such plan exists based on previous attempts
18:     MarkInProgress(LogExpr, PhyProp)          ▷ Optimize a new expression
19:     Moves ← GetAllMoves(LogExpr, PhyProp)      ▷ Generate all applicable
   transformations, potentially with heuristics or guidance
20:     SortedMoves ← SortMovesByPromise(Moves)          ▷ Order by promise
21:     BestPlan ← null
22:     BestCost ← ∞
23:     for m ∈ SortedMoves do                            ▷ Apply each move
24:         if m is a transformation rule then
25:             for NewLogExpr in GetNeighbors(LogExpr, m) do      ▷ Retrieve
   equivalent alternative logical expressions populated in the generation phase
26:                 if !InProgress(NewLogExpr, PhyProp) then
27:                     FindBestPlan(NewLogExpr, PhyProp, Limit)  ▷ Can fail if
   such a plan is not found
28:         else if m is a physical implementation rule then
29:             TotalCost ← DeriveCost(LogExpr, m)      ▷ Cost the root operator
30:             Subplans ← []
31:             for Child in inputs of LogExpr do
32:                 ChildProp ← DerivePhyProp(LogExpr, PhyProp, Child)
33:                 Plan, Cost ← FindBestPlan(Child, ChildProp, Limit − Total
   Cost)                                  ▷ Can fail if such a plan is not found
34:                 TotalCost ← TotalCost + Cost
35:                 Subplans.Add(Plan)
36:             Plan ← CreatePlan(m, SubPlans)
```

```
37:             UpdatePlan(BestPlan, BestCost, Plan, TotalCost) ▷ Update memo
38:         else if m is an enforcer then                ▷ Special handling of enforcers
39:             Cost ← DeriveCost(LogExpr, m)                    ▷ Cost the enforcer
40:             NewProp ← GetPhyProp(LogExpr, PhyProp, m)
41:             if !InProgress(LogExpr, NewProp) then
42:                 Plan, Cost ← FindBestPlan(LogExpr, NewProp, Limit − Cost)
    ▷ Can fail if such a plan is not found
43:                 UpdatePlan(BestPlan, BestCost, Plan, Cost)   ▷ Update memo
44:         Limit ← min(Limit, BestCost)                    ▷ Update the cost limit
45:     Memo.Add(LogExpr, PhyProp, BestPlan, BestCost)              ▷ Cache result
46:     return BestPlan, BestCost
```

generate all alternative logical expressions with *GenerateLogicalExpr* function (line 1). It begins by taking the logical tree expression parsed from the original SQL query and a set of rules, and recursively generates alternative logical expressions from its inputs (line 2-4). After the inputs are explored, the optimizer then generates the alternative logical expressions for the expression itself (line 5). The *MatchTransRule* function (line 6) recursively invokes *GenerateLogicalExpr* to further explore the logical expression and the corresponding new groups (line 10). In addition, the optimizer keeps track of the derivation of equivalent alternative logical expressions for each transformation rule and marks them as *neighbors* of the input logical expression under the transformation rule (line 9). This information is later used in the cost analysis phase. During the generation phase, the logical expressions are cached in the memo to avoid duplicate exploration (line 3 and 10). Throughout the process, new expressions and groups will be generated and added to the memo (line 4 and 9).

After all the alternative equivalent logical expressions are generated, the search moves to the cost analysis phase, where the goal is to find the best physical plans for the expressions (line 11-46 in Algorithm 1). The optimizer performs a top-down dynamic programming based, depth-first search to find the best physical plan with *FindBestPlan* function (line 11). It begins by invoking *FindBestPlan* with the initial logical expression parsed from the original SQL query, the required physical properties of the query result if any (such as sort orders), and an *unlimited cost*. The algorithm first checks if the input logical expression

with the required physical properties has been previously optimized (line 12). If the optimization has been performed before, and the cost of the best physical plan for the expression is less than the cost limit, this plan is returned; otherwise, the optimization has been previously attempted, but such a plan cannot be found (lines 14-17). The attempt of searching for a plan of an expression can fail if there does not exist such a plan within the given cost limit (see line 27, 33, and 42). If the optimization has not yet been performed, the expression is marked as in-progress (line 18) to avoid duplicate invocation of optimizing the same expression (see line 26). Then the algorithm proceeds to generate all possible transformations or *moves* for the expression (line 19). Alternatively, developers can implement heuristics or *guidance* to select only a subset of the moves to pursue (line 19), although this can result in suboptimal output plan if the best plan falls outside of this reduced search space. These moves are then evaluated based on how likely they are to lead to a plan with low cost, measured by their *promise* function (line 20). This function is provided by the developers of the optimizer and, by default, the search is exhaustive and all moves are pursued. We will describe promise and guidance in detail in Section 2.2.3.

There are three kinds of moves: transformation rules, implementation rules, and enforcers. If the move is a transformation rule, the neighbors of the logical expression under the rule, which are generated during the generation phase, are explored unless being marked as *in progress* (line 24-27). If the move is an implementation rule, the physical operator is costed (see Section 5 for more details on cost estimation), and the search for the optimal plan is continued recursively through optimizing the inputs of the logical expression with an updated cost limit (line 28-37). Lastly, if the move is an enforcer, a new physical property is derived (line 40), and the search for the optimal plan proceeds on the logical expression with the new physical property and an updated cost limit (line 38-43). After performing each move, the cost limit is updated with the cost of the best physical plan found so far (line 44). Once the optimization of the logical expression is completed, the best plan and its cost are cached to avoid duplicate optimization in the future (line 45). Finally, the function returns the best plan and its cost of the input expression (line 46).

Algorithm 1 leverages cost-based pruning to improve the efficiency of the search. The cost limit will be updated whenever a plan with lower cost is found (line 44), and the search for such a plan can be pruned if there does not exist a plan with the given cost limit based on previous attempts. In addition, the search can also be pruned if the cost limit becomes 0 or negative. This can happen when the cost of the root physical operator of an expression (line 29), the cumulative cost of optimized inputs of an expression (line 34), or the cost of the enforcer (line 39) is higher than the cost limit. Note that the cost-based pruning in Algorithm 1 is *sound*, i.e., it only prunes the search of suboptimal plans.

When the initial invocation of the $FindBestPlan$ on the original query returns, the cost analysis phase finishes with the best physical plan of the query and its estimated cost.

Example of Volcano search Consider optimizing the query $A \bowtie B \bowtie C$ with the following transformation rules

- Join commutativity (R1): $A \bowtie B \rightarrow B \bowtie A$
- Join right associativity (R2): $A \bowtie B \bowtie C \rightarrow A \bowtie (B \bowtie C)$

and two implementation rules

- Hash join (R3): $A \bowtie B \rightarrow HashJoin(A, B)$
- Table scan (R4): $A \rightarrow TableScan(A)$

In the generation phase, the optimizer explores all alternative equivalent logical expressions and caches them in the memo (Figure 2.4). New groups and expressions are derived and added to the memo during the exploration. It starts with the expression $A \bowtie B \bowtie C$ and updates the memo with group g_1 and logical expression e_1. Then the optimizer recursively calls the function $GenerateLogicalExpr$ in Algorithm 1 to explore the first input $A \bowtie B$ (step 1 in Figure 2.4). During the exploration of $A \bowtie B$, the optimizer creates group g_2 and logical expression e_2 and recursively explores the inputs A and B (step 2 and 3). After exploring the inputs of $A \bowtie B$, the optimizer calls the function $MatchTransRule$ in Algorithm 1 and applies the rule $R1$, resulting in $B \bowtie A$ and a new logical expression e_5 (step 4). Correspondingly, the optimizer adds e_5 to the memo and indicates e_5 is a *neighbor* of e_2 transformed from the rule $R1$. Since the inputs of $B \bowtie A$ have already been explored, the optimizer

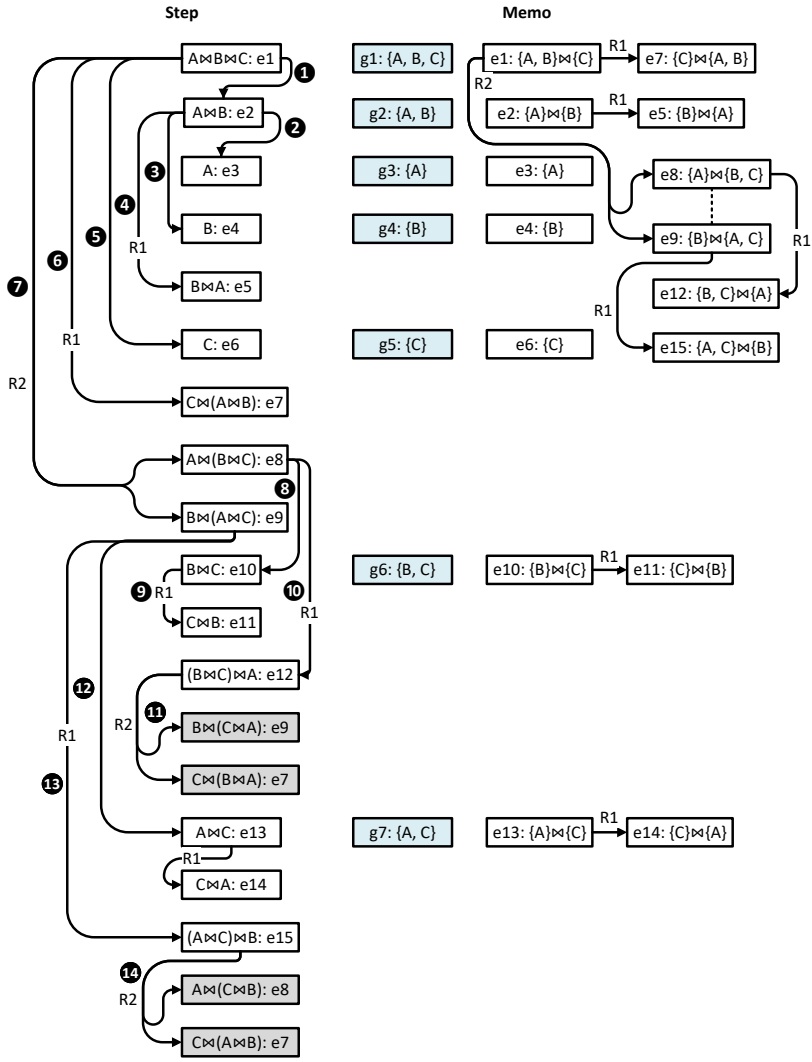


Figure 2.4: Example of the generation phase of Volcano for $A \bowtie B \bowtie C$. The left column shows the derivation of expressions. The depth-first search is annotated by steps and rules, where an arrow without a rule number indicates the recursive exploration of the inputs. The transformation that leads to a duplicate logical expression is marked in gray. The right column shows the memo, with arrows annotating the neighbors of logical expressions grouped by the rules. The dashed line connects neighbors transformed from the same rule. The groups in the memo are colored in blue.

backtracks and moves on to explore the second input of $A \bowtie B \bowtie C$ (step 5). After all the inputs are explored for e_1, the optimizer now calls $MatchTransRule$ and applies the rules to transform $A \bowtie B \bowtie C$ to equivalent logical expressions. Note that the application of rule $R2$ leads to two expressions $A \bowtie (B \bowtie C)$ and $B \bowtie (A \bowtie C)$ (step 7). This is the result of applying the right associativity to two possible expressions of the logical expression $\{A, B\} \bowtie \{C\}$, i.e., $A \bowtie B \bowtie C$ and $B \bowtie A \bowtie C$. The optimizer adds the two resulting expressions to the memo and marks e_8 and e_9 as the neighbors of e_1 transformed by the rule $R2$. Because a new group is created with the inputs of e_8, the function $MatchTransRule$ recursively invokes $GenerateLogicalExpr$ to explore the inputs of e_8 (step 8). The exploration continues until the optimizer generates all the equivalent logical expressions. Note that a logical expression can be derived more than once during the process. For example, applying $R2$ to expression e_{12} (step 11) will result in two expressions that have already been explored before, i.e., e_9 and e_7 marked in gray. The memo will detect such cases to avoid duplicate explorations.

In the cost analysis phase, the optimizer finds the best physical plan for the query (Figure 2.5). The cardinality and the simplified cost functions of the expressions are shown in Table 2.1 (see Section 5 for more details on cost estimation). The optimizer starts by invoking the $FindBestPlan$ function in Algorithm 1 for the initial tree expression $A \bowtie B \bowtie C$ with an unlimited cost (step 1). As this expression has not been optimized, the optimizer enumerates all possible moves of the expression and sort them by their *promise* (step 2). Here, the moves from transformation rules are generated based on the neighbors of the expressions in the memo, which are populated from the generation phase. Assume the rules are prioritized as $R4, R3, R1, R2$. We prioritize implementation rules over transformation rules in order to show the effect of derivation of physical plans early in the search. Because $R4$ does not apply to the expression $\{A, B\} \bowtie \{C\}$, the optimizer first chooses the move with $R3$ and derives the cost of the Hash Join operator between $\{A, B\}$ and $\{C\}$. It then recursively invokes the $FindBestPlan$ on the first input $A \bowtie B$ (step 2). The optimizer again recursively searches the best plan for the inputs of $A \bowtie B$. When the input A is fully optimized

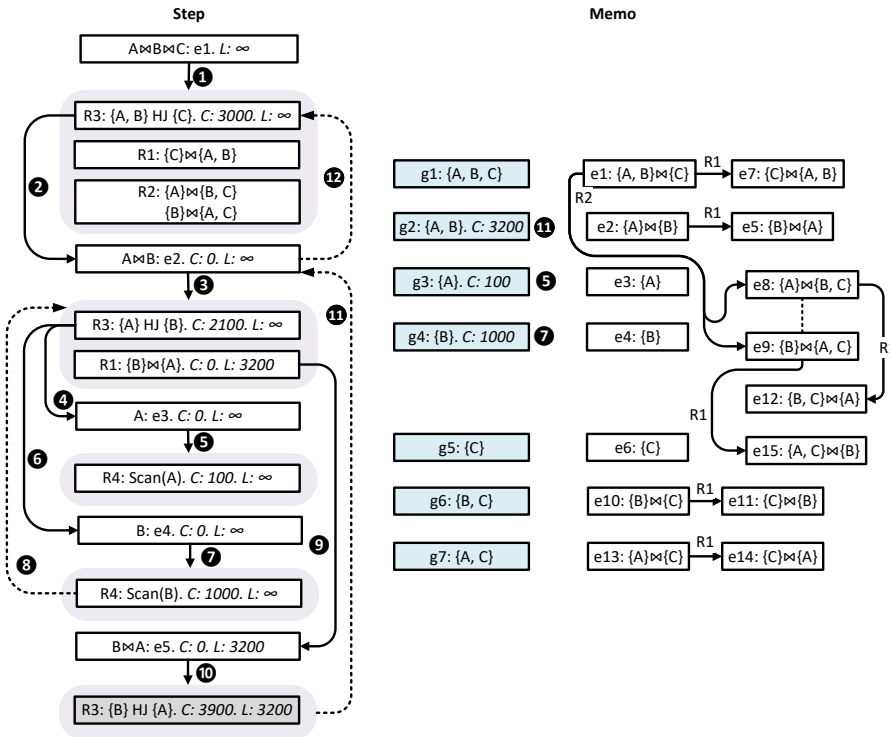


Figure 2.5: Example of the cost analysis phase of Volcano for $A \bowtie B \bowtie C$. The left column shows the process to find the best physical plan. The top-down dynamic programming is annotated by steps and sorted moves (in purple), where a dashed arrow indicates backtracking in the search. Each expression is annotated with the cost derived so far as well as the given cost limit. The expression that is pruned by cost limit is marked in gray. The right column shows the memo, with arrows annotating the neighbors of logical expressions grouped by the rules. The dashed line connects neighbors transformed from the same rule. Some groups are annotated with the corresponding step that derives the best cost. The groups in the memo are colored in blue.

Table 2.1: Cardinality and simplified cost functions used for optimizing $A \bowtie B \bowtie C$ in the example in Figure 2.5

(a) Cardinality

Logical Expression	Cardinality
A	100
B	1000
C	200
$A \bowtie B$	800
$A \bowtie B \bowtie C$	400

(b) Cost function

Physical Operator	Cost Function
$HashJoin(X, Y)$	$3 \cdot \lvert X\rvert + \lvert Y\rvert + \lvert X \bowtie Y\rvert$
$TableScan(X)$	$\lvert X\rvert$

and costed (step 5), the best cost of the group $\{A\}$ is updated to 100. At this point, the optimizer backtracks to search the best plan for the second input $\{B\}$ of $\{A\}HJ\{B\}$ (step $6-7$). After both inputs of $\{A\}HJ\{B\}$ are optimized, the cost of the expression is derived. The optimizer then backtracks and pursues the next move with $R1$ for $A \bowtie B$ with the cost limit updated to 3200 (step $8-9$). When the optimizer tries to find the best physical plan for the expression $\{B\}HJ\{A\}$, however, the cost of the Hash Join operator is already higher than the cost limit (step 10). Thus, there is no need to continue the search recursively to find the best plan of $\{B\}HJ\{A\}$. This is an example of *cost-based pruning*, and the optimizer backtracks to $A \bowtie B$ (step 11). At this point, the group $\{A, B\}$ is fully optimized, and its best cost is updated to 3200. The search process continues until all the groups are optimized. Finally, the best plan of the group $\{A, B, C\}$ is returned as the plan with the least cost for the query.

Efficiency Volcano prevents repeated optimization of the same expression by using top-down dynamic programming to memoize expressions that have already been optimized. In addition, one subplan can be the child of multiple parent expressions and thus a logical or physical expression can be a child expression of multiple parent expressions. Volcano leverages the memo data structure to avoid storing redundant copies of the same expression. Finally, the cost-based pruning during the search, such as in the example shown in Figure 2.5, allows the optimizer to potentially skip the optimization of some expressions that do not lead the optimal plan.

2.2.3 Customization of the search strategy

While the dynamic-programming based search algorithm and the memo in Volcano are effective in avoiding redundant computation, query optimization can still be expensive for complex queries. Thus, Volcano provides additional mechanisms to customize the search by allowing database engine developers to constrain the search space and to determine the order in which to explore the search space.

The search space can be controlled by providing heuristics to decide which rules are applied to the expressions, referred to as *guidance* (line

19 in Algorithm 1). Such guidance can improve the efficiency of the search, sometimes at the cost of sacrificing the quality of the plan returned. For example, heuristics can be used to activate only a subset of the rules for a more focused search space, or to limit the depth of the search by deactivating rules when the number of transformations from the original query to the current expression reaches a threshold. Note that the quality of the plan found when using the above heuristics can be impacted compared to when the search space is not constrained.

The order in which the transformations are applied during the search can also be customized by using the *promise* mechanism. Promise can be implemented as a function that takes the expression and the corresponding move as its input, and returns a number. In Algorithm 1, the moves are ordered by their promise, which influences the order of exploring the alternative plans (line 20). Since Volcano performs cost-based pruning based on the best subplans found so far, where the pruning is sensitive to the order of the exploration of the subplans, the promise can impact the efficiency of the search. For example, in Figure 2.5, the optimizer prunes the search of $HashJoin(B, A)$ based on the cost limit derived from the best subplan of $HashJoin(A, B)$; however, if the optimizer first explores the expression $HashJoin(B, A)$, then the search will not be able to prune the exploration of $HashJoin(A, B)$. The promise can be specified at the rule level as in the example of Section 2.2.2, or it can be specified based on both the expression and the move. For example, one heuristic is to give a lower promise for applying join commutativity if the resulting expression will have a larger outer side (or the build side) than the inner side (or the probe side). For example, for the moves of $\{A\} \bowtie \{B\}$ shown in Figure 2.5 (step 3), $\{B\} \bowtie \{A\}$ will have a lower promise than that of $\{A\}HJ\{B\}$ based on the heuristic. Conversely, if the build side is larger than the probe side, then prioritizing the optimization of a different join order can be beneficial as this often leads to a cheaper plan. The use of promise can benefit both exhaustive and non-exhaustive search. We will revisit the customization of the search strategy in our discussion of Cascades (Section 2.3) and techniques to improve search efficiency (Section 2.4).

2.2.4 Adding new rules and operators

Volcano's extensibility makes it easy to add a new transformation rule to the query optimizer. To add a new rule, the code that checks for patterns that triggers the rule, i.e., *CheckPattern* function, and the code that generates the transformed expression, i.e., *Transform* function, need to be implemented. In addition, the promise and guidance associated with this rule also need to be set. Once a rule is added, no changes are needed to the search algorithm to benefit from the expanded search space. We provide examples of important rules used in practice in Section 4.

To add a new physical operator, the developer needs to implement the operator including the *Open*, *GetNext* and *Close* methods of the iterator model, add new implementation rules that transform existing logical operators to the physical operator, and add derivations of the physical properties and cost estimate for the operator. Adding a new logical operator can be more involved. If the new logical operator is added to support a new SQL construct (e.g., *a window function*), then new logic is needed to derive the logical expression from the AST parsed from the query text corresponding to the new logical operator. Adding a new logical operator can also require adding new transformation rules to transform the new logical operator to existing logical operators. Optionally, new implementation rules can be added to directly transform the new logical operator into existing physical operators. Adding a new logical operator does not always require a new physical operator to be added. For example, the logical operator Union can be implemented by the previously existing Hash Join physical operator.

2.3 Cascades

Volcano introduced a framework for extensible optimizers, which contained important concepts such as transformation rules that define the search space, guidance and promise to limit and prioritize transformations, and a search algorithm that uses top-down dynamic programming with memoization and cost-based pruning. The Cascades framework, proposed in [80], retains these concepts while improving upon some of the limitations in Volcano. In this section, we start by highlighting some

of the key improvements of the Cascades framework compared to the Volcano framework (Section 2.3.1). Then we provide an overview of the search in Cascades (Section 2.3.2) followed by a detailed description of the search algorithm with pseudocode (Section 2.3.3). We revisit the same example join query from Section 2.2.2 that we used to highlight Volcano search and use it to show the similarity and differences between Cascades and Volcano (Section 2.3.4).

2.3.1 Key improvements in Cascades

Search efficiency One distinguishing feature of the search algorithm in Volcano is the separation between the exploration of equivalent logical expressions in the generation phase and the derivation of equivalent physical plans in the cost analysis phase. As a consequence, the optimizer needs to generate all the equivalent logical expressions in the generation phase. For example, as shown in Figure 2.4, the optimization of $A \bowtie B \bowtie C$ needs to generate all 7 groups and 15 expressions before deriving any physical subplan. This behavior is benign if the search is exhaustive, i.e., all the equivalent logical expressions will be explored. Unfortunately, the optimizer is often budgeted with limited resources (e.g., a *timeout*) for finding a good plan, and its search can be non-exhaustive for complex queries where the search space is large. In Volcano, if the optimizer uses heuristics to limit the search space in the cost analysis phase, its effort on exploring all equivalent logical expressions becomes wasteful if only a subset of the logical expressions are considered in the cost analysis phase. The Cascades framework improves the behavior of the optimizer by removing the separation between the generation phase and the cost analysis phase and deriving the logical expressions incrementally. Therefore, compared to Volcano, the derivation of equivalent logical expressions can be non-exhaustive depending on the heuristics used by the optimizer. We will illustrate the difference with an example in Section 2.3.4. In addition, Cascades also abstracts the ad-hoc heuristics to scope the search space in Volcano into a first-class concept, i.e., the *guidance* object, which specifies the set of activated rules applied to an expression during the search.

Task-based search algorithm Volcano uses a depth-first algorithm for the top-down dynamic programming based search (see Section 2.2.2). Instead, the Cascades framework replaces the recursive, depth-first search with a stack-based exploration using tasks, where the tasks perform actions on groups and expressions. This is described in depth in Section 2.3.3. The stack-based approach respects the dependency among tasks from top-down dynamic programming. For example, the best plan of a parent expression is derived after the best plans of its inputs are derived. The task-based approach eliminates the constraint of sequential optimization of independent expressions that arises due to the recursive, dept-first search in Volcano. For example, in Volcano moves of the same expression are optimized sequentially even when there is no dependency among the derivations of these moves. The leading benefit of the task-based approach is that it makes it easier to parallelize the search by running independent tasks concurrently on different threads. For large queries, such parallelization can lead to significantly faster query optimization time. Indeed, the Orca query optimizer (described in Section 3.2), which is built on the Cascades framework, has implemented search parallelization. The task-based search strategy also has other benefits, e.g., it allows a database developer to potentially reorder tasks, which may come from different expressions, heuristically based on promise.

Improved software design Cascades improves the abstraction of physical and logical operators by unifying them as one operator class. In Volcano, an operator is either physical or logical, which requires a special code path to handle the operators that are both physical and logical, e.g., a sargable predicate [178]. In Cascades, an operator can be logical or physical or both, which simplifies the handling of all the operators, including predicates.

Cascades also unifies transformation and implementation rules as one *rule class*, where each rule *checks the pattern* of an expression (i.e., *CheckPattern* function) and *transforms* the expression if the check passes (i.e., *Transform* function). In addition to this unification, enforcers are also modeled via a new type of rule: *enforcer rules*. An enforcer rule inserts physical operators that change the physical properties of a plan.

The invocation and the application of the enforcer rule are similar to other rules. For example, an enforcer rule of adding a Sort operator can be applied if a sort order is required by the physical properties of an expression, which is tested in *CheckPattern*. The *Transform* function generates the same expression but without the required physical property of sort order since that property will be satisfied by the newly added Sort operator. Compared with Volcano, where a code path exists to add an enforcer in an ad-hoc manner as needed during the search (see Algorithm 1), Cascades simplifies the search since no special casing is required for enforcers in the search algorithm itself.

Furthermore, Cascades allows all possible transformations to be applied using a function provided by the database developer, i.e., a *function rule*. This can be convenient when multiple substitutes can be derived by repeatedly applying a rule. One example of function rules is the *index strategies*. Given a logical expression to retrieve the data from a table, there can be potentially multiple options to access the table with different indexes, including index intersections (see Section 4.1). Instead of deriving one index strategy per application of a rule, it is more convenient and efficient to derive all applicable index strategies as the moves of the expression in a single invocation by leveraging a function rule.

2.3.2 Search overview

We start with an overview of the search and tasks in Cascades optimizer in this section, then explain the search algorithm in details in Section 2.3.3, and finally walk through the search algorithm with an example in Section 2.3.4.

The search in Cascades uses the same top-down dynamic programming as in Volcano, where the derived expressions are stored in the *memo* to avoid redundant optimization. However, compared to Volcano, the search in Cascades is broken into more fine-grained pieces called *tasks*, to enable the search to interleave the transformations of logical and physical expressions. These tasks are queued in a stack, which performs last-in-first-out (LIFO) by default. The optimizer can potentially be customized to prioritize the tasks in alternative schedules, as

long as the dependency of the tasks specified by the top-down dynamic programming is respected.

The six types of tasks in the Cascades optimizer are described in [80]. Figure 2.6 shows the overview of the workflow for the tasks. This workflow corresponds to the pseudocode presented in Algorithm 2, which is based on one implementation of Cascades, specifically in Microsoft SQL Server. Due to the specific implementation, there are a few differences between this workflow and the workflow diagram in [80] in terms of the invocations among the tasks.

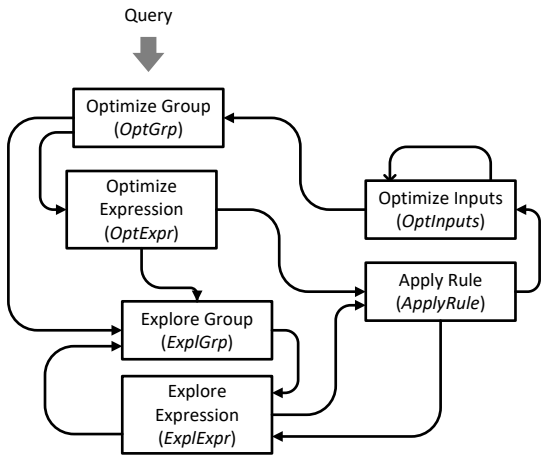


Figure 2.6: Workflow of the tasks in the search of Cascades

At a high level, there are four tasks that *optimize* or *explore* a group or an expression within a group (see Section 2.2 for the definition of groups and expressions). The goal of optimizing a group ($OptGrp$) or an expression ($OptExpr$) is to derive the best physical plan of the group or the expression respectively. The goal of exploring a group ($ExplGrp$) or an expression ($ExplExpr$) is to generate alternative equivalent logical expressions for the group or the expression respectively. The entry point of the search for the best physical plan of an expression is the task $OptGrp$ on that expression. We note that $OptGrp$ is also the entry point for finding the best physical plan of the original SQL query. $OptGrp$ first invokes $ExplGrp$ to generate alternative logical expressions of the input expression, and then it derives the best physical plan of the group

by iteratively optimizing each expression in the group with *OptExpr*. The task *ExplGrp* invokes *ExplExpr* iteratively for the expressions that have not been explored in the group. In the task *ExplExpr* and *OptExpr*, the rules are checked against the input expression, and the fifth task type of *applying a rule* (*ApplyRule*) will be created for each matched rule. In addition, because exploring an expression or optimizing an expression can create new groups, both *ExplExpr* and *OptExpr* will recursively invoke the exploration of their inputs (*ExplGrp*). The task *ApplyRule* performs the actual transformation of the expression by applying the matched rule. Depending on rules, new expressions can be created and need to be explored (*ExplExpr*) or the inputs of the expression need to be optimized (*OptInputs*). The final task *OptInputs* iteratively invokes the optimization on the inputs of an expression (*OptGrp*), and it derives the best physical plan of the expression after all its inputs are optimized.

Since the groups and expressions are explored incrementally during the search, each expression needs to keep track of what rules have already been tried on the expression to avoid the overhead of trying the same rule more than once on that expression. Each expression has a bitmap to remember the rules that have been applied to the expression, which are deactivated from application using the *guidance* mechanism (Section 2.3.1). While both *OptExpr* and *ExplExpr* attempt to match rules that have not been applied to the expression before, *ExplExpr* only matches transformation rules and *OptExpr* matches all other rules, including implementation and enforcer rules.

The physical properties of the expressions are derived and passed through various tasks. For example, when optimizing the inputs of an expression, the required physical properties of the inputs are derived from the expression and passed to the *OptInputs* tasks. Similarly, when applying an enforcer rule to an expression in *ApplyRule*, one or more expressions can be created with the corresponding required physical properties introduced by the enforcer as the result of the transformation.

Throughout the search process, the memo is used to store new groups and new expressions to avoid duplicate computation. The memo structure is similar to the one shown in Volcano (Figure 2.3), except that the groups and alternative logical expressions are derived incrementally

in the tasks instead of being pre-populated in the generation phase of Volcano. Similar to Volcano, the Cascades optimizer also performs cost-based pruning in the search. The best physical plans and the associated cost are derived in *OptInputs* and stored in the memo. These costs are used for deriving the cost limit and passed to the tasks for pruning the search.

Similar to Volcano, Cascades provides the mechanism of promise to customize search. For example, the promise can be used within the *ExplExpr* to prioritize the matching rules or within *ApplyRule* to order the newly generated expressions. Moreover, the task abstraction in Cascades provides additional flexibility in customizing the search with promise and guidance. For example, promise can be used to prioritize across all the independent tasks, and the guidance can also be used to implement heuristics to reduce the search space at the expression level. As another example, the guidance uses a bitmap to keep track the deactivation of rules for each expression across tasks as described previously. This can also be used to reduce the overhead of deriving duplicate expressions by deactivating certain rules as the tasks are performed. For example, if join commutativity has been applied to an expression, this rule can be deactivated to avoid unnecessary consecutive application of join commutativity on the transformed expression.

2.3.3 Search algorithm

In this section, we describe the search of Cascades as well as the tasks in details. Algorithm 2 shows the simplified pseudocode for the search in Cascades with a LIFO stack.

The optimizer starts the search by invoking *OptGrp* with the logical query tree corresponding to the original SQL query (obtained after parsing and validation as described in Section 1), and an unlimited cost limit. If the input to *OptGrp* has not been explored, the optimization of the group is deferred and a new task *ExplGrp* is scheduled (line 4-5); otherwise, all expressions in the group are scheduled to be optimized (line 7-8). Note that the optimization of the group needs to wait for the exploration of the group to complete. Such dependency is implemented by pushing the *OptGrp* task back into the LIFO stack before pushing the *ExplGrp* task (line 4-5).

Algorithm 2 Tasks of the Cascades optimizer. The memo gets updated in *GetGroup* and *UpdateMemo* if a new expression and/or a new group gets created. Each expression keeps track of what rules have been tried with $IsApplied$.

```
1: function OptGrp(expr, limit)                          ▷ Find the best plan for a group
2:     grp ← GetGroup(expr)
3:     if !grp.Explored then                             ▷ Explore the group first
4:         tasks.Push(OptGrp(grp, limit))
5:         tasks.Push(ExplGrp(grp, limit))
6:     else
7:         for expr ∈ grp.Expressions do
8:             tasks.Push(OptExpr(expr, limit))
9: function ExplGrp(grp, limit)                          ▷ Explore every expression in the group
10:    grp.Explored ← true
11:    for expr ∈ grp.Expressions do
12:        tasks.Push(ExplExpr(expr, limit))
13: function ExplExpr(expr, limit) ▷ Explore an expression and its inputs with rules that matches the pattern of the expression
14:    moves ← ∅
15:    for rule ∈ Transformation Rules do
16:        if !expr.IsApplied(rule) and rule.CheckPattern(expr) then      ▷ Can optionally apply guidance
17:            moves.Add(ApplyRule(expr, rule, promise, limit))
18:    Sort the moves by promise in ascending order for the LIFO stack
19:    for m ∈ moves do
20:        tasks.Push(m)
21:    for childExpr in inputs of expr do
22:        grp ← GetGroup(childExpr)
23:        if !grp.Explored then
24:            tasks.Push(ExplGrp(grp, limit))
25: function OptExpr(expr, limit)                        ▷ Find the best plan for the expression
26:    moves ← ∅
27:    for rule ∈ Rules do
28:        if !expr.IsApplied(rule) and rule.CheckPattern(expr) then      ▷ Can optionally apply guidance
29:            moves.Add(ApplyRule(expr, rule, promise, limit))
30:    Sort the moves by promise in ascending order for the LIFO stack
31:    for m ∈ moves do
32:        tasks.Push(m)
33:    for child ∈ inputs of expr do
34:        grp ← GetGroup(child)
35:        if !grp.Explored then
36:            tasks.Push(ExplGrp(grp, limit))
```

```
37: function ApplyRule(expr, rule, promise, limit)      ▷ Apply the rule to the expression and create additional tasks if needed
38:     newExprs ← Transform(expr, rule)
39:     UpdateMemo(newExprs)
40:     Sort the new expressions by promise in ascending order for the LIFO stack
41:     for newExpr ∈ newExprs do
42:         if Rule is a transformation rule then
43:             tasks.Push(ExplExpr(newExpr, limit))
44:         else
45:             limit ← UpdateCostLimit(newExpr, limit)      ▷ Can fail if the cost limit becomes 0 or negative
46:             tasks.Push(OptInputs(newExpr, limit))
47: function OptInputs(expr, limit)      ▷ Optimize the inputs of the expression
48:     childExpr ← expr.GetNextInput()      ▷ Check if all inputs have been optimized
49:     if childExpr is null then
50:         memo.UpdateBestPlan(expr)
51:         return
52:     tasks.Push(OptInputs(expr, limit))      ▷ Add a task to optimize the next input
53:     limit ← UpdateCostLimit(expr, limit)      ▷ Update the cost limit for remaining inputs based on the inputs that have been optimized. Can fail if the cost limit becomes 0 or negative
54:     tasks.Push(OptGrp(GetGroup(childExpr), limit))      ▷ Optimize the current input
```

The task *ExplGrp* explores all expressions in the group (line 9-12). When exploring an expression, the optimizer checks all the transformation rules that have not been applied to the expression and matches each rule's pattern to the expression (line 15-16). Optionally, the guidance associated with the expression can be used to constrain the search space by skipping certain rules. Upon finding a matched pattern, the matched rule is added as a *move* of this expression (line 17). The optimizer sorts the moves by their promise, then it schedules tasks to apply these rules (line 18-20). If the corresponding group of an input of the logical expression has not been explored, a *ExplGrp* task will be scheduled for the input if needed (line 21-24).

When optimizing an expression in *OptExpr*, the optimizer finds all the rules that have not been applied to this expression and matches each rule's pattern to the expression (line 27-28). Here, the guidance of the expression can be potentially used to scope the set of rules to

match. The matched rules are added as the moves of this expression (line 29), and the optimizer schedules these moves by their promise to apply the rule (line 30-32). Since matching the pattern of a rule to an expression can introduce inputs that belong to a new group, e.g., join associativity, a task of *ExplGrp* will be scheduled for the inputs (line 33-36).

The task of *ApplyRule* applies the actual transformation of an expression. This step can result in more than one new expression since there may be multiple options to match the pattern, i.e., multiple *bindings* (line 38). These new expressions are added to the memo (line 39). The optimizer sorts these expressions by their promise (line 40), and then schedules additional tasks based on the type of the rule applied (line 41-46). The cost limit can be updated if an implementation rule or an enforcer is applied to the expression (line 45). If the expression is transformed by an implementation rule or an enforcer rule, the required physical properties of the new expressions need to be derived and passed to the corresponding tasks.

Finally, the *OptInputs* task is invoked iteratively on the inputs of an expression (line 47-55). Similar to *OptGrp*, the parent task needs to wait for the completion of optimizing its inputs. This dependency is implemented by keeping track of the progress in the parent task, i.e., the next input to optimize, and pushing the parent task back into the LIFO stack so that the optimizer can *backtrack* to the parent task (line 52). Upon the invocation of *OptInputs*, it first checks if there is more input to optimize (line 48-49). If not, the best subplan is cached in the memo and returned (line 50-51); otherwise, the parent task pushes two tasks into the stack. First, it will push itself back to the stack for optimizing the next input (line 52). Then it schedules the task of *OptGrp* on the current input with an updated cost limit, i.e., by reducing the cost limit by the best cost of previous input (line 53-54).

Similar to Volcano, the search in Cascades can be pruned if the cost limit becomes 0 or negative (line 45 and 53), which indicates that the optimizer cannot find a plan with the given cost limit. When there are no more tasks to perform and the stack becomes empty, the search finishes with the best plan of the query along with its estimated cost.

2.3.4 Example of query optimization in Cascades

We now revisit the same example as described in the Volcano optimizer (Section 2.2) to show how query optimization works in the Cascades framework. To recap, our simple example optimizes the query $A \bowtie B \bowtie C$ with two transformation rules and two implementation rules as shown in Table 2.2a. We assign higher promise to the two implementation rules so that we can illustrate the cost derivation and pruning for physical plans early in the search. Our simple Cascades optimizer sets the following two *guidance* for the search. First, if an expression is transformed by join commutativity, the join commutativity rule is deactivated in subsequent transformations. This guidance is triggered when applying $R1$, and it avoids duplicate derivation without impacting the quality of the output plan from the search. Second, the plan does not contain cross product. This guidance is triggered when applying $R2$, where a new join expression, potentially with cross products, can be created after applying join associativity. Note that while it is often desirable to avoid cross products, in general, this guidance is a heuristic, and in some cases can compromise the optimality of the plan. Table 2.2 shows the details of the rules, cardinality, and simplified cost functions of the example (see Section 5 for more details on cost estimation).

Table 2.2: Rules, cardinality, and simplified cost functions used in the example of optimizing $A \bowtie B \bowtie C$ in Cascades.

(a) Rewriting rules, promise, and guidance

Rule ID	Rule	Promise	Guidance
$R1$	Join Commutativity	1	No consecutive application
$R2$	Join Right Associativity	2	No cross product
$R3$	Join to Hash Join	3	N/A
$R4$	Get to Table Scan	4	N/A

(b) Cardinality

Expression	Cardinality
A	100
B	1000
C	200
$A \bowtie B$	800
$A \bowtie C$	20000 (cross product)
$A \bowtie B \bowtie C$	400

(c) Cost function

Physical Operator	Cost Function
$HashJoin(X, Y)$	$3 \cdot \|X\| + \|Y\| + \|X \bowtie Y\|$
$TableScan(X)$	$\|X\|$

Our simple Cascades optimizer uses a LIFO stack to schedule the tasks. Figure 2.7 shows a simplified illustration of how the tasks are pushed into and popped out of the stack during the query optimization. When query optimization starts, a task of optimizing the corresponding group (*OptGrp*) of the logical expression $A \bowtie B \bowtie C$ with unlimited cost ($Limit : \infty$) is scheduled to the stack (t_1 in Figure 2.7), which adds a new group ($\{A, B, C\}$) and a new expression ($\{A, B\} \bowtie \{C\}$) in the group to the memo, where the expression is parsed from the AST of the query. Next t_1 is popped out of the stack. Since the expression has not been explored, the *OptGrp* is deferred (t_2), i.e., by pushing itself back into the stack, and the optimizer also pushes a new task to explore the group (*ExplGrp*) to the stack (t_3). Now t_3 is popped, and it creates the task t_4 to explore the expression (*ExplExpr*). Next t_4 is popped, and it finds all the transformation rules that match the pattern in the expression $\{A, B\} \bowtie \{C\}$ (i.e., R_1, R_2), prioritizes the possible moves by their promise (i.e., R_2, R_1), and pushes corresponding tasks to applying the rules to the stack (task t_5-t_6). In addition, *ExplExpr* also creates tasks to explore its inputs (i.e., t_7, t_8), which results in new groups (i.e., $\{A, B\}$, $\{C\}$) and new expressions (i.e., $\{A\} \bowtie \{B\}$, $\{C\}$) added to the memo. At this point, the tasks in the stack from bottom to the top are t_2, t_5, t_6, t_7, t_8.

The optimizer then takes t_8 from the top of the stack, and it tries to explore the expression C (t_9). Since no more transformation rules apply, the optimizer backtracks to the task t_7 to explore the group $\{A, B\}$. t_7 creates the task t_{10} to explore the expression $A \bowtie B$, which then applies $R1$ and creates the task t_{11} as well as the tasks to explore its inputs (task t_{12}-t_{13}). The optimizer continues to explore the expression $\{A\}$ and $\{B\}$, and then it backtracks to task t_{11} and applies the join commutativity rule on $A \bowtie B$. With t_{11}, a new expression $B \bowtie A$ is created, and consequently, the optimizer schedules a task t_{16} to explore the expression. In the task t_{16}, while the rule $R1$ is applicable, it will lead to duplicate expressions. We observe however, that such a duplicate transformation is avoided because of the guidance on no consecutive application of $R1$ (t_{17}).

Next, the optimizer backtracks to t_6 and applies join associativity to the expression $\{A, B\} \bowtie \{C\}$. Note that because of $\{A, B\}$ has two

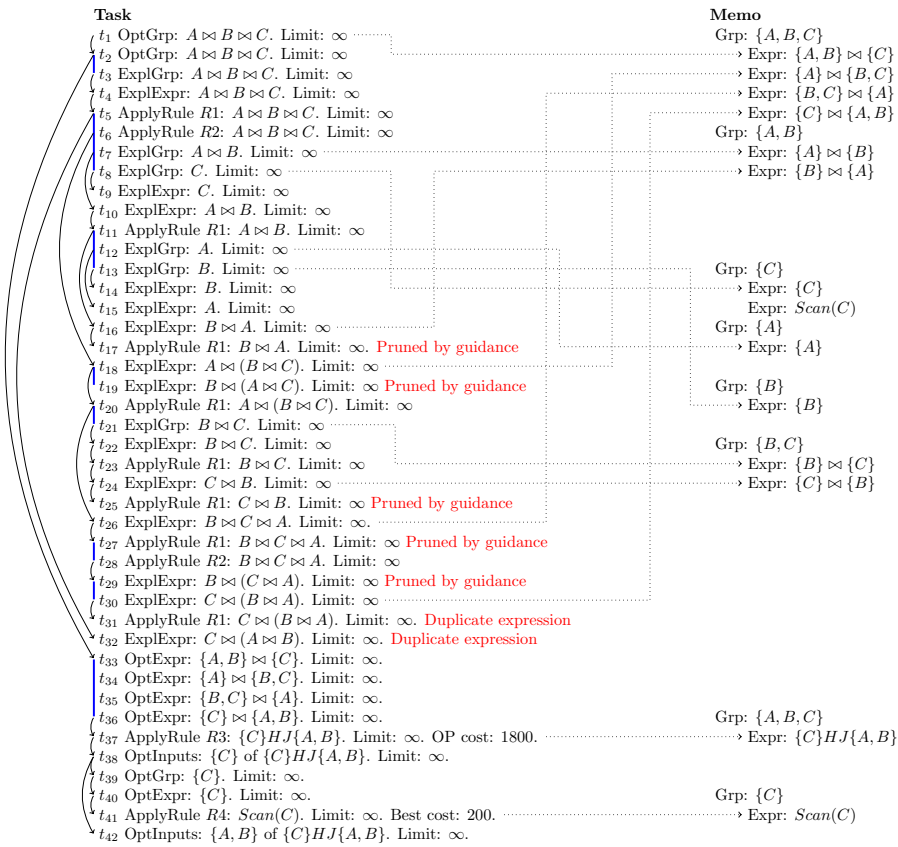


Figure 2.7: Optimize $A \bowtie B \bowtie C$ in Cascades with a LIFO stack. The **Task** column shows how the tasks are pushed into and popped out of the stack. The arrow and the blue line span the child task(s) created by a parent task, e.g., t_5-t_8 are created by t_4. The **Memo** column shows how the groups and expressions are added to the memo. The dotted arrow indicates the task that creates the group and/or the expression.

expressions created by t_7 and t_{16} respectively, this rule has two bindings: $A \bowtie (B \bowtie C)$ and $B \bowtie (A \bowtie C)$. The optimizer creates two new tasks to explore the expression t_{18} and t_{19}. In the task t_{19}, since the expression $A \bowtie C$ is a cross product, which is disabled by our guidance, this task is discarded for further exploration. Note that this also shows an example of using guidance to scope the search space at the expression level in Cascades.

The optimizer continues the exploration of the expressions until task t_{32}, where all the alternative equivalent logical expressions defined by the set of rules and guidance are explored. During the process, there are several tasks where the transformed expression turns out to be duplicated (task t_{31}, t_{32}), which is detected by the memo. Note that because of the guidance on the heuristics of no cross product, the optimizer has only derived 6 groups and 11 logical expressions. In contrast, in Volcano, where the generation phase is exhaustive, the optimizer creates all 7 groups and 15 expressions, as shown in Figure 2.4.

After all the expressions are explored, the optimizer backtracks to the very beginning of the stack to optimize the group (t_2). For each expression in the group $\{A, B, C\}$ from the memo, the optimizer creates a corresponding task *OptExpr* to optimize the expression (task t_{33}-t_{36}). To optimize the expression in t_{36}, the optimizer applies the implementation rule $R3$ for the join operator at the root of the expression and costs the operator (t_{37}), and then it schedules the task to optimize the inputs of the expression. As shown in Algorithm 2, the optimization of all the inputs is implemented by optimizing one input at a time. The optimizer first schedules a task to optimize $\{C\}$ (task t_{37}-t_{41}). When the input $\{C\}$ is optimized, the optimizer backtracks and optimizes the next input $\{A, B\}$ in task t_{42}.

The optimizer repeats this process until all expressions are optimized and returns the best physical plan.

2.4 Techniques to Improve Search Efficiency

Despite the efficient dynamic-programming based search in Volcano and Cascades frameworks, exhaustive search of the best physical plan is still expensive for complex queries. Therefore, in practice, additional heuristics are used to further reduce the cost of query optimization. Here, we outline a few common optimizations.

Simplification rules Empirically, some rules simplify the logical tree *and* almost always improve the cost. Such rules are referred to as *simplification rules*. Examples of simplification rules include eliminating duplicate predicate filters and pushing down simple predicate filters below joins. Instead of creating alternative expressions from these rules,

it is more efficient to simply *replace* the input expression with the transformed expression. Thus, the simplification rules can be implemented as a pre-processing step before applying other rules, or they can be applied to the logical expression tree of the original query (see Figure 1.1) before the query optimization.

Macro rules A macro rule is used to transform the shape of an expression significantly in a single move, which otherwise would require the application of many rules to achieve. Macro rules allow optimizer developers to encode common patterns that are likely to generate plans with low cost. Transformations with macro rules are typically assigned with high promise (Section 2.2.3) so that they are performed at the early stages of the search. An example of a macro rule is heuristic join ordering for star and snowflake queries, which we will describe in detail in Section 4.6.

While macro rules do not augment the search space of the optimizer, they can improve the efficiency of the search and the quality of the plans. First, macro rules perform multiple moves with one invocation, which short-circuits the search and reduces the overhead to reach the transformed expression. Second, if macro rules lead to the derivation of physical plans with low cost early in the search, it can improve the efficiency of the search with more effective cost-based pruning. Finally, when the search is not exhaustive, as macro rules are often invoked at early stages of the search, using these rules ensures the heuristic-based plans are included in the search.

Parallelizing query optimization In Cascades, the search algorithm is broken down into small tasks, and the tasks without dependencies can be performed in parallel. For example, while a Optimize Inputs task depends on the completion of the optimization task for each input, the substitutes created from applying the rule with the same pattern can be optimized in parallel. Prior work evaluates parallel query optimization with Cascades [195]. In the initial phase, the tasks are largely sequential. As more alternative transformations are explored with dynamic programming, more tasks become independent. In general,

the more complex the query and the larger the number of tasks, the more speed-up can be gained with parallel query optimization.

Multi-stage optimization Since exhaustive search can take too long, query optimization is often time bounded. Thus, it is important for the search of the plan to be *any time*, meaning that the optimizer finds a good plan early in the search and improves the quality of the plan over time. One approach to make the query optimization any time is to break it down into multiple stages and progressively explore the search space. As the search progresses, the improvement of the plan quality is often diminishing as the best plan found so far typically gets closer to the optimal plan. Thus, an ideal query optimizer should be cost-efficient: the optimizer should find a good plan as quickly as possible and search for a better plan if needed.

In multi-stage optimization, the optimizer may be invoked several times. In each stage, a different set of rules are activated that gradually increases the search space. Usually, the set of rules for each stage is statically defined in the optimizer, and the sets are not mutually exclusive, e.g., implementation rules of access methods (see Section 4.1) may be included in the rule set of every stage. If the optimizer finds a good plan quickly by using a restricted set of rules in a stage, optimization can stop and the current best plan is returned. Staged optimization can be implemented by using guidance to specify a different set of rules to include at each stage. The alternative expressions explored in previous stages can be cached in the memo for reuse in the next stage.

In particular, the query optimization has four stages in Microsoft SQL Server. For simple queries where the optimal plan is more easily determined, such as lookups on a single table, the optimizer can skip the expensive optimization process and immediately generate a physical plan. This is called a *trivial plan* [184]. For queries that do not qualify for the trivial plan, the optimizer expands the search space progressively in three stages: *transaction processing*, *quick plan*, and *full optimization* [53]. These stages increase the set of transformation rules enabled during the plan search by leveraging the *guidance* mechanism (see Section 2.2.3). For instance, in the transaction processing stage, a set of join orders is initially generated without the use of sophisticated transformations.

For example, *macro rules* (e.g., transformations for join ordering for star and snowflake queries in Section 4.6) and reordering of group-by and joins (Section 4.4), are disabled. In the quick plan stage, additional transformations are enabled, including most rules on join ordering, group-by and aggregation. If the estimated cost of the best plan from any of the two stages meets a cost threshold, the plan is selected for execution; otherwise, the full optimization stage is invoked. This final stage activates all the transformation rules to allow for a thorough exploration of the search space with cost-based pruning, until the search is exhausted or the timeout is reached. The timeout is set as the maximal *number of tasks* performed during the search instead of the maximal elapsed time. This design ensures the output plan of the query is stable across different runs. The best plan produced by this stage is selected for execution.

2.5 Example of Extensibility in Microsoft SQL Server

Microsoft SQL Server has exploited the extensibility of Cascades to make several enhancements to its query processing capabilities over the years. Below, we describe one such example, showing how Microsoft SQL Server exercises multiple aspects of the extensibility in Cascades to add support for column-oriented processing.

Column-oriented database systems (referred to as *columnstore*) have been shown to accelerate the performance of analytic workloads due to more efficient I/O that scans only the required columns, as well as batch-oriented query processing that can leverage techniques such as *vectorization* [1, 19]. For *mixed* workloads consisting of both transactional and analytic queries, also referred to as Hybrid-Transactional-Analytical-Processing (HTAP), it can be beneficial to have both row-oriented (*rowstore*) and column-oriented processing capabilities in the same database engine. In fact, a given table can be stored as a B+-tree index (which is row-oriented) and copy of the table can also be stored in a columnstore format. Thus, a given execution plan can contain access to both row store and columnstore indexes.

Microsoft SQL Server incorporated column-oriented storage and batch-oriented query processing into its query optimization frame-

work [113] to support the above scenario by leveraging various aspects of the extensibility of Cascades. Microsoft SQL Server introduced a new index type, i.e., *columnstore index*, along with the corresponding new physical operator *Columnstore Index Scan*. A columnstore index provides the access to one or more columns of a table. Different from B+-tree indexes, a columnstore index does not provide any sort order, and it also does not support range queries or point accesses to the data. Hence, additional predicate filters are needed to retrieve the selected data.[1] Thus, new implementation rules were added to transform logical expressions into a Columnstore Index Scan operator. To fully leverage the efficient execution of plans involving columnstore indexes, Microsoft SQL Server also introduced a new batch execution mode. In contrast to *row mode*, where the physical operators process a row at a time, in *batch mode* the physical operators process a batch of rows at a time.[2]

Since two versions of a physical operator, row mode and batch mode, are available as options, it increases the search space that the optimizer must consider. Microsoft SQL Server incorporated row mode and batch mode as a new physical property of an expression which is stored in the memo (Section 2.2.2). It also introduced a new *Adaptor* operator, which takes as input rows from its child operator in one format, e.g., in batches, and converts those rows to be ready for processing in the other mode, e.g., into individual rows for processing by its parent operator. This, along with the corresponding enforcer rules, enabled Microsoft SQL Server to integrate the choice of row mode and batch mode in the search space.

For example, Figure 2.8 shows a query plan with a Hash Join (batch mode) involving table R and S which are stored as columnstore indexes, followed by a (row mode) Nested Loops Join with an Index Seek on a B+-tree index I_d on table T. Note that the result of the Hash Join needs to be converted from batches to individual rows with the Adaptor operator before they can be consumed by the Nested Loops Join operator. We note that the cost functions of the operators need to be extended to

[1] It supports zone maps, which can discard segments of data that contain no qualifying rows.

[2] Batch mode operators can sometimes be beneficial even for pure rowstores to improve the efficiency of execution.

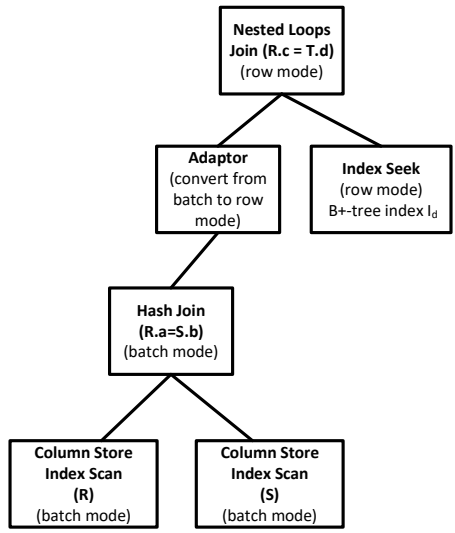


Figure 2.8: A plan showing mixed mode execution containing operators in batch mode and row mode in Microsoft SQL Server. Conversion from batch to row mode is done by the Adaptor operator, which is added via enforcer rules.

reflect the cost of processing in batch mode. Similarly, the cost of the Adaptor operator must capture the overhead of the conversion of its input from one mode to another mode.

As shown above, by leveraging the extensibility of Cascades for adding new operators, implementation rules, and enforcers, the Microsoft SQL Server query optimizer fully integrated columnstore indexes as well as new batch mode versions of physical operators, while requiring no change to the search algorithm. As we will see in Section 2.6, the extensibility of Cascades has been similarly leveraged to enable the optimizer to consider plans where operators can exploit multi-core parallelism, as well as to enable query optimization for distributed query processing.

2.6 Parallel and Distributed Query Processing

In a DBMS that executes on a single server, the query optimizer needs to be able to effectively exploit *multi-core parallelism* to speed up query execution. We discuss how an extensible query optimizer can generate

plans with operators that execute in parallel in Section 2.6.1. Second, in analytic database engines, since query execution is done using multiple compute nodes that need to communicate with each other over the network, query optimizers must generate distributed query plans that take into account the cost of data movement across compute nodes. We show how an extensible query optimizer can handle the challenges arising out of distributed query processing in Section 2.6.2.

2.6.1 Multi-core parallelism

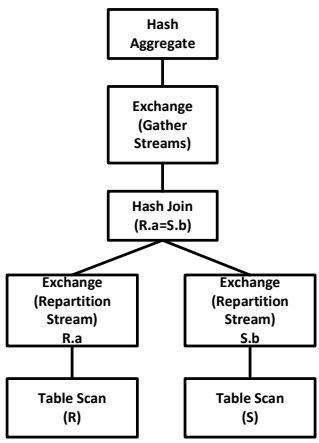
Database servers often have access to 10s or even 100s of cores that they can use to run their workload. Therefore, when a query needs to perform operations such as scan, join, and aggregation on large amounts of data, executing such expensive operators using multi-core parallelism (i.e., running them *multi-threaded*) can significantly reduce the elapsed time of the query. However, parallel execution also introduces overheads, such as CPU, memory, and the cost of synchronization across threads. Since parallel execution may not always be appropriate for an operator due to these overheads, the optimizer needs to make a cost-based decision on the *degree of parallelism* (DOP), i.e., the number of threads to use, for each operator in the plan. In this section, we describe how the *Exchange* operator, which was first introduced in the context of Volcano [79], supports parallelism as a general mechanism, and thereby simplifies the incorporation of parallelism into a plan. Using Microsoft SQL Server as an example, we also discuss how the optimizer expands its search space to include plans containing a mix of single-threaded (*serial mode*) and multi-threaded (*parallel mode*) operators by leveraging the extensibility of Volcano/Cascades frameworks.

Parallelism using the Exchange operator Exchange [79], which is not a relational operator,[3] was proposed to handle operations related to parallelism such as synchronization and flow control, thereby enabling parallel execution while allowing all other operators in the DBMS to be designed and implemented single-threaded. Below, we use an example

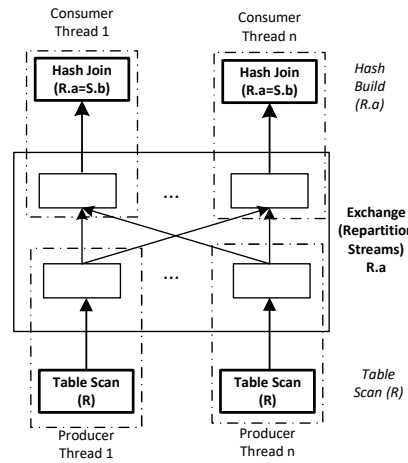
[3]It is referred to as a meta-operator in [79], similar to Choose-Plan, which we discuss in Section 6.2.

based on Microsoft SQL Server to illustrate how the Exchange operator enables parallelism for single-threaded physical operators.

Figure 2.9a shows an example of a plan with three Exchange operators. In this plan, the build and probe side of the Hash Join execute multi-threaded, whereas the Hash Aggregate executes in serial mode. The mixed execution of parallel and serial mode in this plan can be desirable when the base tables are large, i.e., benefit from multi-threaded processing, while the join result is small, therefore making it more efficient to be processed by a single thread. The plan shows two kinds of Exchange operators: *Repartition Streams* and *Gather Streams*. All Exchange operators follow a producer-consumer architecture. We illustrate the behavior of the *Exchange (Repartition Streams)* operator in Figure 2.9b. With the Exchange operator, the Hash Join operator essentially becomes a partitioned hash join. Assume there are n producers and n consumers in the *Exchange (Repartition Streams)* $R.a$ operator. Each of the n producer threads scans a subset of the rows from table R, and depending on the value of the partitioning column $R.a$, it writes the row to the buffer of the appropriate consumer thread.



(a) Example plan with Exchange operators



(b) Illustration of the producer/consumer model of Exchange (Repartition Streams) for the pipeline of operators that builds the hash table for the Hash Join

Figure 2.9: Parallelism using Exchange operators

Each consumer thread, i.e,. a thread of the Hash Join operator which builds the hash table, consumes rows from its respective buffer, applies the hash function, and inserts the row into a bucket of the hash table. After a consumer thread of the hash build completes processing all the rows in its partition, it requests the input from the probe side to perform the hash probe. Similarly, the *Exchange (Repartition Streams) S.b* operator enables parallelism for the hash probe of the Hash Join. Note that the build and probe side use the same partitioning function, and the consumer thread processes the rows from the probe side of the corresponding partition of the build side. Since the necessary logic of partitioning and flow control are encapsulated inside the Exchange operator, the Hash Join operator itself is executed single-threaded by each consumer thread. The *Exchange (Gather Streams)* operator has n producers and 1 consumer, and it gathers the output rows from the n threads of the Hash Join probe into a single output stream. The output rows are consumed by the single-threaded Hash Aggregate operator.

The Exchange operator follows the same iterative execution model as described in Section 1 with *Open*, *GetNext*, *Close* methods [79]. In the *Open* call, the Exchange operator allocates its buffers and initializes threads for its child operators. In the *GetNext* call, the consumer thread in the Exchange operator returns one row from the respective buffer as described in the above example. When all the threads finish the processing, the Exchange operator calls *Close* to clean up its state.

Finally, we note that while the Exchange operator can simplify the implementation of parallelism, it can still be beneficial to design and implement multi-threaded operators. For example, with Exchange operator, each thread would build its own hash table in the Hash Join operator, which could result in hash tables of variable sizes due to data skew. In contrast, if the Hash Join operator is designed to be multi-threaded, it is possible to build a single hash table that supports concurrent operations, which avoids the above issue with data skew.

Extensions to the query optimizer The support of the mixed execution modes requires the query optimizer to explore the search space with serial and parallel mode for each operator in the plan and cost them appropriately. Microsoft SQL Server integrates the serial and parallel

execution modes into the optimizer by extending the *physical properties* (see Section 2.1) of an expression with the execution mode. Consequently, the optimizer uses the Exchange operator as the *enforcer* to change the physical property from serial to parallel or vice versa, and the corresponding enforcer rules are added to support enforcing the required physical property. For example, if the required physical property of a parent expression (e.g., the Hash Aggregate in the example) is serial mode, the optimizer can either request the input of the expression to execute in serial mode, or it can require the input with the physical property of parallel mode and insert an Exchange operator in-between that changes the execution mode from parallel to serial. The extensibility mechanisms in Cascades, including physical properties, enforcers, and rules, enable parallelism to be supported without requiring any change to the search algorithm. We note that new cost functions are needed to reflect the cost of parallel processing for operators in parallel mode as well as that of the Exchange operator.

2.6.2 Distributed query optimization

Data analytics in enterprises analyze large volumes of data, whose size can range to petabytes or more. In the cloud, data used for analysis is stored in a blob storage service and analyzed using a distributed query processing engine [150]. A distributed query processing compute engine consists of a set of *compute nodes* where each compute node is a DBMS process. For a given a query plan, these compute nodes need to work together to execute the operators in the plan and compute the results of the query. There is also a control node that is responsible for generating the plan for the SQL query, and collecting the (partial) results produced by the compute nodes and returning the final results of the query to the application.

An example of the architecture of a cloud data analytical engine for Microsoft Fabric Synapse Data Warehouse (Fabric DW for short), is shown in Figure 2.10. Each compute node is a Microsoft SQL Server DBMS process. The "Frontend" SQL Server is the control node. This node has the distributed query optimizer, referred to as the *Unified Query Optimizer* (UQO), that generates the plan for the query. The

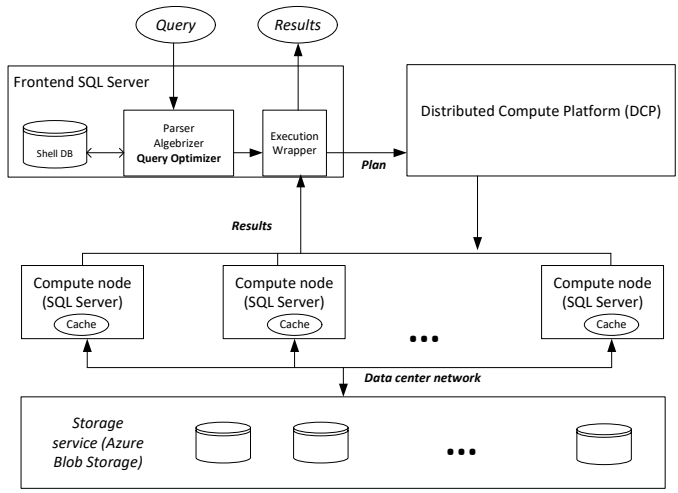


Figure 2.10: Architecture of Microsoft Fabric Synapse Data Warehouse

Distributed Compute Platform (DCP) is responsible for scheduling the execution of operators on the compute nodes and for moving data among compute nodes during execution. More details on Fabric DW and its query optimizer may be found in [4, 24].

Challenges in distributed query optimization

Given the large amounts of data that need to be analyzed, a natural approach is to take advantage of data parallel computation by having multiple compute nodes, each executing the query plan on a subset of the data, and thereby improving the response time of the query. While such a strategy is possible for a plan containing only a simple operator such as Scan, it is much more challenging for commonly used operators in analytic queries such as Join and group-by. This is because with distributed execution of queries, these operators may require *data movement between compute nodes* over the data center network, which can be expensive and may dominate the time to execute the query. Consider Query 4 below:

Query 4

```
SELECT *
FROM R INNER JOIN S
ON R.a = S.a AND R.b = S.b
```

Suppose table R is retrieved from the storage service and is *distributed*, i.e., split into independent subsets by hashing each row in R using a hash function on the columns {R.a, R.b}. We denote this distribution as *Hash* ({a, b}). Consider the operator Hash Join (R, S). Since the query is an equi-join involving columns R.a and R.b, each compute node executing this operator can join a different subset of R, corresponding to a distribution of R. However, to guarantee correctness, we need to ensure that all rows of S that could potentially match rows in that distribution of R being processed by that compute node are available. There are different ways to guarantee the above property. For example, we could *Broadcast* S, i.e., send a full replica of S, to each compute node. Alternatively, we could first hash distribute S on Hash (S.a, S.b) and send each distribution of S to the corresponding compute node, referred to as *Shuffle*. The decision of which alternative is more efficient depends on the cost of data movement required, as well as the cost of computation on the compute nodes. For example, when R is a large table distributed on non join key columns, if S is small, then the option to Broadcast S can be more efficient compared to the alternative which requires a Shuffle of R and S. However, when R and S are both large, using Shuffle of both tables on the join keys can be more efficient. Similar choices of Shuffle vs. Broadcast also arise with intermediate results of the query.

It may appear that one could find the best *serial* plan, i.e., one that ignores how the data is distributed and the cost of data movement, and then use data parallelism to execute the best serial plan. However, it has been shown that such an approach can lead to poor quality plans [181]. For example, a join order that is sub-optimal in a serial plan can become the optimal join order with distributed query execution because it performs much less data movement. Thus, the search space of alternative plans that a distributed query optimizer must consider can become very large due to alternatives arising from choices in distributions.

Extensions to the query optimizer We use the Unified Query Optimizer (UQO) [24] to describe how an extensible optimizer built using the Cascades framework handles distributed query optimization. Later in this section, we briefly discuss distributed query optimizers in other

cloud analytic engines. UQO consists of a set of extensions to Microsoft SQL Server's query optimizer that enables it to generate distributed query execution plans in a cost-based manner. To derive the cost of a plan, the optimizer uses statistics on the data stored in a *shell* database on the frontend SQL node, i.e., control node. A shell database consists of all metadata of the database (but not the actual data) required for plan generation and costing, including information about tables, columns, indexes, materialized views, and statistics.

The key changes for enabling distributed query optimization include introducing: (1) New physical *properties* for capturing information related to the distribution for an expression in the memo (see Section 2.2.1) (2) Implementation rules for operators such as Join, Group-by, and Union, that leverage distribution information of its inputs, as well as create requests for the required distribution property of its inputs. (3) An enforcer rule that guarantees that the required distribution properties are satisfied by an expression. Below, we briefly describe each of these key changes. We use the example Query 4 introduced earlier to illustrate the concepts. The table *R(a, b, c)* is distributed using *Hash({a,b})*, whereas table *S(a, b, d)* is distributed on *Hash({a})*.

Distribution properties

The output of any expression, whether the Scan of a table from the storage service, or an intermediate expression of the query, has *Distribution* properties that capture how that expression is distributed. Here, we discuss a subset of the ways an expression may be distributed: (1) Serial: the expression is not distributed, e.g., this is what would be expected for the final query result. (2) Replicated: data is replicated on all compute nodes. This is useful for a broadcast join. (3) Hash(*cols*): Data is hash-distributed on *cols*. This is important for a distributed Hash Join. (4) Any(*cols*): Data is distributed on *cols* with an unknown distribution function. This is useful for a group-by (cols) which can take advantage of any distribution on its grouping columns and thus, it may require its child satisfies Any (cols). Finally, we note that if an expression is distributed on a set of columns C, e.g., Hash (C) or Any (C), then any two rows that agree on C are part of the same distribution.

UQO uses distributions as *derived* properties to determine how an expression is distributed. It also uses distributions as *required* property to request during the optimize group (*OptGrp*) task of the Cascades search algorithm (see Section 2.3.3).

As explained in the context of Algorithm 2, an optimization task of a group with a *required* physical property (e.g., sort order) may result in tasks with requirements of specific physical properties propagated. The distribution physical property behaves similarly. Assume that a group (e.g, Join (R,S) in Query 4) has the *required distribution* as Hash distribution on the column {R.a, R.b}. It can generate tasks for optimizing Scan(R) with a required hash distribution on either {a,b}, or {a}, or {b}. The above constraint on hash distribution is expressed in UQO as Any({a,b}). To facilitate the alternative of broadcast join, an optimization task for Scan(R) is also created with a *required distribution* for Replicated. Given the equi-join constraint, tasks for Scan(S) with the required distribution constraint Any({a,b}) as well as Replicated are created.

Implementation rules

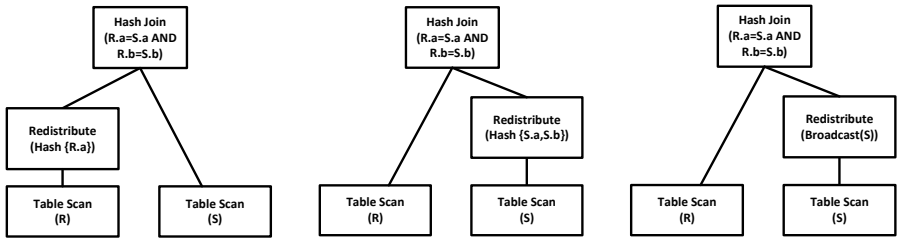
Once the optimization tasks for Scan(R) and Scan(S) are completed, we must consider implementation rules for Join(R,S). Specifically, let us consider distributed hash join. For this join implementation to be *correct*, it is crucial that both R and S are hash distributed on an *identical set of columns*. Thus, if Scan(R) were hash distributed on {a} and Scan(S) is hash distributed on {a,b} (i.e., two sides of the Join with different distributions), then the propagation of the required distribution property will be incorrect. For broadcast join, one of R or S must have the distribution property of Replicated.

In addition to ensuring correctness, another important consideration is pruning the many alternatives for distribution. As implied by the example above, all identical subsets of columns of equi-join columns of the two relations would result in a correct implementation of distributed hash join. However, in a distributed join scenario, alternatives that avoid data movement are the alternatives that need to be considered. For example, we leverage the distributions of the base tables when

appropriate to reduce data movement. This is analogous to what was done for handling interesting physical orders. For example, if the result of Scan(R) was hash distributed on {a}, of special interest are cases where Scan(S) is distributed on {a} (and vice versa). If neither Scan(R) nor Scan(S) was hash distributed on {b}, then that alternative for partitioned hash join of Join (R,S) on {b} is unattractive as that alternative requires data movement for both R and S. Thus, {a} will be considered as an example of an *interesting distribution* for Scan(R) and {b} will not be considered an interesting distribution. The intuition is similar to that for Merge Join with a sorted relation on join column for one of the relations. For more information on how interesting distributions are leveraged to reduce the space of alternatives considered by the optimizer for Join, group-by and Union, we refer the readers to [24].

Enforcer for redistribution

To ensure that the constraints of required distribution property for the optimization tasks are satisfied, the query optimizer for the distributed queries needed to introduce a new enforcer. This enforcer inserts a physical Redistribute operator, which performs data movement and guarantees that the required distribution properties are satisfied. Examples of redistribute operations include *Merge Move* that coalesces all data that resides in multiple distributions into a single node (e.g., to prepare the final output of the distributed query at the frontend SQL node), *Broadcast Move* to transfer data from each source distribution into all the target nodes to support the Replication distribution requirement, and *Hash Move* that hashes all data in the source distributions and sends them to the appropriate target distributions. In our example Query 4, if the required distribution for the second input of *Hash Join(R, S)* is *Hash({a,b})*, then a *Hash Move Redistribute* operator may need to be applied to the result of *Scan(S)* to ensure this property. Figure 2.11 illustrates different plans for our example Query 4, for the join between R and S and illustrates the roles the Redistribute operator plays.



(a) Redistribute R (shuffle to compute nodes)

(b) Redistribute S (shuffle to compute nodes)

(c) Redistribute S (broadcast to all compute nodes)

Figure 2.11: Examples of distributed plans for Query 4. R is distributed using Hash({a, b}), S is distributed using Hash({a}).

Distributed query optimization in other engines

As we have explained, the distributed query optimizer in Fabric DW treats the distribution of data and their movement like other aspects of search space during query optimization. This is why they needed to introduce the new physical property of distribution. Like interesting orders, the distribution property can have impact on the choice of the plan. AWS Redshift [10] too uses cost-based optimization that accounts for the cost of data movement. For example, if a join key matches the underlying distribution of both participating tables, the optimizer picks a plan where each compute node processes the join locally and therefore avoids unnecessary data movement. Other alternatives explored in the industry have opted for staged query optimization and use of runtime adaption instead of holistically considering the impact of data distribution. For example, Snowflake [50] postpones distribution related decisions until execution time, e.g., the type of data distribution for joins. This may be viewed as an example of two-stage query optimization with the second phase related to distribution and data movement deferred to execution time when accurate statistics are available. BigQuery [133] uses an adaptive approach, reminiscent of plan competition [7] (see Section 6.2). It uses the default operator of distributed hash join where both the relations execute a shuffle. However, if the data movement for one of the relations in the join finishes within a pre-determined threshold due to its small size, the join implementation is switched to a broadcast join. Thus, BigQuery cancels the second shuffle and

replicates the first relation, thus enabling a broadcast join. Although both Snowflake and BigQuery do not consider as many alternatives as Fabric DW, the dynamic schemes of the above systems avoids the potential bad plans chosen by the optimizer due to errors in cost and cardinality estimations.

2.7 Suggested Reading

Citation numbers below correspond to numbers in the References section.

[83] G. Graefe *et al.*, "The Volcano Optimizer Generator: Extensibility and Efficient Search," in *Proceedings of IEEE 9th international conference on data engineering*, pp. 209–218, 1993

[79] G. Graefe, "Volcano - An Extensible and Parallel Query Evaluation System," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, 1994, pp. 120–135

[80] G. Graefe, "The Cascades Framework for Query Optimization," *IEEE Data Eng. Bull.*, vol. 18, no. 3, 1995, pp. 19–29

3

Other Extensible Optimizers in the Industry

When describing Volcano and Cascades in Section 2, we use Microsoft SQL Server's query optimizer to illustrate how it implements the abstractions of Cascades and how it leverages extensibility in Cascades to simplify incorporation of new functionality into the query optimizer. In this section, we briefly review a few other extensible query optimizers used in the industry: Starburst (Section 3.1), Orca (Section 3.2), Apache Calcite (Section 3.3), and Catalyst (Section 3.4). While Orca and Calcite are based on Cascades and Volcano respectively, Starburst and Catalyst use different extensibility frameworks. Therefore, for Starburst and Catalyst, we draw comparisons with Volcano/Cascades, whereas for Orca and Calcite we compare with Microsoft SQL Server's implementation of Cascades. Finally, while PostgreSQL's query optimizer does not possess the extensibility capabilities[1] of the other query optimizers discussed in this section, due to its popularity in the industry, we include a short review of its query optimizer in Section 3.5.

[1]The PostgreSQL database is known for its extensibility with respect to data types, indexing, functions, etc. but its query optimizer is not built using an extensible framework.

3.1 Starburst

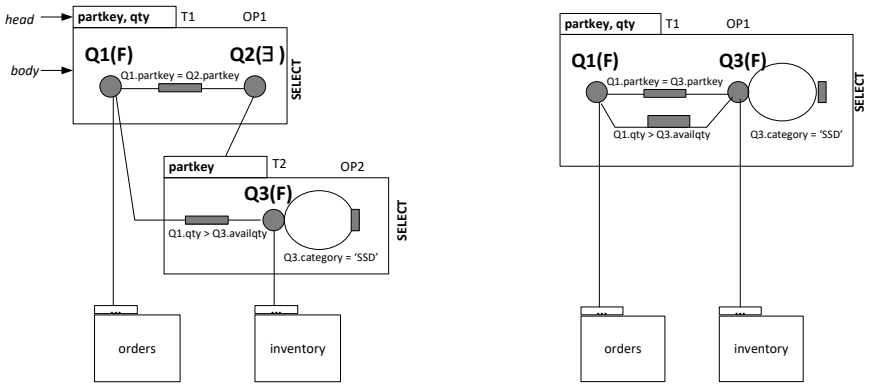
Starburst is an extensible query optimizer designed for IBM DB2 as described in [86, 166]. Query optimization in Starburst is done in distinct phases: parsing and semantic checking, query rewriting and plan optimization. A key data structure used in Starburst is the Query Graph Model (QGM), which represents a SQL query. We begin with a brief description of the QGM.

Query Graph Model The QGM representation is used through all the phases of query optimization. In the QGM, a *box* represents a query block and labeled arcs between boxes represent predicates across blocks. Each box contains information about predicates and properties such as orderedness of the results of that query block. To illustrate the QGM, consider Query 5 shown below that finds all parts from the category 'SSD' in the orders, where there is insufficient quantity of the part in the inventory to meet the order.

Query 5

```
SELECT partkey, qty
FROM orders Q1
WHERE Q1.partkey IN (
    SELECT partkey
    FROM inventory Q3
    WHERE Q3.category = 'SSD'
            AND Q3.availqty < Q1.qty)
```

Figure 3.1a shows the QGM for the above query. Each SELECT block in the query appears as a *box* in the QGM. The head of the box describes the output relation produced by the operations represented by the box. For example, in the top box, $T1$, with columns *partkey* and *qty* is the relation generated by the box $OP1$. The body of the box represents the operations. Each vertex represents an iterator, which may be a stored relation (annotated as F) or a quantifier (annotated as $\exists$ or $\forall$). For example, in box $OP1$, the vertex $Q1$ represents access to the stored table *orders*, and $Q2$ represents access to the intermediate relation $T2$ generated by $OP2$ and has the $\exists$ quantifier corresponding to the IN predicate. Each conjunct of a predicate is represented by a line



(a) Original QGM **(b)** QGM after applying rewrite rules

Figure 3.1: Example of Query Graph Model (QGM) in Starburst

connecting the vertices, and a loop represents a single table predicate, e.g., for predicate *Q3.category = 'SSD'*.

Rule rewrite phase In the query rewrite phase, rules are applied to transform a QGM into another logically equivalent QGM. A rule is defined by a pair of functions: a *condition* function and an *action* function. Each function is passed a context, which corresponds to either a box in the QGM (i.e., a SELECT block) or a quantifier. The *condition* function performs a check and returns True/False. If the condition function evaluates to True, the *action* function performs the transformation. The outer loop of the query rewrite phase is driven by a search strategy that traverses the QGM (both depth-first and breadth-first strategies are supported), and provides the context (box or quantifier) for the rules to work on. The rule engine decides which rules to apply and in what order. The control strategies for rule application include: sequential, priority, and statistical, where the next rule is chosen randomly based on a user defined probability distribution.

Figure 3.1b shows the QGM for the above query after applying the rewrite rule that replaces a nested subquery with a join. Note that the modified QGM is a single block query. In general, when an alternative QGM is generated, it may not be possible to determine if it will be more efficient without using cost estimation, which is not available during

the query rewrite phase. Hence, in such cases, Starburst maintains both (in general multiple) alternatives as children of a CHOOSE operator. During the plan optimization phase described below, each QGM is optimized and the one with lowest cost is selected. The CHOOSE operator is similar to Choose-Plan operator of Volcano/Cascades [45] discussed in Section 6.2.

Plan optimization phase In the plan optimization phase, an execution plan is chosen for a QGM. In Starburst an execution plan is *constructed bottom-up* using a set of grammar-like production rules [123]. The "terminals" of these rules are physical operators, referred to as LOw-LEvel Plan OPerator (*LOLEPOP*). They include ACCESS (access methods such as indexes and heaps), GET (for each row retrieves a set of columns from a table), JOIN (Hash, Nested Loops and Sort-Merge), SORT. Rules are named, parameterized objects referred to as STrategy Alternative Rules (STARs for short). STARs are the "non-terminals" in the grammar, and each STAR defines an execution plan or sub-plan. A STAR can reference other STARs or LOLEPOPs.

Every table, either a base table or the result of a sub-plan, has three types of properties: a relational description, a set of physical properties such as the order of rows, and a set of estimated properties such as cardinality and cost. The properties required for a relation are ensured using a special *Glue* mechanism. The Glue mechanism can add an operator that guarantees the required property, e.g., if one of the inputs to a Merge Join is not ordered appropriately, Glue introduces a **SORT** operator on the result of the input. It returns the lowest cost alternative among all alternatives that meet the required property.

An example STAR called JoinMethod that defines three kinds of join methods is shown in Figure 3.2. **JOIN** is a LOLEPOP and JoinMethod is a STAR. A **JOIN** takes the following parameters: the join type (Nested Loops, Sort-Merge, Hash Join), outer input (T1), inner input(T2), join predicates, and residual predicates.

The relational description of a plan, its estimated cost, and physical properties (e.g., order) are propagated as plans are built bottom-up. When a STAR rule is derived, comparable plans that represent the same logical and physical properties but have a higher cost are pruned.

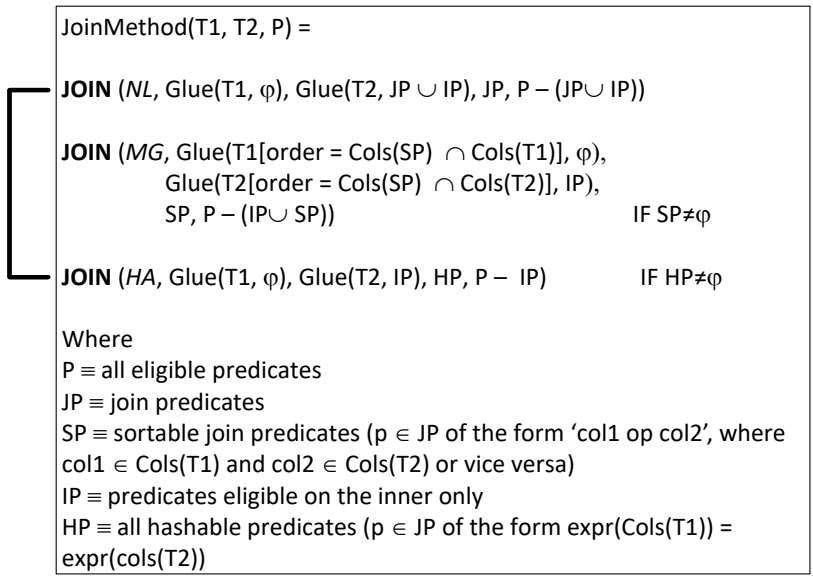


Figure 3.2: Example STrategy Alternative Rule (STAR) that can generate a plan with one of the three different join methods.

Join enumeration in Starburst is similar to System R's bottom-up algorithm [117, 178] with some differences. For example, when a STAR corresponding to a join is applied, what constitutes a joinable pair of relations is expanded to also allow cross-products when the input relations are small. Another example is to allow composite inners, i.e., where the inner side is a join, thereby allow bushy plans.

Comparison with Volcano/Cascades There are several similarities and differences between Starburst and Volcano/Cascades. We note that Starburst uses two distinct rule engines, one per phase, whereas in Volcano/Cascades there is a single rule engine that supports both transformation and implementation rules.

The mechanism of specifying a query rewrite rule in Starburst and a transformation rule in Volcano/Cascades share similarities: the (condition, action) function pair and (CheckPattern, Transform) functions respectively serve the same purposes. Therefore, in both cases arbitrary checks and transformations can be encoded. Further, both allow control

for the order in which rules are applied: using promise in Cascades and rule priority in Starburst. However, unlike the Volcano/Cascades framework which does goal-driven application of rules, Starburst's rewrite phase applies rules in a forward chaining manner. Since the query rewrite phase in Starburst is not cost-based, this module must either retain all QGM alternatives generated via rule applications or heuristically prune rule applications and thereby potentially compromise plan quality. However, it is worth noting that since the condition function's code of a rewrite rule can include arbitrary checks, in principle it can leverage information about cardinality and cost for heuristic pruning.

The separation into two distinct phases is common to Starburst and Volcano, but is different from Cascades where the generation of physical plans is interleaved with the application of logical transformation rules in a single phase. Physical plan generation in Starburst is quite different compared to both Volcano and Cascades. It constructs physical plans in a bottom-up manner by combining physical operators into trees using dynamic programming, whereas Volcano and Cascades both use top-down dynamic programming with memoization. Finally, the satisfaction of required properties is guaranteed via the Glue mechanism in Starburst, whereas enforcers provides this capability in Volcano and Cascades.

3.2 Orca

Orca is an extensible cost-based query optimizer for distributed database systems developed by Pivotal [183]. It serves as the optimizer for two different analytic database systems: Greenplum Database, a shared-nothing massively parallel processing (*MPP*) data warehouse engine, and HAWQ, a distributed, SQL-compliant query engine on top of HDFS. Similar to Microsoft SQL Server, Orca is also based on the Cascades framework, and it uses the same concepts of the memo, logical transformation rules and implementation rules, property enforcement using enforcer rules, etc. Below we focus on two main differences between Orca and Microsoft SQL Server.

Comparison with Microsoft SQL Server's optimizer First, unlike Microsoft SQL Server, where the query optimizer is tightly coupled with

the database server, Orca is designed to work with multiple DBMSs, including Greenplum Database and HAWQ, and therefore runs as a standalone process. In order to support such decoupling from the DBMS, Orca proposes the *Data eXchange Language (DXL)*, a framework for exchanging information between the DBMS and Orca. This framework uses an XML-based language to encode the information such as the input query, database metadata (e.g., tables and columns), and the output plan. The database system needs to include translators that consume and emit information in DXL format, e.g., to convert a query parse tree into a DXL query, to convert a DXL plan into an executable plan in the corresponding database system, and to provide metadata. Orca runs outside the DBMS as a standalone process, and this architecture provides it the flexibility to be used with any DBMS with the necessary translators.

Second, as noted in Section 2.3, the Cascades framework is designed to allow parallelization of query optimization. Orca leverages this capability and uses multiple CPU cores to parallelize query optimization to enhance its efficiency [195]. The optimization process is broken into small work units called jobs. These jobs correspond to different types of actions, e.g., generating a logically equivalent group expression, generating an implementation of an expression, finding the lowest cost plan for a group, etc. Jobs in Orca correspond to the concept of tasks in Cascades (Section 2.3). Orca tracks the dependency graph of these jobs and implements a specialized job scheduler to maximize the fan-out of the jobs for parallel query optimization. An example of a dependency is that a group expression cannot be optimized until its child groups are optimized. During parallel query optimization, different concurrent tasks may issue concurrent requests to modify a group in the memo. To minimize synchronization overheads, when a task with a goal, e.g,. exploring the same group, is executing, and a new task with the same goal arrives, the new task is queued until the current task completes.

3.3 Calcite

In the past couple of decades, there has been a proliferation of specialized database engines, such as NoSQL engines, column stores, and

stream processing engines, that are often used within a single enterprise. Hence querying data across these database engines has gained importance. Apache Calcite [15] provides query optimization and query execution capabilities over multiple data processing systems such as Apache Cassandra [93], Apache Hive [97], and Apache Flink [25]. Thus, Calcite can be used as a standalone database engine that federates multiple storage and query processing backends. Therefore, with regard to federated query optimization, it shares similar goals as Orca (see Section 3.2).

Calcite relies on *adapters* to access data from a source DBMS. The adapter allows Calcite to access the metadata information of the database and allows it to define how data should be retrieved from the source DBMS (i.e., "access methods") for a given Calcite query. Note that an access method in this context corresponds to a relational expression on the source database. A simple example is that a Table Scan (T) operator which returns columns A and B in the Calcite query may get converted to a query "SELECT A, B FROM T" on a relational engine such a Apache Hive. Observe that the execution of this query on the relational engine might itself rely on access paths supported by that engine.

Comparison with Microsoft SQL Server's optimizer Calcite has an extensible query optimizer that is based on the Volcano framework. Its extensibility with respect to transformation rules is therefore similar to Orca and Microsoft SQL Server. Calcite's query optimizer takes advantage of the extensibility of Volcano to generate plans for federated queries, e.g., a query that references one table residing in a Flink database and a second table residing in Hive. Calcite's rules, e.g., its *CheckPattern* function for a rule pertaining to access methods, must be aware of the source DBMS's capabilities and restrictions. For example, Apache Cassandra partitions data by a subset of columns in a table, and within each partition, sorts the rows based on another set of columns. Hence, a rule that pushes a Sort in the query into a Sort in Apache Cassandra must be aware of the above properties of partitions, and add appropriate checks before transforming the expression. The Calcite query optimizer provides a cost-based, top-down dynamic programming

algorithm similar to Volcano's cost analysis phase. Calcite also provides the capability of multi-stage optimization, similar to that described in Section 2.4.

Finally, we briefly mention Substrait [190], a format for describing compute operations on structured data. It consists of a formal specification and a cross-language binary representation. Substrait can be used for scenarios where the specification of a computation on structured data must be communicated across different systems. For example, in the context of query optimizers for federated and distributed databases such as Calcite and Orca, Substrait can be used to serialize and communicate a plan between the optimizer and the execution engine.

3.4 Catalyst

Spark SQL is a module in Apache Spark [9] that integrates relational processing with Spark's Scala based functional programming API. Catalyst is an extensible optimizer for Spark SQL and has been open-sourced as part of Apache Spark. Catalyst is built using Scala, a functional programming language also used for developing Spark SQL. The input query is represented as a Scala tree object where the nodes are operators. Catalyst contains a library for representing and manipulating trees.

Comparison with Volcano/Cascades Similar to Volcano and Cascades, Catalyst allows database engine developers to add a set of transformation and implementation rules. A rule in Catalyst is a function that transforms one tree into another. Although, in general, a rule may require expressing arbitrary code on the input tree to check applicability of the rule (similar to the *CheckPattern* function in Volcano/Cascades), for many rules the built-in pattern matching functions of Scala can be used to find and manipulate sub-trees, thereby making such rules concise to express.

However, in contrast to Volcano and Cascades, there are key differences in the search algorithm. In Catalyst, the search algorithm is broken down into a sequence of phases shown in Figure 3.3. The *analysis* phase resolves tables and columns referenced in the query by consulting the database catalog, determining types of expressions, etc.

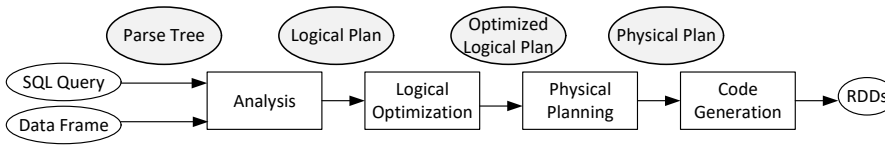


Figure 3.3: Phases of query planning in Catalyst. The shaded ovals are Catalyst trees.

The *logical optimization* phase applies a set of transformation rules, such as predicate push down, null propagation, and constant folding, to the input tree, resulting in a modified tree. These rules are applied iteratively until no more rules can be applied. Since the application of transformation rules in this phase are not cost based, rules in this phase are limited to those that are almost always beneficial in practice, similar to simplification rules in Volcano/Cascades described in Section 2.4. The *physical planning* phase takes a logical tree and generates one or more physical plans. This phase uses a combination of cost-based search as well as heuristics. Some implementation rules such as pushing operations from the logical plan into data sources that support predicate or projection push down (similar to Orca and Calcite), that are considered always beneficial to apply are also applied during the physical planning phase. Due to the separation of search into phases described above, the space of plans that are explored in a cost-based manner can be significantly smaller compared to Volcano and Cascades, potentially leading to loss in plan quality.

The final step of code generation is orthogonal to how a query optimizer generates the plan (as noted in Section 1), and in principle, could be applied to the physical plan generated by any query optimizer. In Catalyst, the *code generation* phase is performed at query execution time. It generates Java bytecode corresponding to the physical plan. For this purpose, Catalyst relies on a feature of the Scala language, quasiquotes, using which it programmatically constructs an abstract syntax tree (AST) in Scala. The Scala compiler then generates bytecode from the AST.

3.5 PostgreSQL

PostgreSQL is a widely used open-source database system. It is recognized for its extensibility with respect to user-defined types, user-defined functions, and indexing capabilities [8, 187]. This has enabled database engine developers to build extensions for richer data types beyond the native data types, such as spatial data, full text, and arrays. Developers have also leveraged extensibility to build new kinds of indexes for native data types, e.g., bitmap indexes, partial indexes. However, with respect to query optimization, unlike other query optimizers described in this section, PostgreSQL's query optimizer is not designed for extensibility with respect to transformation rules. Rather, the query optimizer follows an approach similar to System R's query optimizer described in Section 1.2. For completeness, we will briefly discuss the broad architecture of the PostgreSQL's optimizer.

Query optimization in PostgreSQL is split into three main stages: (1)*query flattening*, (2)*scan/join planning*, and (3)*post scan/join planning* [169]. We review each of these steps below, followed by a comparison with Volcano/Cascades.

Query flattening In this step, the optimizer performs a series of transformations to the parsed query tree such as replacing a reference to a view with the view's definition, eliminating an uncorrelated subquery by "pulling up" the relations referenced in subquery into the outer block of the query, simplifying constant expressions, representing a WHERE clause in a canonical form, etc. Similar to simplification rules in Volcano/Cascades (Section 2.4), these transformations are expected, but not guaranteed, to improve the quality of the final plan found by the optimizer. The following Query 6 illustrates an example rewriting performed by the optimizer.

For Query 6, the optimizer eliminates the subquery in the FROM clause. For the rewritten query, the optimizer has an opportunity to find an efficient plan joining the orders and lineitem tables. We note that such rewrites are only attempted when the subquery does not use aggregates, GROUP BY, or DISTINCT.[2]

[2]In Section 4.5, we discuss transformation rules for eliminating subqueries.

Query 6

```
SELECT o_orderkey, L.l_orderkey, L.l_linenumber
FROM orders,
    (SELECT l_orderkey, l_linenumber
    FROM lineitem
    WHERE l_shipmode = 'AIR') as L
WHERE orders.o_orderkey = L.l_orderkey

-- Query after rewriting

SELECT o_orderkey, l_orderkey, l_linenumber
FROM orders, lineitem
WHERE o_orderkey = l_orderkey AND l_shipmode = 'AIR'
```

Although flattening can help significantly in the above cases, its scope is limited. For example, flattening is not performed when the subquery is correlated, the query contains outer joins, or when the number of relations in the subquery exceeds a threshold. In these cases, the subquery must be optimized independently. This is achieved by recursively invoking query optimization on the subquery.

Scan/join planning In the scan/join planning phase, when the number of relations referenced in the query is below a predefined threshold, the optimizer uses bottom-up dynamic programming to produce the optimal join order similar to the approach in System R [178] (described in Section 1.2). When the number of relations in the query is above the threshold, the optimizer switches to genetic optimization (*geqo*) search strategy [171] for efficiency.

Post scan/join planning In the post scan/join planning phase, the optimizer determines the physical operators to use for any remaining logical operators in the query and produces the final plan. The operators handled in this phase include GROUP BY and aggregation, window functions, DISTINCT, ORDER BY, and LIMIT. Each logical operator is optimized in a cost-based manner by considering alternative physical operators and choosing the alternative with the lowest cost.

Comparison with Volcano/Cascades In PostgreSQL the separation of scan/join planning from query flattening and post scan/join phase

prevents the exploitation of important transformations such as the reordering of GROUP BY and joins, and decorrelation of correlated subqueries (see Sections 4.4 and 4.5). This results in a reduced search space of plans and therefore can result in missing out plans with lower cost. Furthermore, in PostgreSQL the transformations are applied in a fixed order. This can also result in missed opportunities for better plans. For example, in the post scan/join planning phase, the choice of physical operator for GROUP BY is determined before ORDER BY. However, if the ORDER BY delivers the ordering of rows required by the GROUP BY, then a potentially lower cost and lower memory footprint alternative (e.g., using Stream Aggregate rather than Hash Aggregate) might be missed. In contrast, the Volcano/Cascades frameworks apply rules (including enforcers) in a goal-driven manner so that many more potential alternatives can be considered.

3.6 Suggested Reading

Citation numbers below correspond to numbers in the References section.

[123] G. M. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives," *ACM SIGMOD Record*, vol. 17, no. 3, 1988, pp. 18–27

[166] H. Pirahesh *et al.*, "Extensible/Rule based Query Rewrite Optimization in Starburst," *ACM Sigmod Record*, vol. 21, no. 2, 1992, pp. 39–48

[183] M. A. Soliman *et al.*, "Orca: a Modular Query Optimizer Architecture for Big Data," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 337–348, 2014

[9] M. Armbrust *et al.*, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pp. 1383–1394, 2015

[15] E. Begoli *et al.*, "Apache Calcite: A Foundational Framework for Optimized Query Processing over Heterogeneous Data Sources," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 221–230, 2018

4

Key Transformations

In Section 2 we described a framework for an extensible query optimizer. However, an extensible query optimizer framework only gets its power when equipped with the right set of rules. While the application of a single rule can result in an expression with lower cost, it is the successive application of multiple rules driven by the search algorithm (Section 2) that realizes the full power of an extensible optimizer. In this section, we review a sample of the key logical transformation rules and implementation rules. We focus on a relatively small set of rules that are commonly used in practice. We discuss rules related to access path selection (Section 4.1), inner joins (Section 4.2) and outer joins (Section 4.3), group-by (Section 4.4), subqueries (Section 4.5), and a few *advanced* rules (Section 4.6) related to star and snowflake queries, sideways information passing, user-defined functions (UDFs), and materialized views. We also provide references to a few other important transformations not covered in this work.

As noted in Section 2.1, a rule is defined by the $CheckPattern$ and $Transform$ methods. $CheckPattern$ checks if the rule is applicable to the given expression. If it returns True, the $Transform$ method is called on the expression, which performs the actual transformation.

When describing the rules in this section, we use pseudocode for the $CheckPattern$ and $Transform$ methods as well as examples.

Table 4.1 summarizes the notations used in this section.

Table 4.1: Notation used in figures and pseudocode of rules

Notation	Definition
Notations in figures	
Join	Inner join
LOJ	Left outer join
ROJ	Right outer join
FOJ	Full outer join
LogOpAnd	Logical operator AND
Filter	Filter operator
Select	Project operator
Agg, GroupBy($Cols$)	Group by $Cols$ columns with aggregate function Agg
Get	Logical operator to access a table
Index($Cols_1$, $Cols_2$)	Index access with key columns $Cols_1$ and included columns ($Cols_2$)
Key Lookup	Phyiscal operator to retrieve the row by the key of a table
Hash Join	Physical operator Hash Join
Nested Loop Join	Physical operator Nested Loop Join
Index Scan	Physical operator Index Scan
Index Seek	Physical operator Index Seek
Notations in pseudocode	
expr	expression with logical and/or physical operators
expr.root	root operator of *expr*
expr.left	left child of *expr*, e.g., used in join, predicate filter
expr.right	right child of *expr*, e.g., used in join, predicate filter
expr.child(i)	i^{th} child of *expr*, e.g., used in predicate filter, aggregate
expr.pred	predicate filter(s) on *expr*, which is also an expression
expr.quantifier	quantifier on *expr* if any, e.g., used in predicate filter
expr.cols	set of columns in *expr*
expr.aggs	set of aggregates in *expr*
LogOpJoin(left, right, joinCond)	Logical operator Inner Join
LogOpSemiJoin(left, right, joinCond)	Logical operator Semi-join
LogOpGroupBy(expr, cols, aggs)	Logical operator Group By with groupby columns
LogOpLOJ(left, right, joinCond)	Logical operator Left Outer Join
LogOpROJ(left, right, joinCond)	Logical operator Right Outer Join
LogOpFOJ(left, right, joinCond)	Logical operator Full Outer Join
LogOpApply(left, right, joinCond)	Logical operator Apply
LogOpGet(expr)	Logical operator Get
LogOpSelect(expr, pred)	Logical operator Select with predicate filters
LogOpFilter(left, right, pred)	Logical operator Filter with a single predicate
LogOpProject(expr, columns)	Logical operator Project
LogOpAnd(left, right)	Logical operator AND, e.g., used in predicate filter
LogOpOr(left, right)	Logical operator OR, e.g., used in predicate filter
LogOpSubquery(expr)	Logical operator for a subquery
PhyOpIndexSeek(index)	Physical operator Index seek
PhyOpIndexScan(index)	Physical operator Index Scan
PhyOpHashJoin(left, right, pred)	Physical operator Hash Join
PhyOpStreamAggregate(expr, groupByCols, agg)	Physical operator Stream Aggregate

4.1 Access Path Transformations

A SQL query expresses a logical relational expression over a set of base relations. Typically, these base relations are tables stored in the database.[1] Access methods, such as indexes and heaps, which are persistent data structures, provide a mechanism for the database engine to access the data stored in base relations. In the discussion below we focus on B+-trees which are commonly used in most DBMSs. The physical operator used to access a heap is called Table Scan, whereas the Index Scan, Index Seek and Key Lookup operators are used to scan an index, seek an index, and perform a key lookup on a primary key index respectively. For a brief overview on access methods, we refer the readers to the Appendix. A more complete discussion is available in [120].

Access path selection focuses on choosing the access method(s) that minimize the cost of retrieving the requested data from the base relations [178]. Since the cost of accessing data using different access methods can vary widely, it is important that the transformation rules are able to generate multiple alternatives for retrieving data from the base relations for a given expression.

We illustrate some alternatives for access path selection using an example of a single-table query. Consider a table $S(id, a, b, c)$ with four columns, where id is the *primary key* of table S, and suppose the following B+-tree indexes exist on the table:

- Index I_{id} is a primary key *clustered* index where the *key* column of the index is the id column.
- I_a and I_b are single-column *non-clustered* indexes where the *key* columns are a and b respectively.
- $I_a(c, b)$ is a *non-clustered* index where the *key* column is a and the *include* columns are c and b.
- $I_b(c, a)$ is a *non-clustered* index where the *key* column is b and the *include* columns are c and a.

[1] In data lakes, base relations are stored as files in a format optimized for analytics, e.g, Parquet. In a federated database, a base relation may correspond to a SQL query against data stored in a DBMS.

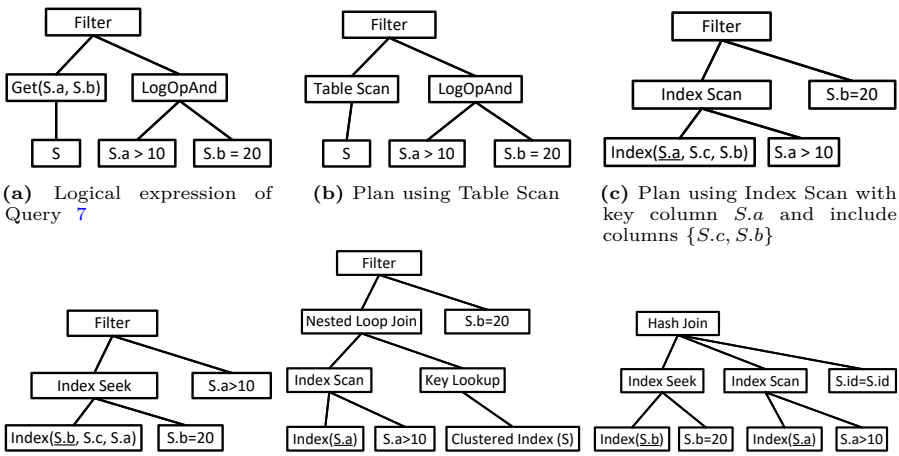
An *include* column of an index cannot be used to seek values, however, the value of each include column is available in the index, and hence can be used for projecting and filtering once the rows have been retrieved from the index.

Now, consider the following Query 7:

Query 7

```
SELECT S.a, S.b
FROM S
WHERE S.a > 10 AND S.b = 20
```

Query 7 retrieves the rows from table S that must satisfy the predicates on both S.a and S.b specified in the WHERE clause. Furthermore, each row returned must include the columns S.a and S.b. Figure 4.1 illustrates how different access methods can be used to answer Query 7. The logical query tree corresponding to Query 7 is shown in Figure 4.1a.



(a) Logical expression of Query 7

(b) Plan using Table Scan

(c) Plan using Index Scan with key column $S.a$ and include columns $\{S.c, S.b\}$

(d) Plan using Index Seek with key column $S.b$ and include columns $\{S.c, S.a\}$

(e) Plan using Index Scan with key column $S.a$ followed by Key Lookup of the clustered index

(f) Plan using index intersection of two indexes with key columns on $S.a$ and $S.b$ respectively

Figure 4.1: Plans using different access methods to execute Query 7

Plans using a single access method The query can be answered by using a Table Scan on table S followed by a Filter where the predicates in the WHERE clause are applied as shown in Figure 4.1b. Alternatively,

using an Index Scan on the index $I_a(c, b)$, we can retrieve the rows where $S.a > 10$. Since the column b is an include column, we can then apply a Filter operator with the predicate $S.b = 20$ on the retrieved rows to return the result of Query 7 (Figure 4.1c). Similarly, using an Index Seek on the index $I_b(c, a)$, we can retrieve the rows with $S.b = 20$ and then apply the Filter operator with the predicate $S.a > 10$ on the retrieved rows as column a is an include column of the index (Figure 4.1d).

Plans combining multiple access methods In some cases, we can retrieve rows from a single table by *combining* multiple access methods on the table. Consider the plan shown in Figure 4.1e. Here, we can use the index I_a to identify the rows satisfying the predicate $S.a > 10$. Then for each such row, we use a Key Lookup on the clustered index I_{id} to lookup the value of S.b for that row and then apply the Filter operator with the predicate $S.b = 20$. Finally, a plan using *index intersection* is also possible as shown in Figure 4.1f. Here, we first retrieve the qualifying rows from I_a satisfying the predicate $S.a > 10$ and the qualifying rows from I_b satisfying the predicate $S.b = 20$. We then intersect these two row sets by performing a Hash Join on the *id* column. The index intersection plan can be particularly beneficial when both predicates individually are not very selective, but their conjunction is selective, i.e., the predicates are anti-correlated.

The pseudocode and figure of the rule for index intersection are shown in Transformation 1 and Figure 4.2 respectively.[2] Since there can be multiple ways to intersect indexes for a given expression in the *CheckPattern* function in Transformation 1. For each such way, the transformation, i.e., *Transform* function in Transformation 1, will be invoked.

Note that index intersection and key lookup can also be used together resulting in even more complex access path strategies. Since the effectiveness of these access strategies depends on the selectivity of the predicates and the cost of retrieval, the optimizer needs to choose the best index strategy in a cost-based manner.

[2]We omit the details of this multi-step transformation here. In general, the index strategy of index intersection can introduce additional logical expressions, including projections and filters, because suitable post-processing is often required to deliver

Transformation 1 Intersect two indexes to access the required columns from a table. The outer side index is accessed using Index Scan, and the inner side index is accessed by Index Seek.

```
 1: function CheckPattern(expr)
 2:     result ← null
 3:     if expr.root = LogOpGet then
 4:         cols ← ExtractColumns(expr)
 5:         indexes ← GetIndexes(expr.root.table)
 6:         for index_1 ∈ indexes do
 7:             cols_1 ← GetColumns(index_1)                    ▷ outer side
 8:             for index_2 ∈ indexes, index_2 ≠ index_1 do     ▷ inner side
 9:                 cols_2 ← GetColumns(index_2)
10:                 if cols ⊆ cols_1 ∪ cols_2 then
11:                     if ∃seekCols ⊆ cols_1 and seekCols is a prefix of
    GetKeyColumns(index_2) then
12:                         result ← result ∪ {(index_1, index_2, seekCols)}
13:         return True, result
14:     else
15:         return False, result
16: function Transform(expr, index_1, index_2, seekCols)
17:     cols ← ExtractColumns(expr)
18:     cols_1 ← cols ∩ GetColumns(index_1) ∪ seekCols
19:     cols_2 ← ((cols \ cols_1) ∩ GetColumns(index_2)) ∪ seekCols
20:     left ← LogOpProject(LogOpGet(index_1), cols_1)
21:     right ← LogOpProject(LogOpGet(index_2), cols_2)
22:     joinCond ← ExtractJoinCond(left, right, expr)
23:     return LogOpJoin(left, right, joinCond)
```

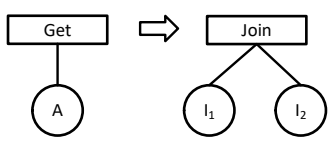


Figure 4.2: Index intersection

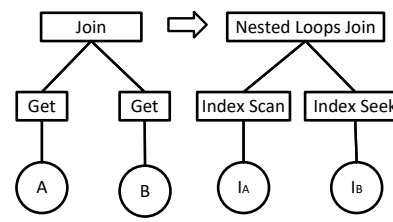


Figure 4.3: Join with Index Seek

Using indexes for joins So far our discussion has focused on using indexes for accessing data from a single table. Indexes can also be used in joins for seeking the data of the inner side based on the values from

the rows and columns of the requested data.

the outer side, i.e., an *outer reference*. In this case, the Join operator may be implemented using Nested Loops Join. The transformation is illustrated in Figure 4.3.[3] Note that the example of access methods with Key Lookup shown in Figure 4.1e can be considered as a special case of using indexes for joins, where the join is a *self join*. Here, for each row from the outer side I_a, the equi-join on $S.id$ is implicitly applied to the rows from the inner side by using the Key Lookup operator with the clustered index on S.

4.2 Inner Join Transformations

The *join* operator combines rows from two relations and outputs a new relation, and is widely used in practice. Depending on how the relations are joined together, ANSI-standard SQL:2003 specifies five types of logical join operators: inner join, left outer join, right outer join, full outer join, and cross join. Given a set of relations to join, one of the most important decisions that the query optimizer needs to make is join ordering, i.e., determining the cheapest join order from the space of all join orders (e.g., [178, 185]). Figure 1.4 in Section 1 shows two different join orders for the same query, one a linear sequence of joins, and the second a bushy plan. In Section 4.2 and 4.3 we focus mainly on transformation rules that allow the optimizer to enumerate the space of join orders for inner joins and outer joins respectively.

Inner join is the most commonly used join type, which outputs a tuple corresponding to a pair of rows from the joined relations if and only if that pair satisfies the join condition. Query 8 shows an example of inner join with the join condition specified in the WHERE clause. When the join condition is an equality condition, e.g., $A.z = B.z$ in the example Query 8, the join is referred to as an *equi-join*.

[3]We omit the details of this multi-step transformation here. A more accurate description will require the introduction of the *Apply* operator (see Section 4.5), which is a *logical* operator that invokes the inner side with parameterized input values, i.e., the values of the join columns of the rows from the outer side. Thus, the Join operator is first transformed into the Apply operator, and then the expression and its inputs are further transformed by the implementation rules in the search.

Query 8

```
SELETC A.x, B.y
FROM A INNER JOIN B
WHERE A.z = B.z
```

4.2.1 Join commutativity and associativity

We start with the logical transformation rules of join ordering for equi-joins [63], including:

- Commutativity: $A \bowtie B \Rightarrow B \bowtie A$
- Right associativity: $(A \bowtie B) \bowtie C \Rightarrow A \bowtie (B \bowtie C)$
- Left associativity: $A \bowtie (B \bowtie C) \Rightarrow (A \bowtie B) \bowtie C$

Transformation 2 shows the pseudocode for the join commutativity transformation rule and Figure 4.4 shows how the rule transforms the input expression (*expr*). To illustrate why join commutativity is useful, consider the transformed expression $B \bowtie A$. If there is an index on column $A.z$, then the plan where a Nested Loops Join is used with an Index Seek on $A.z$ becomes possible, but only after the join commutativity rule is invoked on the original expression.

Transformation 2 Join commutativity

```
1: function CheckPattern(expr)
2:     if expr.root = LogOpJoin then
3:         return True
4:     else
5:         return False
6: function Transform(expr)
7:     return LogOpJoin(expr.right, expr.left, expr.joinCond)
```

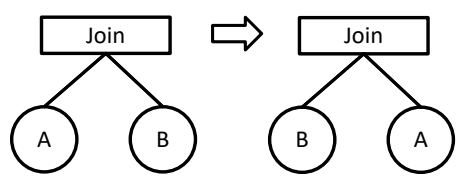


Figure 4.4: Join commutativity

The pseudocode for Join Right Associativity is shown in Transformation 3. To see why the Join Right Associativity rule can be useful,

Transformation 3 Join right associativity

```
1: function CheckPattern(expr)
2:     if expr.root = LogOpJoin and expr.left = LogOpJoin then
3:         return True
4:     else
5:         return False
6: function Transform(expr)
7:     newLeft ← expr.left.left
8:     joinCond ← ExtractJoinCond(expr.left.right, expr.right, expr)
9:     newRight ← LogOpJoin(expr.left.right, expr.right, joinCond)
10:    joinCond ← ExtractJoinCond(newLeft, newRight, expr)
11:    newExpr ← LogOpJoin(newLeft, newRight, joinCond)
12:    return newExpr
```

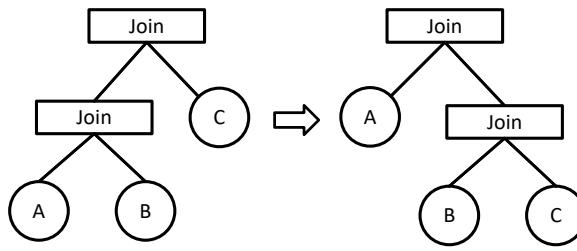


Figure 4.5: Join right associativity

consider the example plan on the right shown in Figure 4.5 after the rule is applied. Suppose C is a large table, e.g., a fact table in a data warehouse, and A and B are smaller tables, e.g., dimension tables. In this case, if both join operators are Hash Joins with table A and B on the build side respectively, then using a *single scan* of table C, we can probe the hash table for B and the resulting output rows can be used to probe the hash table for A. Such a plan can be efficient when there is sufficient memory to hold hash tables of A and B.

Enumerating join orders using commutativity and associativity rules in Volcano/Cascades Join orders of bushy trees can be enumerated by using right (or left) associativity and commutativity [100]. Consider enumerating the join orders of table R, S, T. Starting from the expression $R \bowtie S \bowtie T$, Figure 4.6 shows how all the 12 join orders are enumerated from top left to down right using right associativity and commutativity.

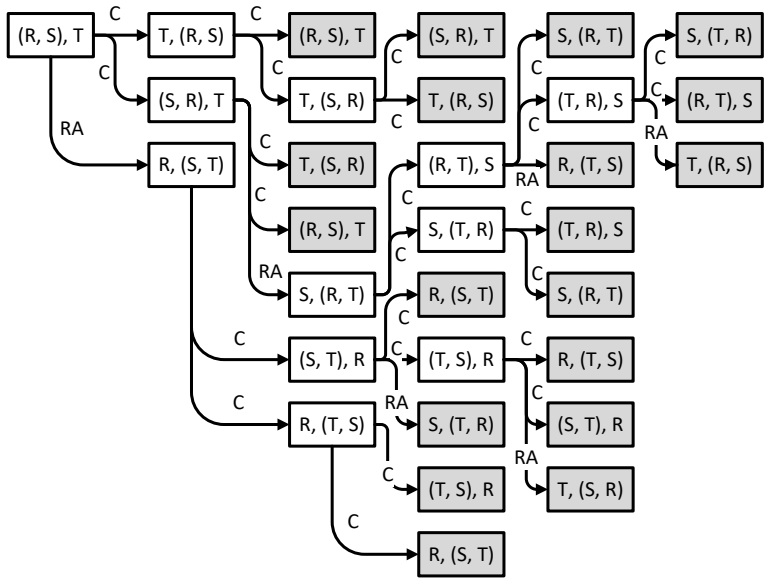


Figure 4.6: Explore join ordering with commutativity (C) and right associativity (RA). Duplicate expressions derived are marked in gray.

We also observe from the figure that many expressions are derived more than once. While these duplicate expressions are detected by the search algorithm in Volcano/Cascades due to memoization (as described in Section 2) and therefore will not be further transformed, even *deriving* these expressions incurs significant overhead in the search, especially when joining a large number of relations. We note that duplicated derivation can be avoided in some cases by applying the rules in a specific sequence [163]. This can be implemented in Volcano/Cascades by using the guidance mechanism as described in Section 2.

4.2.2 Push-down (pull-up) of filters below (above) join

Selection conditions, also referred to as filters, are common in SQL queries. Since a filter can reduce the number of rows to be processed by the operators above the filter, it is often beneficial to evaluate the filter early in a query plan, especially when the filter is selective. Consider the following example Query 9:

Query 9

```
SELECT COUNT(*)
FROM partsupp, part
WHERE ps_partkey = p_partkey
AND p_size > 10
AND ps_comment LIKE '%complaints%'
AND p_retailprice > 2 * ps_supplycost
```

Because the predicate $p\_size > 10$ only involves the column of *part* table, we can filter the rows from the *part* table with the predicate before joining them with *partsupp*. On the contrary, because $p\_retailprice > 2 * ps\_supplycost$ involves columns from both *part* and *partsupp* table, this predicate can only be evaluated after the join.

In general, if a filter only involves columns from one child expression of the join, then the filter can be pushed down to the child expression while preserving logical equivalence. Observe that after the push-down, it may be possible to use available access methods on the base table, e.g., an index, to efficiently retrieve the qualifying rows using a physical operator such as Index Scan or Index Seek. Since not all filters can be pushed down, the filters that are not pushed down need to be evaluated after the join. Note that it is possible that part of the filters are pushed down to the left child of the join, part of the filters are pushed to the right child, and the remaining filters are evaluated after the join. The transformation rule that pushes a predicate filter to the left child of the join is shown in Transformation 4 and illustrated in Figure 4.7.

Transformation 4 Push down the i^{th} predicate filter to the left child of the join

```
1: function CheckPattern(expr, i)
2:     if expr.root = LogOpJoin and expr.pred(i).cols ⊆ expr.left.cols then
3:         return True
4:     else
5:         return False
6: function Transform(expr, i)
7:     expr.left.pred.add(expr.pred(i))
8:     expr.pred.remove(i)
```

It is not always beneficial to push down a predicate filter below a join. For example, in Query 9, the evaluation of the $LIKE$ predicate,

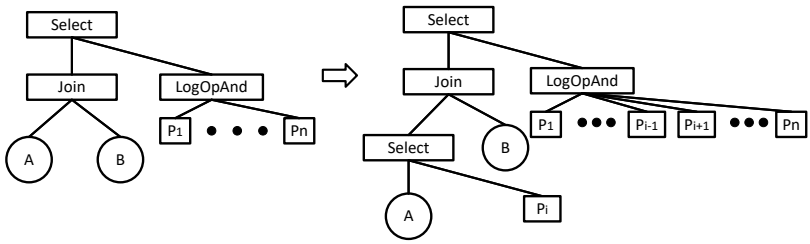


Figure 4.7: Push down the i^{th} predicate filter to the left child of the join.

i.e., $ps\_comment\ LIKE\ '\%complaints\%'$, can be expensive. If the join of *part* and *partsupp* is selective, i.e., it produces only a few rows, then it can be more efficient to evaluate the $LIKE$ predicate after the join. Therefore, the push-down of the predicate filter needs to be cost-based. Conversely, a filter on a base table or a child expression of a join can be also *pulled up* above the join if the filter is expensive and the join is selective.

4.2.3 Physical transformation rules of inner join

Inner joins can be implemented with a variety of join methods, the most common of which are Nested Loops Join, Hash Join, and Merge Join. Nested Loops Join is the only join method that applies to arbitrary inner joins, e.g., if the join predicate is a user-defined function. In contrast, Hash Join can be used only for equi-joins. Since Merge Join requires its inputs to be sorted on the join column(s), it can be effective when an explicit sort operation is not required on the results of one or both of its inputs. This can happen, for example, if the input is an Index Scan, where the index's key column is the join column.

Transformation 5 shows the pseudocode for the implementation rule of transforming a join into a Hash Join. Note that if the join predicate has non-equi join predicates, Hash Join can evaluate them as residual predicates after the join (line 9-10). However it must have at least one equi-join predicate to be applicable. For example, if the join predicate is $R.a = S.b$ AND $R.c = S.d$ AND $R.e > R.f$, the function $ExtractEquiJoinCond$ extracts the predicate $R.a = S.b$ AND $R.c = S.d$ and the residual predicate (in line 9) is $R.e > R.f$.

Transformation 5 Transform Join to Hash Join

```
1: function CheckPattern(expr)
2:     if expr.root = LogOpJoin and ExtractEquiJoinCond(expr) ≠ ∅ then
3:         return True
4:     else
5:         return False
6: function Transform(expr)
7:     joinPred ← ExtractJoinCond(expr)
8:     equiJoinPred ← ExtractEquiJoinCond(expr)
9:     resPred ← SubtractPred(joinPred, equiJoinPred)
10:    return PhyOpHashJoin(expr.left, expr.right, equiJoinPred, resPred)
```

Finally, we note that prior work has also proposed more specialized join algorithms that can more effectively exploit the properties of the join predicates. For example, consider an inner join of relations R and S that is a *band join*, i.e., where the join attribute of R falls within a specified range of the value in the join attribute of S. For example, the join predicate is of the form $R.a - c_1 \leq S.b \leq R.a + c_2$. Such joins occur in real-world queries when R.a and S.b represent event timestamps, and the query is to find all pairs of events that occurred within a specified interval from each other. Then, a new operator called the *partitioned band join* [56] can be used to implement the join efficiently. To add an implementation rule for the partitioned band join, the *CheckPattern* would need to include a check that the expression is a join, and that the join predicate conforms to the requirements of a band join.

4.3 Outer Join Transformations

Unlike an inner join, where the join result has only *matching* rows from the two joined relations, an outer join also preserves the non-matching rows from one or both of the relations in the join result depending on the choice of the specific outer join operator. Depending on which relation the rows are retained from, i.e., left, right, or both, there are three types of outer joins: left outer join ($\leftarrow$), right outer join ($\rightarrow$), and full outer join ($\leftrightarrow$). If a row from a relation is preserved in the outer join without a match, the *NULL* value is padded to the join result for each column from the other relation. Table 4.2 shows two tables *Student*

Table 4.2: Example of outer joins.

(a) Student table

Student Name	Department ID
Alice	1
Bob	2
James	3
Mary	-1

(b) Department table

Department ID	Department Name
1	Computer Science
2	Social Science
3	Mathematics
4	Business

(c) Left outer join of Student and Department

Student Name	Department ID	Department Name
Alice	1	Computer Science
Bob	2	Social Science
James	3	Mathematics
Mary	-1	NULL

(d) Right outer join of Student and Department

Student Name	Department ID	Department Name
Alice	1	Computer Science
Bob	2	Social Science
James	3	Mathematics
NULL	4	Business

(e) Full outer join of Student and Department

Student Name	Department ID	Department Name
Alice	1	Computer Science
Bob	2	Social Science
James	3	Mathematics
NULL	4	Business
Mary	-1	NULL

(f) Left outer join of Department and Student

Student Name	Department ID	Department Name
Alice	1	Computer Science
Bob	2	Social Science
James	3	Mathematics
NULL	4	Business

and *Department* and four examples of outer joins between these two tables.

As with inner joins, different join orders for outer joins can result in widely varying costs. Unfortunately, the transformation rules for join reordering of inner joins do not hold for outer joins in many cases. Intuitively, different outer join orders can retain different non-matching rows and the corresponding padded NULLs in the join result. For example, while an inner join is commutative, an outer join is not. The left outer join of *Student* and *Department* is not equivalent to the left outer join of *Department* and *Student* (Table 4.2c and Table 4.2f). In the former case, all rows from *Student* are preserved in the output whereas in the latter that is not the case.

4.3.1 Commutativity and associativity

Even though, unlike inner joins, commutativity and associativity properties do not hold for outer joins, certain forms of these transformations still apply. Intuitively, if the reordering of the outer joins preserves the same set of non-matching rows and the corresponding padded NULLs, the result of the outer joins stays the same. For example, it is possible to commute a left outer join into a right outer join as shown in Table 4.2d and Table 4.2f. The corresponding transformation rule is shown in Transformation 6 and illustrated in Figure 4.8.

Transformation 6 Join commutativity for left outer join

```
1: function CheckPattern(expr)
2:     if expr.root = LogOpLOJ then
3:         return True
4:     else
5:         return False
6: function Transform(expr)
7:     return LogOpROJ(expr.right, expr.left, expr.joinCond)
```

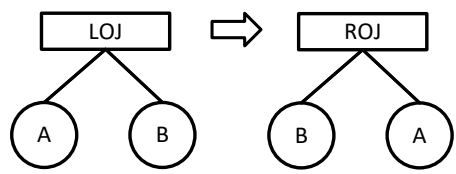


Figure 4.8: Join commutativity for left outer join

Similarly, associativity also holds in some cases. One example with Join Right Associativity for left outer joins is shown in Transformation 7 and Figure 4.9. Similar to inner join (as discussed in Section 4.2), the transformed expressions can lead to more efficient plans depending on factors such as cardinality of relations, availability of relevant indexes, and selectivity of join predicates.

Finally, we list a few examples of the commutativity and associativity rules for outer joins using the outer join notation presented earlier:

- Commutativity: $A \leftarrow B \Leftrightarrow B \rightarrow A$, $A \leftrightarrow B \Leftrightarrow B \leftrightarrow A$.
- Associativity: $(A \leftarrow B) \leftarrow C \Rightarrow A \leftarrow (B \leftarrow C)$, $(A \leftrightarrow B) \leftrightarrow C \Leftrightarrow A \leftrightarrow (B \leftrightarrow C)$

Transformation 7 Join right associativity for left outer joins

```
1:  function CheckPattern(expr)
2:      if expr.root = LogOpLOJ and expr.left = LogOpLOJ then
3:          return True
4:      else
5:          return False
6:  function Transform(expr)
7:      newLeft ← expr.left.left
8:      joinCond ← ExtractJoinCond(expr.left.right, expr.right, expr)
9:      newRight ← LogOpLOJ(expr.left.right, expr.right, joinCond)
10:     joinCond ← ExtractJoinCond(newLeft, newRight, expr)
11:     newExpr ← LogOpLOJ(newLeft, newRight, joinCond)
12:     return newExpr
```

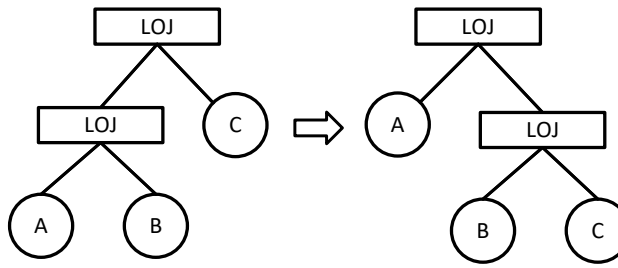


Figure 4.9: Join right associativity for left outer joins

For details on additional transformations for outer joins, we refer the readers to [70].

4.3.2 Redundancy rule

An important class of transformations for outer joins aims at identifying conditions where outer joins can be replaced by inner joins, which enables additional join orders that can be potentially more efficient. The *redundancy rule*, introduced in [68], is an example of such a rule. It uses the property of *null-rejecting* predicates. If a predicate p evaluates to false or undefined for NULL values, then p is said to be null-rejecting. For example, consider the predicate $S.a > 10$. Since it evaluates to false when $S.a$ is NULL, the predicate is null-rejecting. Intuitively, if the ancestor operator of an outer join contains a null-rejecting predicate on a column with padded NULLs introduced by the outer join, then

the outer join can be rewritten as an inner join without changing the result. For example, $\sigma_{S.a>10}(R \leftarrow_{R_k=S_k} S)$ can be rewritten as $\sigma_{S.a>10}(R \bowtie_{R_k=S_k} S)$ because the predicate $S.a > 10$ is null-rejecting on S, i.e., the outer join result with non-matching rows from R and the corresponding padded NULL values in $S.a$ will be filtered out by this predicate. This rule is shown in Transformation 8, and an example of transforming an expression using this rule is shown in Figure 4.10.

Transformation 8 Redundancy rule for outer joins

```
1: function CheckPattern(expr)
2:     if expr.root = LogOpLOJ or expr.root = LogOpROJ or expr.root =
   LogOpFOJ then
3:         if expr.parent.root = LogOpSelect and IsNullReject(expr.parent)
   then      ▷ Check if there exists null-rejecting predicates on the columns with
   padded NULL values introduced by the outer join
4:             return True
5:     return False
6: function Transform(expr)
7:     return LogOpJoin(expr.left, expr.right, expr.joinCond)
```

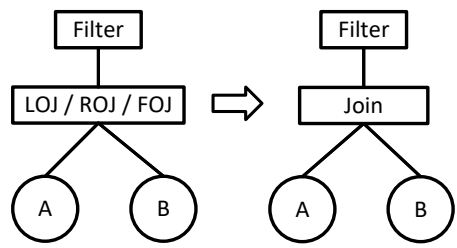


Figure 4.10: An example of transforming an expression using the redundancy rule for outer joins when the filter is null-rejecting.

For implementation rules, similar to inner joins, Hash Join, Merge Join, and Nested Loops Join can be used to implement outer joins if the join predicate only has equality predicates, whereas non-equi outer joins are implemented using Nested Loops Join.

4.4 Group-by and Join

The specification of a group-by operator has a set of *grouping columns* and a set of aggregation functions on one or more columns of its input relation. The group-by operator partitions the rows into groups of rows, one for each distinct combination of values of the grouping columns, and it invokes each of the aggregate functions specified in the operator for the rows in each group. Common aggregate functions include COUNT, SUM, AVG, MIN, MAX, and DISTINCT, and the SQL language has added many more analytic functions over the years. Table 4.3 shows two tables representing information of students and class schedules, and Query 10 below computes the total time of courses taken by each student.

Query 10

```
SELECT student, SUM(end - start) AS TotalCourseTime
FROM StudentClass AS S INNER JOIN ClassSchedule AS C
ON S.class = C.class
GROUP BY student
```

Table 4.3: Tables in the example of the group-by query

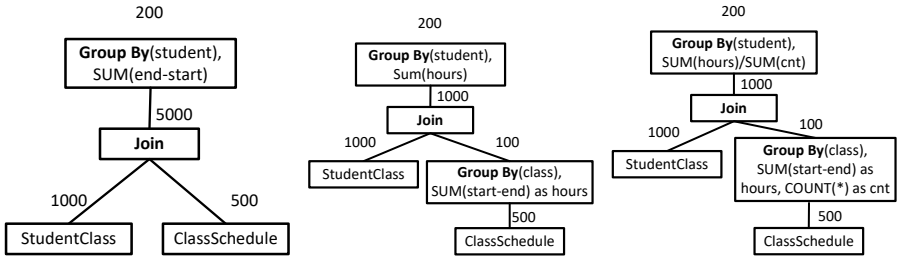
(a) StudentClass table (1000 rows)

student	class
Alice	Linear Algebra
Alice	Database
Bob	Operating System
Bob	Database

(b) ClassSchedule table (500 rows)

class	start	end
Database	Monday 9AM	Monday 10AM
Operating System	Tuesday 1PM	Tuesday 2PM
Database	Wednesday 9AM	Wednesday 10AM
Operating System	Thursday 1PM	Thursday 2PM

One option to evaluate the above query is to first join the table *StudentClass* and *ClassSchedule* and then execute the group-by operator (Figure 4.11a). However, this plan has a many-to-many join between *StudentClass* and *ClassSchedule*, which outputs 5000 rows. On the other hand, as shown in Figure 4.11b, if we first apply the group-by operator on the *ClassSchedule* table (and sum up the total course hours per week for each course) before joining it with the *StudentClass* table, we can avoid the expensive many-to-many join, which can lead to a more efficient plan.



(a) Plan P_1 joining the tables before group-by for calculating the total course hours taken by each student in Query 10

(b) Plan P_2 with partial aggregates for calculating the total course hours taken by each student in Query 10

(c) Plan P_3 with partial aggregates for calculating average hours per course taken by each student in Query 13

Figure 4.11: The plans for executing the group-by query. The number on the edge shows the output size for the corresponding operator.

As illustrated in the above example, because the result of a group-by operator only contains one row per group, evaluating a group-by operator can reduce the number of rows significantly when the number of distinct values in the group-by column(s) is much smaller than the number of rows in the relation. Therefore, in some cases, it is desirable to evaluate the group-by early in the plan to take advantage of data reduction. Such data reduction before evaluating expensive joins can lead to significant cost savings. Additionally, such transformation may potentially result in a different join order with lower cost.

In some cases, we can eliminate the group-by operator on top of the join and replace that with a group-by on either the right or left operand of the join. We refer to such transformations as *complete* group-by push-down. In other cases, as shown in Figure 4.11b, the plan benefits from partial data reduction by retaining a group-by operator on top of the join and adding another group-by operator atop one of the operands of the join, which is referred to as *partial* group-by push-down. Ensuring correctness of such transformations require careful thought. Below, we discuss each of these two classes of group-by push-down below joins.

4.4.1 Complete group-by push-down

The necessary and sufficient conditions for a group-by operator GBY on top of $R_1 \bowtie R_2$ to be eligible for a complete one-sided group-by

push-down to R_2 with a group-by operator GBY' atop R_2 are presented in [38, 69]:

1. All aggregate functions specified in the group-by operator GBY only uses columns from R_2
2. A primary key of R_1 is a subset of the grouping columns of GBY.
3. The grouping columns of the group-by operator GBY' is the union of grouping columns of GBY and the equi-join columns of R_2 in $R_1 \bowtie R_2$. The specification of the aggregations in GBY' is identical to those in GBY.

Without the first condition, no one-sided push-down of the group-by with elimination of the group-by above join is possible as columns of R_1 are not available to GBY'. Without the second condition, it will be possible for a single tuple of R_1 to be part of two or more different groups in the final output (and thus contribute to aggregated values in multiple distinct groups). If that were the case, multiple groups in GBY' will need to be coalesced and thus GBY cannot be eliminated. The last condition ensures that no tuples that would have been eliminated during the join $R_1 \bowtie R_2$ are used in producing aggregate results. As we will see later, the strategy of *fattening* the set of grouping columns leveraged in the third condition will be essential when we discuss partial group-by push-down later in this section. The complete group-by push-down transformation applies to *all aggregate functions, including user defined functions*. The pseudocode for the complete group-by push-down rule is shown in Transformation 9.

The opportunity for complete group-by push-down often occurs in the context of *primary-key-foreign-key* joins. Interestingly, a special case of complete group-by push-down is when the grouping columns of GBY contain the foreign key of R_1, where the third condition becomes redundant. In such cases, the specifications of GBY and GBY' are identical. This special case is referred to as *invariant grouping* [38]. For example, in the following Query 11:

Query 11

```
SELECT SUM(R.a)
FROM R INNER JOIN S
ON R.fk = S.k
GROUP BY S.k, R.fk
```

Transformation 9 Complete group-by push down. Push down a group-by operator below the right child of a join operator, where the join operator is the first child of the group-by operator.

```
1: function CheckColumnCondition(expr) ▷ Check the first two conditions of a complete group-by push-down
2:     join ← expr.child(0)
3:     cols ← GetGroupByColumns(expr)
4:     keyCols ← GetKeyColumns(join.left)
5:     rightCols ← GetColumns(join.right)
6:     joinCols ← ExtractJoinColumns(join.left, join.right, join)
7:     aggCols ← GetColumns(expr.aggs)
8:     if keyCols ⊆ cols and aggCols ⊆ rightCols then
9:         return True
10:    return False
11: function CheckPattern(expr)
12:    if expr.root = LogOpGroupBy and expr.child(0) = LogOpJoin then
13:        if CheckColumnCondition(expr) then
14:            return True
15:    return False
16: function Transform(expr)
17:    join ← expr.child(0)
18:    joinPred ← ExtractJoinCond(join.left, join.right, join)
19:    rightCols ← GetColumns(join.right)
20:    joinCols ← ExtractJoinColumns(join.left, join.right, join)
21:    cols ← GetGroupByColumns(expr) ∪ (joinCols ∩ rightCols) ▷ Fatten the group-by columns
22:    newGbExpr ← LogOpGroupBy(join.right, cols, expr.aggs)
23:    return LogOpJoin(join.left, newGbExpr, joinPred)
```

When $R.fk$ is a foreign key of R and $S.k$ is the primary key of S, the group-by operator can be pushed down to R and the aggregate SUM on $R.a$ is computed for each group prior to joining with S. Figure 4.12 shows the plan before and after the group-by push-down.

4.4.2 Partial group-by push-down

When all aggregate functions specified in the group-by operator GBY over $R_1 \bowtie R_2$ still only use columns from one of the operands R_2 of the join (i.e., the first condition for complete group-by push-down is satisfied but *not* the second), there may still be opportunities to *add* a group-by operator before the join to reduce the cardinality of the input of the join.

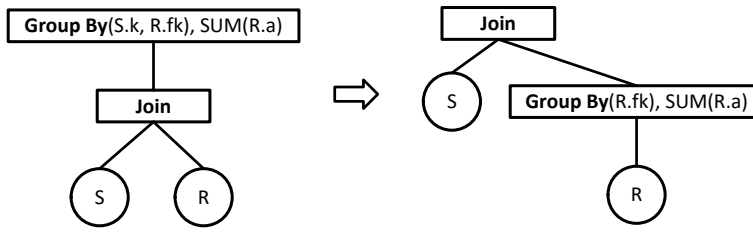


Figure 4.12: Push down group-by below join for Query 11.

The *partial aggregates* (also referred to as *local* aggregates) from the added group-by operator before the join are combined to calculate the final *global aggregates*. However, for this to be possible, each aggregate function *agg* specified in GBY over $R_1 \bowtie R_2$ must satisfy *distributive properties*. Specifically, we say that the aggregation function *agg* satisfies the *simple distributive property* if the following holds: $agg(S \cup S') = agg(agg(S), agg(S'))$. Examples of aggregate functions that satisfy this property include MIN, MAX, DISTINCT, and SUM. In such cases, the partial group-by push-down transformation is referred to as *simple coalescing grouping*. The transformation leaves GBY over $R_1 \bowtie R_2$ unchanged. The additional group-by operator GBY' introduced atop R_2 has the same specification for the aggregates but its grouping columns will be the union of the join columns of R_2 in $R_1 \bowtie R_2$ and the subset of grouping columns of GBY that are from R_2. The last condition ensures, as in condition (3) of complete group-by push-down that aggregations computed in GBY' do not erroneously include contributions from R_2 tuples that would be eliminated in the output of $R_1 \bowtie R_2$. Our earlier group-by query shown in Figure 4.11b is an example of simple coalescing grouping. We now discuss another example of partial push-down of group-by below join using simple coalescing grouping illustrated by Query 12:

Query 12

```
SELECT SUM(R.a)
FROM R INNER JOIN S
ON R.fk = S.k
GROUP BY S.b
```

Note that because the key of S is not part of the group-by columns, we cannot apply the complete group-by push-down as it violates the conditions described in Section 4.4.1. Instead, we can use the partial push-down of group-by (simple coalescing grouping) by computing a partial aggregate of $SUM(R.a)$ on R and then aggregate on the result of the join as shown in Figure 4.13. The pseudocode for the rule is shown in Transformation 10.

Transformation 10 Push down partial aggregates below the right child of a join for SUM aggregate function, where the join is the first child of the partial aggregates.

```
1:  function CheckPattern(expr)
2:      if expr.root = LogOpGroupBy and expr.child(0) = LogOpJoin then
3:          aggCols ← GetColumns(expr.aggs)
4:          cols ← GetColumns(expr.child(0).right)
5:          if aggCols ⊆ cols then
6:              for agg ∈ expr.aggs do
7:                  if agg is not SUM then
8:                      return False
9:          else
10:             return False
11:         return True
12:     return False
13: function Transform(expr)
14:     gbCols ← GetGroupByColumns(expr)
15:     join ← expr.child(0)
16:     joinCols ← ExtractJoinColumns(join.left, join.right, join)
17:     partialGbCols ← gbCols ∪ (joinCols ∩ GetColumns(join.right))
18:     globalAggs ← expr.aggs
19:     for agg ∈ expr.aggs do
20:         newAggCol ← GetNewColumnName(agg)
21:         globalAggs.Remove(agg)
22:         globalAggs.Add(SUM(newAggCol))
23:     newPartialGb ← LogOpGroupBy(join.right, partialGbCols, expr.aggs)
24:     joinPred ← ExtractJoinCond(join.left, join.right, join)
25:     newJoin ← LogOpJoin(join.left, newPartialGb, joinPred)
26:     newGlobalGb ← LogOpGroupBy(newJoin, gbCols, globalAggs)
27:     return newGlobalGb
```

A more *general version of the distributive property* for an aggregate function agg opens the door for more opportunities for partial group-by push-down. The required property may be stated as:

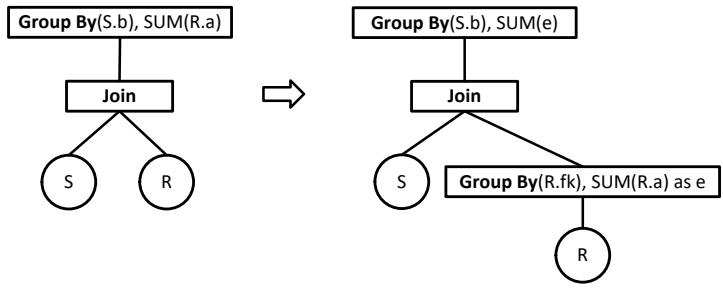


Figure 4.13: Push down partial aggregates below join.

$agg(S \cup S') = f(agg'(S), agg'(S'), count(S), count(S'))$, where agg' is an auxiliary aggregate function that is not necessarily the given aggregate function in the query, and f combines the partial aggregates into the final aggregate while leveraging the additional aggregate *count*. For example, the function AVG satisfies this property, where:

$$AVG(S \cup S') = \frac{SUM(S) + SUM(S')}{COUNT(S) + COUNT(S')}$$

We refer to the above case of group-by push-down as *generalized coalescing grouping*. For this partial group-by push-down transformation, the additional local group-by operator GBY' introduced above R_2 (but prior to the join) has the same set of group-by columns as in the case of simple coalescing grouping, but each aggregation function agg is replaced by agg'. In addition, an extra aggregation function *count* is added to the list of aggregation functions agg'. As in the case of simple coalescing grouping, the grouping columns of GBY' will be the union of the join columns of R_2 in $R_1 \bowtie R_2$ and the subset of grouping columns of GBY that are from R_2. Unlike simple coalescing grouping where the specification of the GBY operator after $R_1 \bowtie R_2$ was unchanged, in the case of generalized coalescing grouping, the aggregation functions agg are changed to agg' corresponding to the computation needs of the global aggregate (see the example of AVG above).

To illustrate the generalized coalescing grouping, we modify Query 10 to aggregate the *average* hours per course taken by each student instead of the total hours as the following Query 13:

Query 13
```
SELECT student, AVG(end - start) AS AvgCourseTime
FROM StudentClass AS S INNER JOIN ClassSchedule AS C
ON S.class = C.class
GROUP BY student
```

In this case, as shown in Figure 4.11c, we can first calculate the total number of hours (*hours*) and the number of counts (*cnt*) per course when aggregating on the *ClassSchedule* table, and then join with the *StudentClass* table to produce the final global aggregate. Observe how the aggregation functions have been modified to support the computation of the local and global aggregates.

Further generalization of the class of permissible aggregate functions are possible, e.g., multiple auxiliary aggregate functions that are not part of the original aggregate may be used to compute global aggregates, e.g., $agg(S \cup S') = f(agg_1(S), agg_1(S'), \ldots, agg_n(S), agg_n(S'))$. An example of this is the standard deviation function:

$$STD(S \cup S') = \sqrt{\frac{SUM(S^2) + SUM(S'^2)}{N} - (\frac{SUM(S) + SUM(S')}{N})^2}$$

where $N = COUNT(S) + COUNT(S')$.

As an interesting observation, we discuss how generalized coalescing grouping can be modified for partial group-by push-down for arbitrary aggregate function that may not satisfy any of the distributive properties above. In such cases, data reduction is still possible through the use of generalized coalescing grouping. Specifically, we add all columns over which such arbitrary aggregate functions are defined and treat them as additional grouping columns for the query. We then compute the only local aggregate of *count* for the additional group-by operator that is introduced below the join and thus still constitutes data reduction. As a result of this approach, all columns that these aggregate functions are defined are preserved and available to the global aggregate. The global aggregate computation takes into account the local *count* to reconstruct the accurate value of the final aggregate.

The complete and partial push-down of group-by operators below join can be integrated into an extensible optimizer similar to other

Transformation 11 Transform Group-by Aggregate to Stream Aggregate

```
1:  function CheckPattern(expr)
2:      if expr.root = LogOpGroupBy then
3:          groupByCols ← ExtractGroupByColumns(expr)
4:          sortCols ← ExtractSortColumns(expr)
5:              ▷ Check if the prefix of the sort order is the set of group-by columns
6:          if |groupByCols| > |sortCols| then
7:              return False
8:          prefixCols ← GetPrefixCols(sortCols, |groupByCols|)
9:          if prefixCols ⊆ groupByCols then
10:             return True
11:         return False
12: function Transform(expr)
13:     groupByCols ← ExtractGroupByCols(expr)
14:     agg ← ExtractAgg(expr)
15:     return PhyOpStreamAggregate(expr.child(0), groupByCols, agg)
```

transformation rules. It is also possible to push-down group-by below an outer-join [69]. Note that the transformations of group-by push-down below joins and outer-joins may not always be beneficial. If the join is very selective and the aggregate function is expensive, then it may be preferable to first perform the join before calculating the aggregates. Indeed, the "reverse" transformation of *pulling up* the group-by above joins has also been studied [198].

4.4.3 Physical transformation rules of group-by

A Group-By operator is implemented either by a Stream Aggregate or a Hash Aggregate operator. We observe that the Stream Aggregate operator requires that its input is sorted, and that the group-by columns are a prefix of the columns on which the input is sorted. The pseudocode of the implementation rule that transforms a Group-By to a Stream Aggregate is shown in Transformation 11. The *ExtractSortColumns* retrieves the sort columns from the physical property associated with the expression in the memo. The Stream Aggregate operator has $O(1)$ space complexity, since it only needs to incrementally maintain the aggregates associated with the *current* group. It can output the result once all rows of the current group have been consumed, and free up

the memory used for computing the aggregates. In contrast to Stream Aggregate, the Hash Aggregate operator cannot take advantage of the sortedness property of its input. Therefore, its memory consumption is proportional to the number of groups, i.e., the number of distinct values in its input. Furthermore, it is a *blocking* operator since it can only produce the output after it has consumed all input rows. However, unlike Stream Aggregate, Hash Aggregate is applicable in all cases, even when the input is not sorted.

4.5 Decorrelation

A subquery is a query that appears inside another query statement. Subqueries are also referred to as *nested* subqueries. Microsoft SQL Server allows up to 32 levels of nesting. Below, we provide two examples of nested subqueries.

If the nested subquery shares a variable with the outer query block, then it is called a *correlated subquery*. For example, Query 14 is *not* a correlated subquery as the inner subquery shares no variable with the outer query block. In contrast, Query 15 is a correlated subquery as subquery shares the variable *p_partkey* with the outer query block.

Query 14

```
SELECT o_orderkey
FROM orders, lineitem
WHERE o_orderkey = l_orderkey AND l_quantity > 10
AND l_suppkey IN (
    SELECT s_suppkey
    FROM supplier
    WHERE s_name IN ('Alice', 'Bob'))
```

Query 15

```
SELECT p_partkey
FROM part
WHERE p_size < 10 AND 100 > (
    SELECT SUM(l_quantity)
    FROM lineitem
    WHERE l_partkey = p_partkey)
```

Nested subqueries add to programming convenience even though they do not endow SQL with any more expressive power. However,

nested subqueries present new challenges for efficient query execution. This is because for correlated subqueries, the subquery may need to be executed for every tuple in the outer relation. For example, for Query 15, the inner subquery needs to be executed for each value of *p_partkey* from the outer query block. Such nested loop style of execution is expensive, and prohibitively so for distributed data platforms as noted in the System R* project [126]. Fortunately, modern query optimizers are able to *flatten* (or *unnest*) such queries while preserving semantic equivalence. This process of flattening nested subqueries is referred to as *decorrelation*. Decorrelation makes it possible for a large class of complex queries to be executed efficiently, including on distributed data platforms.

For simplicity of exposition, we will discuss decorrelation for queries where the subquery occurs in the WHERE clause of the outer query. For this class of queries, several variants are possible:

1. WHERE scalar_expression [NOT] IN (subquery)
2. WHERE scalar_expression comparison_operator [ANY | ALL] (subquery)
3. WHERE [NOT] EXISTS (subquery)

The result of the subquery may be a scalar value or a single column relation (e.g., if the comparison operator is IN and the subquery returns a set of rows). In case neither ANY nor ALL is used in (2), then the subquery must produce a single-column table with either zero or one row – a runtime error occurs if the subquery returns more than one row. Query 14 is an example of (1), with the scalar expression *l_suppkey*. Query 15 is an example of (2), where the result of the subquery is a scalar value $SUM(l\_quantity)$, the scalar_expression is 100 and the comparison_operator is $>$.

Prior work has identified several important cases where the query can be decorrelated [16, 52, 62, 153, 180]. We begin by presenting examples of a few transformations that remove nested subqueries. Next, we discuss what extensions to our algebraic framework for query optimization are needed to represent subqueries. We end with an example of a transformation that does not directly decorrelate the queries but helps reduce the cost of executing the nested subqueries.

4.5.1 Nested subqueries without correlated variables

The simplest case for a nested subquery is one where the subquery has no reference to the tuple variables from the outer block, i.e., the subquery has no correlated variables. A straightforward strategy to execute such a query is to generate a query plan by optimizing each block of the query separately. For example, Query 14 finds the orders in a TPC-H database that purchase more than 10 of the same item from a given set of suppliers (i.e., {'Alice', 'Bob'}). In this query, we have a predicate that for each supplier key in the join of *orders* and *lineitem* tables, checks if that supplier key belongs to the set of supplier keys of {'Alice', 'Bob'}. The main block of the query is a join query on the *orders* and *lineitem* tables. The subquery is a selection on the *supplier* table without correlated variables.

For such queries, we can optimize the outer query block independently from the inner query block. Moreover, the inner subquery needs to be executed only once [178]. Once the inner subquery is executed, its results (ideally an in-memory list) are used for evaluating the predicate of the outer block, e.g., *l_suppkey* IN <list>.

Despite the ease of executing nested queries without correlated variables, the potential benefit of unnesting such queries remains compelling. We now describe a logical transformation that is possible for queries with the IN predicate between the outer query block and the subquery. As an example, Query 14 can be transformed in the following unnested form without changing its semantics as *s_suppkey* is a key of the *supplier* table [109] as shown in Query 16:

Query 16

```
SELECT o_orderkey
FROM orders, lineitem, supplier
WHERE o_orderkey = l_orderkey AND l_quantity > 10
AND l_suppkey = s_suppkey AND s_name IN ('Alice', 'Bob')
```

In the above query, we have transformed the lookup in the list (*l_suppkey* IN <list>) with an inner join (see Section 4.6.2) between *lineitem* and *supplier* using the predicate $l\_suppkey = s\_suppkey$. Note that this is effectively a semi-join (a special case of inner join) as the rest of the query only needs attributes from *lineitem* and none from *supplier*, and

s_suppkey is a key of the *supplier* table. Moreover, such a transformation potentially unlocks the ability to reorder the semi-join with other joins in the query and compare the relative costs of these plans.

4.5.2 Nested subqueries with correlated variables

For *correlated* subqueries, flattening while preserving semantic equivalence needs more care. For a comprehensive discussion of transformations that benefit correlated subqueries, we refer the reader to [153, 180]. In this subsection, we discuss the widely used decorrelation transformation applicable to queries with scalar aggregates that was originally proposed in [52].

Subqueries returning a scalar aggregate occur commonly in practice. A query that exemplifies occurrence of such subqueries is Query 15: It finds the parts whose size is below 10 and whose total quantity ordered is less than 100. In this query, the subquery on *lineitem* refers to the value of *p_partkey* from the outer block. The nested subquery in the above example returns a *scalar aggregate* as it is not accompanied with a GROUP BY clause. The SQL semantics of scalar aggregates over a relation without GROUP BY (such as the subquery in the example) is that the aggregate always returns *exactly one row*. If the number of rows on which the scalar aggregate is computed is zero (i.e., if a relation is empty), then the value of the scalar is 0 for COUNT, and NULL for aggregates such as SUM or AVG [134]. In addition to the the subquery in Query 15 returning a scalar aggregate $SUM(l\_quantity)$, the outer reference in the nested subquery, $p\_partkey$, is a key of the outer relation.

In the *tuple substitution* (or *nested loop*) method of execution, for each tuple from *part*, we will calculate the aggregate on *l_quantity* for the tuples that match *p_partkey* in the *lineitem* table. If the *lineitem* table does not provide an efficient way to access the data by *l_partkey*, e.g., via an index, each invocation can require scanning all the rows in the *lineitem* table, thereby making the query expensive. Even if the *lineitem* table contains a suitable index but if there are many parts qualifying the restriction on *p_size*, then repeatedly looking up the index for each qualifying part could be expensive as we will be doing

too many index seeks. Therefore we ask ourselves what other execution plans are semantically equivalent to the above tuple substitution model of execution for the inner subquery that the optimizer should consider.

An alternative way to execute the query will be to convert this to a join query with GROUP BY as shown in Query 17. Intuitively, the transformed query collects for each *p_partkey* all matching rows in *lineitem* through the equi-join, aggregates the matching rows by grouping on *p_partkey* to compute SUM(l_quantity), and finally filters rows based on the value returned by the aggregation. Once the aggregate SUM(l_quantity) is available for each group, we select groups with SUM(l_quantity) < 100 using the HAVING clause. A corner case that we must also consider is when for a given *p_partkey* there is no matching tuple in *lineitem*, i.e., the equi-join condition in Query 17 fails. In such cases, there will be no rows for that *p_partkey* in the modified query. In the original Query 15, when *p_partkey* does not match any tuple in *lineitem* table, per the SQL semantics, the result of the inner subquery will be a single row with the value of SUM aggregate as NULL. In such a case, the WHERE clause fails and the original query with the "tuple substitution" mode of execution also will not output any tuple corresponding to that value of *p_partkey*.

Query 17

```
SELECT p_partkey
FROM part INNER JOIN lineitem on p_partkey = l_partkey
WHERE p_size < 10
GROUP BY p_partkey
HAVING 100 > SUM(l_quantity)
```

Let us now consider another query that is identical to the previous query except that we replace the aggregate function SUM with COUNT as shown in Query 18:

Query 18

```
SELECT p_partkey
FROM part
WHERE p_size < 10 AND 100 >
      (SELECT COUNT(*)
        FROM lineitem
        WHERE l_partkey = p_partkey)
```

However, that makes a significant difference! This is because when a relation is empty (i.e., has no tuples), the scalar aggregate COUNT emits the value 0 instead of NULL unlike the cases for SUM and AVG. Thus, for Query 18 when for a specific *p_partkey*, there are no matching *l_partkey*, COUNT returns zero and thus satisfies the predicate that connects the subquery to the outer block. However, if we were to generate a transformation similar to what was used in Query 17, we will miss in its output tuples from *part* for which the subquery yielded empty results as they would fail the INNER JOIN condition. However, if we were to replace the INNER JOIN with the LEFT OUTER JOIN, all such missing tuples from *part* will be retained in the output. Thus, Query 19 is the correctly rewritten query with LEFT OUTER JOIN:

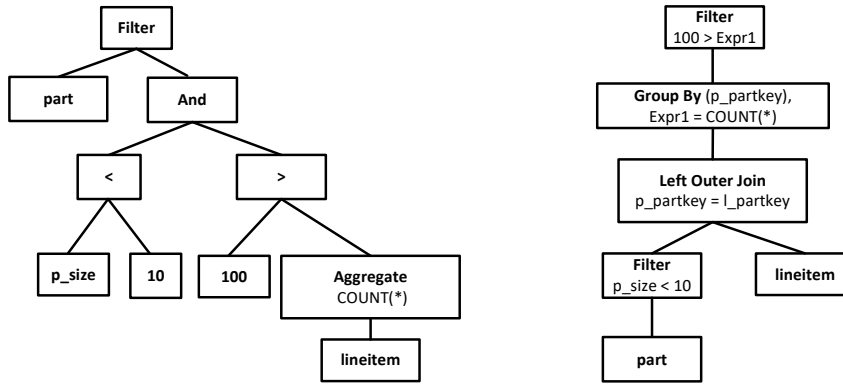
Query 19

```
SELECT p_partkey
FROM part LEFT OUTER JOIN lineitem ON p_partkey = l_partkey
WHERE p_size < 10
GROUP BY p_partkey
HAVING 100 > COUNT(*)
```

In our example Query 18, the logical expressions before and after applying this transformation are shown in Figure 4.14a and Figure 4.14b respectively. Note that the Boolean-valued subqueries, e.g., those with EXISTS, NOT EXISTS, can be rewritten as a subquery with a scalar COUNT aggregate [69]. Once such a rewrite is performed, the decorrelation transformation discussed above for scalar aggregate subqueries can be applied to them as well.

Our discussion of Query 18 illustrates a logical transformation that is widely used for decorrelation when the correlated subquery returns a scalar aggregate and the reference to the outer query block in the subquery is a key of the outer relation. This transformation results in a single block query with a left outer join (LOJ) [52]. Transformation 12 sketches the pseudocode of the corresponding transformation. The pseudocode is presented for the simplifying case when there is a single predicate in the outer block.

Finally, note that after decorrelation, the optimizer can subsequently apply other transformation rules for outer join and group-by as described in Section 4.3 and Section 4.4, which can potentially result in even more



(a) Original logical expression with the correlated subquery

(b) Transformed single block logical expression with Left Outer Join

Figure 4.14: Transform Query 18 by replacing the subquery with Left Outer Join followed by Group By and Filter.

Transformation 12 Transform query with a *single* predicate filter containing a subquery that returns a scalar aggregate

```
1: function CheckPattern(expr)
2:     cols ← ExtractOuterReference(expr.pred.right)
3:     if expr.root = LogOpFilter and expr.pred.right = LogOpSubquery and
       IsScalarAggregate(expr.pred.right) and IsKey(cols) then   ▷ The subquery
       must return a scalar aggregate
4:         return True
5:     else
6:         return False
7: function Transform(expr)
8:     joinCols ← ExtractColumns(expr.pred)
9:     groupByCols ← ExtractOuterReference(expr.pred.right)
10:    agg ← ExtractAggregate(expr.right)
11:    joinPred ← ExtractJoinPred(expr.left, expr.right.child(0), expr)
12:    newJoin ← LogOpLOJ(expr.left, expr.right.child(0), joinPred)
13:    newGroupBy ← LogOpGroupBy(newJoin, groupByCols, agg)
14:    return LogOpFilter(expr.pred.left, newGroupBy, expr.pred.root)
```

efficient plans. For instance, direct application of the transformation above rewrites Query 15 using the left outer join (LOJ). However, as noted above, the SUM scalar aggregate function returns NULL when there are no matching rows in the subquery. For example, the predicate $100 > SUM(l\_quantity)$ returns FALSE. Therefore, for the scalar

aggregate function SUM, the LOJ can be further transformed into an inner join (see Section 4.3.2), resulting in Query 17. Once the LOJ is transformed into an inner join, additional transformations such as join reordering among relations in the query would be possible.

4.5.3 Representing subqueries algebraically using Apply

The Apply meta-operator [69] provides an algebraic approach to represent computation of correlated subqueries. A different algebraic framework for decorrelation was proposed in [153] through the use of Dependent Joins. These two frameworks share significant conceptual similarity and allow us to reason about decorrelation transformations. A comprehensive exposition of these algebraic frameworks and the set of logical transformation rules that enable decorrelation are beyond the scope of this subsection. Instead, we provide a brief introduction to the Apply meta-operator and a few examples of transformation rules for decorrelation.

A correlated subquery may be viewed algebraically as a *parameterized relational expression (PRE)* where the parameters (i.e., correlated variables) are provided through the binding from the outer query block. For example, in Query 15, the inner subquery is a PRE, parameterized by $p\_partkey$. A degenerate case of a subquery is Query 14 where the inner subquery is not dependent on any parameter binding from the outer block.

We now consider how we can algebraize the evaluation of queries involving a subquery such as $PRE(p\_partkey)$. One way to represent the evaluation of queries with nested subqueries is to create a new (derived) column that collects together the results of evaluating the subquery for every possible binding of the outer query block. In our example Query 15, this means we create a new column Sum_of_Parts of values that for every binding for $p\_partkey$ from the outer query records the result of the subquery (the aggregated total quantity of the parts ordered). Once we have defined this new column Sum_of_Parts, the outer query can complete its evaluation by joining with the relation so produced consisting of columns $(l\_partkey, Sum\_of\_Parts)$ on $p\_partkey = l\_partkey$. Thus, the predicate between the correlated

variable $p\_partkey$ and the subquery in the given query is now turned into a selection on the column $(100 > Sum\_of\_Parts)$. Therefore, the outer query no longer requires correlated subqueries to be executed tuple at a time.

The *Apply* meta-operator algebraically represents the computation outlined above. Formally, it takes an outer relation R and a $PRE(Y)$ with the variables Y as its parameters for which bindings are provided by the outer relation R. It takes a relational operator (Op) as part of its specification as well. We denote the Apply construct compactly as A^{Op}. It iterates over every tuple t in R, binding the parameter variables Y in PRE using t. It then produces tuples that result from applying the relational operator Op for every binding of the outer to the output of $PRE(Y)$. As an example, when Op is the Cartesian Product $\times$, then

$$R\ A^{\times}\ PRE(Y) = \bigcup_{t \in R} (\{t\}\ \times\ PRE(t.Y))$$

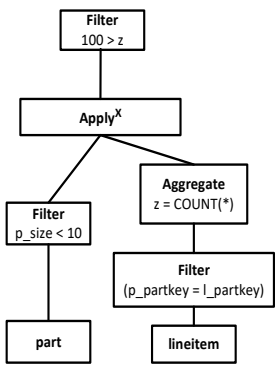
If instead of the cross product, the left outer-join (LOJ) variant of Apply (A^{LOJ}) is used, then the above expression changes to:

$$R\ A^{LOJ}\ PRE(Y) = \bigcup_{t \in R} (\{t\}\ LOJ\ PRE(t.Y))$$

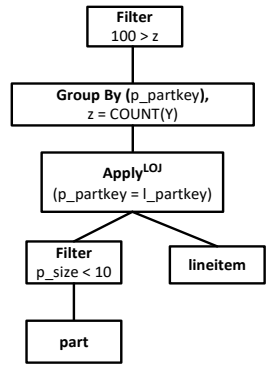
In this case, any non-matching tuple from R is passed to the output, padded with NULLs. Note that A^{Op} directly maps to the tuple substitution semantics of correlated subqueries mentioned in Section 4.5. As an example, Figure 4.15a shows the representation of Query 18 using Apply.

Given a query with the nested subquery, the following steps need to be taken:

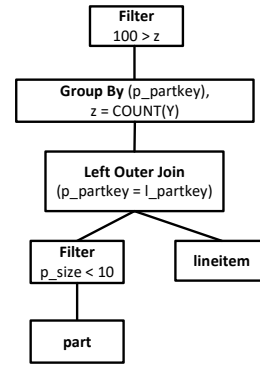
1. Represent the subquery evaluation using Apply A^X with cross product on the PRE corresponding to the subquery.
2. Use one of the transformation rules for Apply. A few examples of such transformations are described below.
3. Iterate on (2). If at any point, as a result of application of transformation rules, the PRE operand of Apply has no correlated variables from the outer relation of *Apply*, then the the instance of Apply may be removed and decorrelation is achieved.



(a) Plan P_1 with $Apply^X$



(b) Transformed plan P_2 with $Apply^{LOJ}$ operator. Y is the derived column created by Apply.



(c) Transformed plan P_3 with Left Outer Join removing *Apply*. Y is the column output by Left Outer Join.

Figure 4.15: Transform Query 18 by successively applying the transformation rules shown in Eq. 4.2 and Eq. 4.1.

We present a few of the transformation rules for Apply below. We use Exp, $Exp1$, and $Exp2$ to denote PREs. In the following, Op refers to a relational operator although for the rest of this subsection we limit ourselves to the operator being either a cross product ($\times$) or left outer join (LOJ). Selections and join conditions are denoted by s. A Scalar Aggregate is denoted by $SAgg$. Its single parameter is the aggregation function with the specification of the column expressions. For simplicity, we assume that the column expression is a single column. The attributes that form a key of a relation S is denoted by $Key(S)$. A group-by operator has two parameters. The first is the set of group-by columns, and the second parameter consists of aggregation functions along with the specification of columns over which the aggregation functions are defined.

We have the following transformation rule

$$R\ A^{Op}(\sigma_s Exp) = R\ Op_s\ Exp \tag{4.1}$$

if Exp has no correlated parameters that derive its binding from R.

We have the following transformation rule

$$R\ A^{X}\ (SAgg_w\ Exp) = GroupBy_{(Key(R),SAgg_w)}(R\ A^{LOJ}\ Exp) \tag{4.2}$$

if R has its key columns included. In other words, R has no duplicates. An exception to the above rule is for the case when w is $COUNT(*)$. In that case, Rule 4.2 still applies with the modification that we will need to replace w with w' on the right side of the equation where $w' = Count(Y)$, i.e., Y is the derived column that results from the application of A^X.

Similarly, we have the following transformation rule

$$R\ A^X\ (Exp1 \bigcup Exp2) = R\ A^X\ Exp1 \bigcup R\ A^X\ Exp2 \tag{4.3}$$

if R as its key column included. In other words, R has no duplicates.

The above transformation rules follow from the semantics of Apply as defined earlier in the section. The first transformation allows us to remove *Apply*, and its application results in a decorrelated expression. The second transformation addresses the case corresponding to subqueries with scalar aggregates. It allows the computation of scalar aggregates to be deferred and replaced by computation of the group-by after *Apply*. Note that when for a tuple of the outer relation, the inner subquery is empty, the Scalar Aggregate w is still computed (returns NULL or zero depending on the aggregate function). However, in contrast, group-by does not output a tuple if a partition is empty. To accommodate this difference, A^X is replaced with A^{LOJ} to preserve the semantics of scalar aggregates. Since R includes its key columns, by doing a group-by on the key columns of R, we ensure that no spurious duplicate tuples are produced in the output.

We now revisit the decorrelation of Query 18. We note the outer relation contains its key *p_partkey* and in fact the same key attribute is also the parameter of the PRE for the subquery. The first step is representing the correlated subquery algebraically using *Apply* with cross product, i.e. A^X (or $Apply^X$). This step of algebraization is represented in Figure 4.15a. Subsequently, since the conditions for Rule 4.2 are satisfied, we apply that rule to obtain the query tree shown in Figure 4.15b. In this figure, the aggregation function $COUNT(Y)$ is over the derived column created by *Apply*. We note that there are no occurrences of correlated variables below A^{LOJ}, and we can therefore use the Rule 4.1 that results in the removal of *Apply* as shown in Figure 4.15c. Thus, the final step of decorrelation is achieved and the resulting query tree is

equivalent to Query 19, which has its nesting removed. The same steps apply for Query 15 as well. However, given the semantics of SUM, at the end, LOJ may be replaced by equi-join resulting in a query tree equivalent to the unnested query Query 17.

Note that unlike the other two transformations for *Apply*, Rule 4.3 results in duplication of expressions as the outer relation occurs in both the subexpressions on the right side of the transformation rules. The duplication raises the possibility of increased work. Therefore, application of this transformation must be strictly cost-based. These, as well as additional transformation rules for Apply are described in [69], which are able to decorrelate a large class of queries.

Many decorrelations are performed as a normalization step. Example queries discussed in this section all fall in that category. Such normalization is especially important for many distributed data platforms where decorrelation is a necessary condition to execute SQL queries in a distributed manner. However, as the discussion on Rule 4.3 notes, not all the decorrelations necessarily result in reduced cost. Certain corner cases, such as use of subqueries in contexts where they are required to have a single row only but are not guaranteed to comply with that at query optimization time, or where there is interaction between decorrelation and short-circuiting in conditional scalar evaluation, add to complications with decorrelation. When there are user-defined-functions in the subqueries, decorrelations through removal of Apply are inhibited. However, even in such cases where Apply cannot be removed, techniques such as caching of results of Apply, partially sorting the parameters with which Apply is invoked, and prefetching are useful in reducing cost of evaluating subqueries [62]. Additional opportunities for optimizations of nested subqueries are also discussed in the rest of this section.

4.5.4 Additional optimizations for queries with nested subqueries

Even though decorrelation is the most important class of optimizing transformations for complex queries with nested subqueries, there are other transformations that can contribute to more efficient computation of queries with nested subqueries even though they do not contribute to decorrelation. Specifically, when two subqueries of the same query have

shared common expressions, opportunities to simplify the structure of the subquery arise. Let us consider the example Query 20 below which has two EXISTS clauses on the same table, i.e., the *orders* table.

Query 20

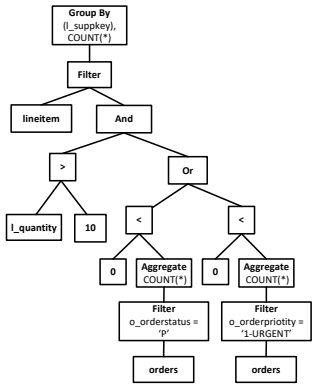
```
SELECT l_suppkey, COUNT(*)
FROM lineitem
WHERE l_quantity > 10 AND
  EXISTS (SELECT *
    FROM orders
    WHERE o_orderkey = l_orderkey
    AND o_orderstatus = 'P')
  OR
  EXISTS (SELECT *
    FROM orders
    WHERE o_orderkey = l_orderkey
    AND o_orderpriority = '1-URGENT')
GROUP BY l_suppkey
```

For each row r from the *lineitem* table in the outer block, both EXISTS clauses check if there exists a row from *orders* table that joins with r on *o_orderkey* and satisfies a predicate, i.e., *o_orderstatus = 'P'* in the first subquery, and *o_orderpriority = '1-URGENT'* in the second. Since the two subqueries in the EXISTS clause only differ in the predicate filter, we can combine the two subqueries with a disjunction of their predicate filters as shown in Query 21:

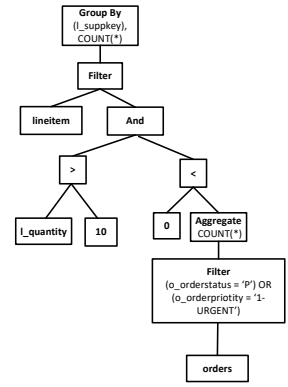
Query 21

```
SELECT l_suppkey, COUNT(*)
FROM lineitem
WHERE l_quantity > 10 AND
  EXISTS (SELECT *
    FROM orders
    WHERE o_orderkey = l_orderkey AND
    (o_orderstatus = 'P' OR
    o_orderpriority = '1-URGENT'))
GROUP BY l_suppkey
```

Figure 4.16b shows the logical expression before and after the transformation. Note that the EXISTS clause is expressed as subquery with a scalar aggregate predicate COUNT(*) > 0. Formally, if a query contains a disjunction of two EXISTS subqueries, and the subqueries differ only in their filter predicates, then the two subqueries can be coalesced into a



(a) Original logical expression with two correlated subqueries



(b) Transformed logical expression with single subquery

Figure 4.16: Transform a query with two EXISTS subqueries into a query with a single EXISTS subquery.

single subquery with a disjunction of the original filter predicates. A dual of this transformation can also be applied for the case of conjunction of multiple NOT EXISTS subqueries with a common core. This as well as other rules related to combining subqueries that take advantage of common subexpressions among nested subqueries may be found in [16], and they help avoid paying the cost of unnecessary computation of similar subexpressions. Later in this section, we will discuss *magic sets* that too benefit evaluation of nested subqueries (Section 4.6.2).

4.6 Other Important Transformation Rules

We discuss four important classes of transformation rules that are commonly used in practice and are more complex than the rules we have presented thus far: (1) Join ordering for star and snowflake queries (Section 4.6.1), (2) Sideways information passing (Section 4.6.2), (3) User-defined functions (Section 4.6.3), and (4) Materialized views (Section 4.6.4). We illustrate the transformations using rules from Microsoft SQL Server.

4.6.1 Star and snowflake

Many data warehouses are designed using a star or snowflake schema, which are optimized for efficiency in querying and in loading data [31]. In a star schema, the database consists of a single fact table, e.g., an $Orders$ table that stores a row for each order, and one table for each dimension such as $Products$, $Customers$, $Country$, $Time$. Each row in the fact table consists of a foreign key to each of the dimensions, and stores the numeric measures for that particular combination of dimensional values. In our example, the numeric measures corresponding for a given $Product$, $Customer$, $Country$, and $Time$ value might include quantity of the item sold and unit price of the item sold. Each dimension table consists of columns that correspond to attributes of the dimension, e.g., the customer name, address, phone number for the $Customer$ dimension table. Snowflake schemas provide a refinement of star schemas where the dimensional hierarchy is explicitly represented by normalizing the dimension tables, e.g., the Country dimension might be represented as a hierarchy of tables, one each for Country and Region.

Queries over star and snowflake schemas typically follow canonical patterns. A star query joins the fact table with one or more dimension tables using primary-key foreign-key joins. They use filter predicates on the dimension tables to restrict the data that needs to be accessed from the fact table, and the result of the joins is aggregated using measures from the fact table. The results are optionally grouped using columns from the dimension tables. A simple example of such a query might be: for each $(Product, Country)$ find the total $Sales$ (which is an attribute in the fact table $Orders$) for $Year = 2024$ (which is a predicate on the $Time$ dimension table).

Microsoft SQL Server has several rules in place for leveraging the above query patterns by identifying promising plans for *star* and *snowflake* queries [72]. They heuristically detect if the query should be classified as a star or snowflake join, and if so, they generate new logically equivalent join orders, which are added to the memo. An example of such a rule is one that generates a right deep tree that joins the fact table with the dimension tables, where the dimension tables are ordered from the most selective to the least selective, based on the estimated

selectivity. Selectivity for a dimension table is defined as the fraction of rows of the fact table that join with the filtered dimension table. The pseudocode for this transformation is shown in Transformation 13 and the logical expression generated by this transformation is shown in Figure 4.17. The *ExtractStarJoin* function attempts to identify if the join graph of the query meets the conditions of a star join, and if so, extracts the fact and dimension tables. Besides the check for primary-key foreign-key joins, additional checks are used, such as a minimum absolute size of the fact table, since the benefits of this optimization are significant only for large fact tables.

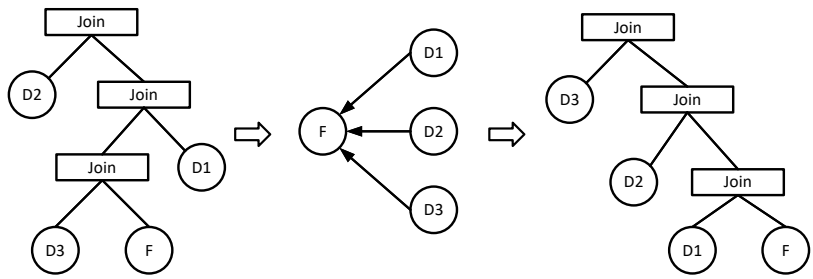
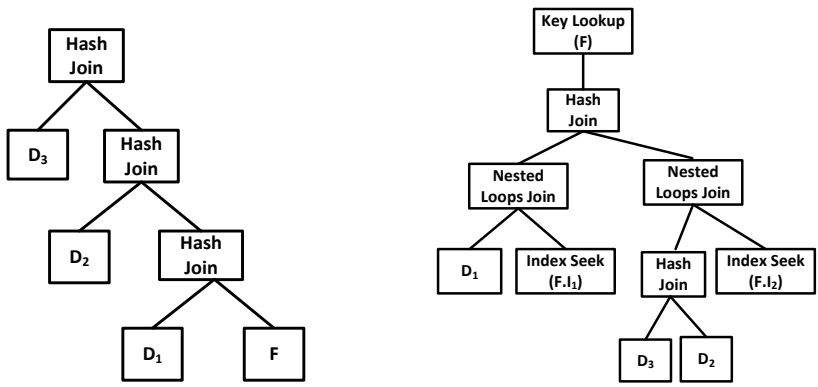


Figure 4.17: Example of transforming a star query with fact table F and dimension table D_1, D_2, D_3, where D_1 is the most selective and D_3 is the least selective.



(a) Right deep plan with Hash Joins **(b)** Index Intersection plan

Figure 4.18: Examples of plans generated for the star query shown in Figure 4.17

Transformation 13 Transform a star query into a right deep tree, where the fact table joins with the dimension tables from the most selective to the least selective.

```
1: function ExtractJoinTables(expr)
2:     if expr.root = LogOpJoin then
3:         tables0 ← ExtractJoinTables(expr.left)
4:         tables1 ← ExtractJoinTables(expr.right)
5:         if table0 = null or table1 = null then
6:             return null
7:         else
8:             return tables0 ∪ tables1
9:     else if expr.root = LogOpGet then
10:        return {expr.table}
11:    else
12:        return null
13: function ExtractStarJoin(expr, tables)
14:    fact ← null
15:    dims ← null
16:    for t ∈ tables do                          ▷ Check if t is a fact table
17:        flag ← True
18:        for s ∈ tables, s ≠ t do
19:            cols ← ExtractJoinColumns(t, s, expr)
20:            if !IsManyToOneJoin(s, t, cols) then
21:                flag ← False
22:                break
23:        if flag = True then
24:            fact ← t
25:            dims ← tables \ {t}
26:            break
27:    return fact, dims
28: function CheckPattern(expr)
29:    tables ← ExtractJoinTables(expr)
30:    if tables = null then return False
31:    fact, dims ← ExtractStarJoin(expr, tables)
32:    if fact = null then
33:        return False, fact, dims
34:    else
35:        return True, fact, dims
36: function Transform(expr, fact, dims)
37:    sortedDims = SortBySelectivity(fact, dims)
38:    newExpr ← LogOpGet(fact)
39:    for d ∈ sortedDims do
40:        left ← LogOpGet(d)
41:        joinCond ← ExtractJoinCond(left, newExpr, expr)
42:        newExpr ← LogOpJoin(left, newExpr, joinCond)
       return newExpr
```

In addition to the logical expressions generated by these transformation rules, specific star join *implementation* rules are also applied to generate physical plans corresponding to the generated logical expressions. For example, one implementation rule generates the right deep tree containing Hash Join, shown in Figure 4.18a, from the logical expression shown in Figure 4.17. In this plan, the build phase of each Hash Join accesses the respective dimension tables, and each of these builds are completed prior to the probe phase where the fact table is scanned. Therefore, one additional benefit of this plan is that it enables the use of bitmap filters created during the build phase of each Hash Join to filter out rows during the Scan of the fact table during the probe phase (see Section 4.6.2 for more details).

Query optimizer developers can also generate other promising plans for star and snowflake queries. Another example of such a plan is shown in Figure 4.18b. This plan aims to reduce the cost of accessing the fact table by using indexes on the join columns of the fact table, if available. Such a plan may be suitable when the selectivity of the dimension tables is high, since it allows the qualifying row IDs of the fact table to be retrieved using the Index Seek operator. The two sets of row IDs are then intersected using a Hash Join, and the rows corresponding to the matched row IDs are retrieved from the fact table using the Key Lookup operator. Other examples of plans generated for star and snowflake queries are described in [72]. The plans thus generated are added to the memo, and the choice of the final plan for the query is done in a cost-based manner as part of the search as described in Section 2.3.3.

Finally, we note that these rules are invoked on the expression of the query at the beginning of the query optimization to *seed* the search space with promising logical and physical expression, and the rules can be disabled for the rest of the search with the guidance mechanism (see discussion in Section 2.4 on macro rules).

4.6.2 Sideways information passing

Execution of subexpressions of a query may induce constraints on the remainder of the queries due to predicates in the query. Such constraints help reduce the amount of data that needs to be processed for that

subexpression. Optimization techniques that take advantage of the above insight are referred to as *sideways information passing (SIP)*. In this subsection, we review a few examples of optimization based on SIPs. We start with *semi-join*, a technique that is widely applicable in many facets of query processing. Next, we discuss *bitvector filtering*, a technique related to semi-join that is widely used in query processing in most database engines, including in Microsoft SQL Server. Finally, we describe *magic sets*, a technique that complements decorrelation optimization we have discussed previously. Although not presented in this subsection, interested readers may want to learn about the use of SIPs to enable data skipping in big data systems [106].

Semi-join The semi-join operator ($\ltimes_\theta$) takes two relations R and S, and a join condition θ, and it returns the rows in R that join with at least one qualifying row in S [17]. In other words, the semi-join operator filters out the rows in R that do not have any matching row in S. Table 4.4 shows two examples of semi-joins between tables R and S, where the tables are joined on column b, specifically, $R \ltimes_{R.b=S.b} S$ and $S \ltimes_{S.b=R.b} R$.

Table 4.4: Examples of semi-joins on R and S.

(a) Table R

$R.a$	$R.b$
1	4
2	5
3	6

(b) Table S

$S.b$	$S.c$
4	1
6	3
8	5

(c) $R \ltimes_{R.b=S.b} S$

$R.a$	$R.b$
1	4
3	6

(d) $S \ltimes_{S.b=R.b} R$

$S.b$	$S.c$
4	1
6	3

The semi-join operator can be helpful in reducing the cost of a join. By applying the semi-join operator on R to filter out irrelevant rows in R before joining it with S, the cost of the actual join can be reduced. For example, in Table 4.4, if we first execute $R' = R \ltimes S$, then when joining S with R', instead of joining three rows from R, we only need to join two rows from R'. In particular, in the distributed setting where R and S are at two different sites, semi-join is able to reduce the cost of data transfer. Thus, we are using the following transformation rule:

- $R \bowtie_\theta S \iff (R \ltimes_\theta S) \bowtie_\theta S$, where θ is the join predicate between R and S.

Note that the introduction of semi-join based filtering may not always reduce the query plan cost because the overhead of creating and applying the filter can outweigh its benefit if the filter is not selective enough. Thus, the introduction of a semi-join based filtering needs to be a cost-based decision.

All implementation methods for join can be used for semi-join. Since semi-join does not need to find all matching rows but need only verify at least one match, early termination could be used to speed up the execution of the join methods [62]. However, the decision on what join method should be used needs to be cost-based. For example, depending on the selectivity and availability of the access methods, Nested Loops Join with Index Seek or Hash Join may be the preferred implementation.

The following two equivalent transformations are important for semi-joins [62]:

1. $(Group\ By(g, A)R) \ltimes_\theta S = Group\ By(g, A)(R \ltimes_\theta S)$, where g are the group-by columns and A is the set of aggregates, as long as θ is over a subset of g of the relation R.
2. $R \ltimes_\theta S = Group\ By(R.key, Any)(R \bowtie_\theta S)$, where Any denotes the columns of any row in the group, i.e., there is exactly one row in each group as $R.key$ is the primary key of R.

The first equivalence shows how the semi-join behaves much like filter expressions. The second equivalence shows how a semi-join may be converted to an inner join. The significance of the second transformation above is that by converting a semi-join to a join, we may be able to expand alternatives for the optimization, e.g., enable reordering with other joins.

Bitvector filters Semi-join performs membership testing for values of the join columns from each row in R with the set of join column values in S. Such exact membership testing can be implemented efficiently with a *bitmap filter* when the domain of the join column values is small by assigning one bit from the bitmap for each value in the domain. However, when the domain of the join column values is large or the values of the join column populate sparsely in the domain, bitmap filters can consume too much memory. Fortunately, for the purpose of reducing the cost of joins, the membership testing does not need to be exact as

long as it has no false negatives, i.e., no matching rows are incorrectly eliminated. Therefore, *approximate* membership testing, i.e., one that allows *false positives*, is acceptable as such false positives will not impact the result of the join. Probabilistic data structures such as *Bloom filters* provide an efficient way to perform approximate membership testing, allowing a trade-off between space efficiency and false positive rate. For example, instead of performing $R \ltimes S$ to filter out the rows from R as shown in Table 4.4c, we create a Bloom filter $\mathcal{B}$ on $S.b$ and then perform membership testing on $R.b$ against $\mathcal{B}$ for each row in R. Only the rows that pass the membership test need to perform the actual join with S.

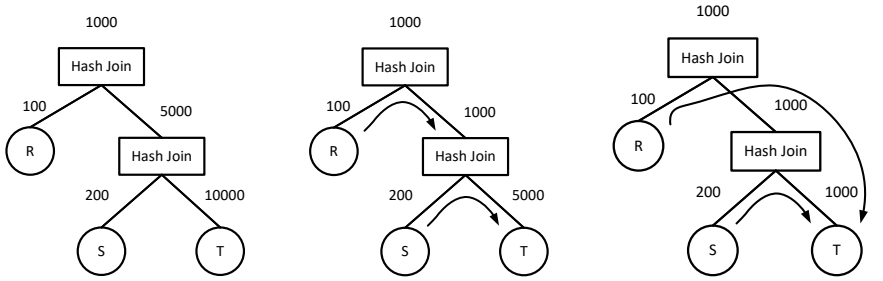
We use the term *bitvector filtering* to denote both exact, i.e., bitmap filters, and approximate filters, i.e., Bloom filters. Bitvector filtering becomes even more powerful when the filter is pushed multiple levels down the query operator tree [78]. Consider the following Query 22:

Query 22

```
SELECT *
FROM R, S, T
WHERE R.a = T.a AND S.b = T.b
```

Figure 4.19 shows three plans for Query 22 with no bitvector filter push-down, one-level bitvector filter push-down, and multi-level bitvector filter push-down. If there is no bitvector filtering, the join of S and T takes 10000 rows from T and outputs 5000 rows for its parent Hash Join operator (Figure 4.19a). If we create a bitvector filter $\mathcal{B}$ from the build side of each Hash Join operator and apply to the probe side, assuming $\mathcal{B}$ has no false positives, then the join of S and T takes 5000 rows from T and outputs 1000 rows (Figure 4.19b), which reduces the cost of both the join of R and S as well as its parent join. Finally, because the filter $\mathcal{B}_{\mathcal{R}}$ created from $R.a$ actually applies on the column $T.a$, instead of applying $\mathcal{B}_{\mathcal{R}}$ on the output of joining R and T, we can push down $\mathcal{B}_{\mathcal{R}}$ to T. As shown in Figure 4.19c, this further reduces the cost of joining S and T by eliminating additional rows from T.

The decision of pushing down filters is based on the column(s) used for the membership testing. If all the columns used for membership testing are from a single child operator of the parent operator, the filter can be pushed down to the child operator. Otherwise, the filter can be



(a) Plan P_1 without bitvector filter push-down

(b) Plan P_2 with one-level bitvector filter push-down

(c) Plan P_3 with multi-level bitvector filter push-down

Figure 4.19: The plans for query $S \bowtie (R \bowtie T)$ with and without bitvector filters. The arrow shows the creation and the application of bitvector filters. The number on the edge shows the output cardinality of the operator.

pushed down either to all the child operators with the relevant columns, or the filter can be applied only to the parent operator if the filtering is only effective at the parent operator.

To enable bitvector filtering, query execution engines implement a physical operator to create a bitvector (Bitvector Create) and a physical operator to evaluate the bitvector filter (Bitvector Filter). The decision of where to place these operators in the plan is done by most query optimizers in a post-processing step *after* the query optimizer has found the best plan for the query. While such a posteriori placement of bitvector filters can improve the efficiency of joins for a given plan, there is an opportunity to influence the plan generated by the optimizer by considering placement of bitvector filters during search. This is because the placement of bitvector filters can change the cardinality of an expression. Integrating such filtering into the query optimization introduces new challenges in search due to the increased search space of query optimization [58]. Finally, we note that for distributed query execution, constructing bitvector filters may become a scheduling barrier [106], and therefore the use of bitvector filtering should be a cost-based decision.

Magic sets Magic sets generalize the idea of semi-joins of imposing restriction on relations involved in joins to subqueries [148]. Consider Query 23 that finds each product whose color is Blue and weight is less

than 1 lb, and whose price is higher than the average price of products in its category:

Query 23

```
SELECT name
      FROM products AS p1
WHERE  color = 'Blue' AND weight < 1 AND price >
  (SELECT AVG(p2.price) FROM products AS p2
  WHERE p2.category = p1.category)
```

The above query contains a correlated subquery, where the subquery that computes AVG(*price*) is invoked with the value *p1.category* from the outer block. For this query, there are two opportunities to save computation. First, we only need to compute the average price for product categories that qualify after applying the filter $color =' Blue'$ and $weight < 1$. Second, we can compute the average price for these categories only once so that we do not need to compute them repeatedly with every choice of the product from the outer relation.

Based on these observations, we can execute the query in four steps:

1. Find all the products with $color =' Blue'$ and $weight < 1$ (view T_1).
2. Find distinct categories in T_1 (view T_2).
3. Find the average price of *products* for each product category that is among the product categories in T_2 (view T_3). For this step, T_2 acts as the magic set and it helps limit the number of subqueries that are computed.
4. Find the products with $color =' Blue'$ and $weight < 1$ whose price is higher than the average price by joining T_1 with T_3 on the product category attribute. For this step, T_3 is a magic set that restricts T_1.

As with semi-joins, the creation of magic sets introduces additional overhead, which may not always pay off. For example, in the extreme case, all the categories have blue products with weight > 1 lb in Query 23, then the auxiliary table T_2 will not reduce any rows from *products*. Thus, the application of magic sets need to be *cost-based*.

In Section 4.5, we have discussed how the algebraic framework of *Apply* may be used for decorrelation. We now discuss how the magic sets technique may be cast in that framework for decorrelation. If we review the steps outlined in the example above, we note that we first executed the outer query block in Query 23 without considering the effect of the

correlated subquery. Thus, T_1 is a *superset* of the qualifying tuples of the outer relation. In the second step, we identified all *distinct* categories in T_1. These are the parameters with which the nested subquery will need to be evaluated. This is the set T_2. In the third step, we evaluated the nested subquery for every value of the product category in T_2. In other words, T_2 is the "outer" in *Apply* using which we evaluated the following expression to obtain T_3:

$$T_3 = T_2 \; Apply^X (\sigma_{T2.category=p2.category} \; AVG(p2.price) \; products \; as \; p2)$$

Once we have obtained T_3, we joined T_3 with the outer relation(T_1) on product category to produce the final result. Thus, instead of invoking the PRE with the set of parameter values P from the outer relation R, the PRE is pre-computed with a set of parameter values P', where $P \subseteq P'$. Here, P' is the magic set. A common choice of P' is the set of all possible distinct parameter values, e.g., all distinct category values in the *products* table. In the example above, P' is the set T_2. Once this step is completed and a relation is obtained (T_3 in the example), a traditional equi-join completes the query. In the example above, this is the join between T_1 and T_3. As the example illustrates, magic sets in itself does not directly do decorrelation but it leverages semi-join techniques to reduce the cost of evaluating the nested subqueries [62].

4.6.3 User-defined functions (UDFs)

User defined functions (UDFs) allow users to add customized functions which can be invoked through the SQL language interface. Many SQL queries used for data analytics contain user-defined functions written in languages such as Python, Java, C#, etc. Such user defined functions could occur as tables, aggregate functions, or filters in a SQL statement. There are many unique challenges for optimizing queries with user-defined tables, aggregate functions, and filters, e.g., the costing information for UDFs have to be provided to the optimizer. Although we do not offer a full discussion of how UDFs are handled by the query optimizer and the query execution engine, we discuss two important optimization opportunities.

Let us consider the case of optimizing SQL queries with one or more UDF filters. For user-defined filters, there is a need to trade-off

selectivity and the cost of evaluating the filter. Therefore, the rules for push-down and pull-up that we discussed in Section 4.2.2 are important for user-defined filters as well. Using the above rules, query optimizers can generate alternative plans with different placement of UDF filters in the plan, including placing them above a join expression, and such choices should be made in a cost-based manner. We refer the readers to [39, 91] for details of how UDF filters may be evaluated in a cost-based manner.

In the rest of this section, we address a sound way UDFs may be transformed into an equivalent SQL statement without any UDFs. Such transformations are especially significant as after the transformation, the optimizer is able to use its full repertoire of query optimization techniques, and it also does not require any information on UDF costing.

Rewriting UDFs to SQL The techniques that consider UDFs as a black box during query optimization can lose opportunities for improving plan quality. In contrast, rules that rewrite a UDF to an equivalent sequence of SQL statements can enable the optimizer to apply other transformation rules, such as those described in this section, and thereby lead to the generation of a more efficient plan.

Consider the following example UDF Query 24, which finds the name and product classification for each product based on the amount of sales.The use of the UDF conveniently separates the main query and the logic of determining the classification of a given product. However, the query invokes the UDF with $p\_productkey$ for every row in $product$. Observe that the UDF can be expensive since for each $pkey$ value the $orders$ table needs to be accessed. While rewriting techniques are available for this query if we embed the UDF as a subquery in the main query (see Section 4.5), the query optimizer will not be able to exploit these techniques if it treats the UDF $product\_classification$ as a black box. Moreover, compared to nested evaluation of subqueries, executing the query with a UDF can be even slower because of the overheads of serialization/deserialization and function invocation for each row.

Query 24

```
SELECT p_name, product_classification (p_productkey)
FROM product

-- UDF definition
CREATE FUNCTION product_classification(@pkey int)
RETURNS char(10) AS
BEGIN
    DECLARE @total float;
    DECLARE @class char(10);
    SELECT @total = SUM(o_totalprice)
        FROM orders WHERE o_productkey = @pkey;
    IF (@total > 5000000) SET @class = 'High';
    ELSE IF(@total > 25000) SET @class = 'Medium';
    ELSE SET @class = 'Low';
    RETURN @class;
  END
```

Given that UDFs are expressive, it is challenging to design techniques to allow the query optimizers to reason about the semantics of UDFs in general. Below, we discuss one technique that opportunistically "opens up" a certain class of UDFs by converting them to equivalent SQL.

The work by FROID [172], which is used by Microsoft SQL Server, proposes a technique to automatically inline *scalar* UDFs i.e., UDFs that return a scalar value as shown in the example Query 24 with imperative constructs including DECLARE, SET, SELECT, IF/ELSE, RETURN, and UDF (i.e., nested UDFs). The key insight is to derive an equivalent transformation from imperative statements to SQL statements so that the UDF is replaced by an equivalent nested subquery. For example, the IF/ELSE statement can be transformed to CASE WHEN; the SET statement can be transformed into SELECT AS. FROID combines individual statements into a single expression, potentially with nesting and derived tables, and rewrites the original query with UDFs to SQL query without UDFs. Thus, the optimizer can leverage transformation rules such as decorrelation (Section 4.5) to optimize the query. For the UDF shown in Query 24, FROID produces an equivalent SQL query as shown in Query 25 where DT1 and DT2 are derived tables:[4]

[4]OUTER APPLY is the Microsoft SQL Server syntax for the $Apply^{LOJ}$ operator discussed in Section 4.5.

Query 25

```
SELECT DT2.class FROM
(SELECT
  (SELECT SUM(o_totalprice) FROM orders
    WHERE o_productkey = @pkey) AS total) DT1
OUTER APPLY
  (SELECT
    CASE WHEN DT1.total > 5000000 THEN 'High'
       WHEN DT1.total > 250000 THEN 'Medium'
       ELSE 'Low'
    END
   AS class) DT2
```

In FROID the above transformation from a UDF in an imperative language to SQL is done prior to query optimization, and this decision is not cost-based. While this is the right decision for most queries (due to overheads of UDF execution), in general this transformation can be done during query optimization in a cost-based manner in Volcano/Cascades. Finally, we note that it is also important to enable transformation of UDFs in imperative languages such as Python, Java or C#, which are used widely in practice, to SQL, e.g., as in [42].

4.6.4 Materialized views

A *materialized view* is defined by a SQL SELECT statement. In contrast to a *view*, when a materialized view is created, the *view definition*, i.e., the SELECT statement corresponding to its definition is executed, and its results are materialized – i.e., persisted in the database similar to a base table. Thus, a materialized view is a precomputation of the results of a SQL query. Since the expression corresponding to the materialized view may consist of expensive operations such as join and aggregation, there is an opportunity to potentially speed up the execution of any query Q for which the query optimizer is able to use the materialized view to answer a subexpression of Q. Materialized views are especially important for speeding up the execution of analytic queries which perform expensive operations on large amounts of data. We illustrate the use of a materialized view with an example. Consider the following Query 26:

Query 26

```
SELECT p_size, SUM(l_quantity)
FROM lineitem, part
WHERE p_partkey = l_partkey
GROUP BY p_size
```

Suppose we have created a materialized view *mv* that aggregates *l_quantity* by *l_partkey* (as defined in Query 27). Then, it is possible to equivalently rewrite the query to join *part* with the materialized view *mv* as shown below in Query 27:

Query 27

```
---- define materialized view ---
CREATE MATERIALIZED VIEW mv AS
    SELECT l_partkey, SUM(l_quantity)
    FROM lineitem
    GROUP BY l_partkey

---- query with materialized view ----
SELECT p_size, SUM(l_quantity)
FROM mv, part
WHERE p_partkey = l_partkey
GROUP BY p_size
```

The rewritten query may execute much faster, e.g., when *mv* happens to be much smaller than the base table *lineitem*. This may happen when there are many fewer distinct values of *l_partkey* compared to the number of rows in *linietem*. In general, using a materialized view to answer a query may not always result in a faster execution plan, and the decision must be done in a cost-based manner.

To take advantage of materialized views, the query optimizer needs to: (1) Determine which materialized views can be used to answer a given expression. This problem is referred to as *view matching*. View matching can be applied for any sub-expression of the query. The problem of answering queries using views has been studied extensively (e.g., see [87] for a survey). (2) When a view can be used to answer the query, but does not exactly match the expression, derive the *residual* expression to apply to the view to ensure that the results exactly match the expression, and (3) Estimate the cost of the physical plan that uses

the materialized view. Below, we describe how each of these steps are accomplished in an extensible optimizer.

View matching In Volcano/Cascades, view matching is achieved using transformation rules. The transformation of a logical expression to use materialized views is integrated into the optimizer as a rule, where the *CheckPattern* function contains the view matching logic and the *Transform* function derives the new logical expression with the view and the residual, if any. This rule can be triggered for each logical expression in the memo during the search, or applied selectively with heuristics, e.g., only on expressions referencing large tables, by using the guidance mechanism (Section 2.3). For example, in Query 26, when the optimizer searches the best plan for $LogOpGroupBy(part \bowtie lineitem, p\_size, SUM(l\_quantity))$, by pushing down the group-by (see Section 4.4), the logical expression can be transformed into $part \bowtie LogOpGroupBy(lineitem, l\_partkey, SUM(l\_quantity))$. The optimizer finds that the view mv defined in Query 27 is an exact match for $LogOpGroupBy(lineitem, l\_partkey, SUM(l\_quantity))$, which it then transforms into an equivalent logical expression $LogOpGet(mv)$. Finally, we note that view matching is an expensive step, and therefore techniques for efficiently filtering out views that cannot match an expression can significantly speed up this step (e.g., [76]).

Residuals When the materialized view contains the rows needed to compute the expression, but is not an exact match for the expression, a residual expression is needed as post-processing. Consider the following example Query 28. While the materialized view mv is not an exact match of the logical expression of the original query in Query 28, it *contains* the result of the expression. Thus, mv can still be used for the original query with a residual predicate filter as shown in Query 28.

Implementation rules and costing As far as access methods are considered, a materialized view behaves like a base table, and can have clustered and non-clustered indexes on it. In the above example, implementation rules that convert $LogOpGet(mv)$ to use access methods (e.g., Clustered Index Scan) can therefore be applied similar to any

base table as discussed in Section 4.1. Costing of a physical plan that references a materialized view is also done in the same manner as any other physical plan as described in Section 5.

Query 28

```
---- original query ----
SELECT l_partkey, SUM(l_quantity)
FROM lineitem
WHERE l_partkey > 10
GROUP BY l_partkey

---- query with materialized view ----
SELECT *
FROM mv
WHERE l_partkey > 10
```

Other aspects There are several other important aspects of materialized views that are beyond the scope of this monograph. We briefly mention some of these aspects and point to relevant references. (1) *View maintenance:* When data in the base tables is updated, a materialized view needs to be updated to prevent it from becoming stale. Therefore, techniques for incrementally updating the materialized view, i.e., without having to recompute it from scratch, become crucial for scalability e.g., see [85] for a detailed description of the problems and solutions. (2) *Expressiveness:* The class of materialized views that can be supported by a DBMS is influenced by whether or not there are efficient algorithms for incrementally maintaining the view. Algorithms for rewriting queries to use Select-Project-Join views with group-by and aggregation are widely supported in DBMSs. (3) *View selection:* There has been a large body of work on deciding which materialized views to create for a given database workload. We refer readers to [43, 84] for more details on the above aspects.

4.7 Suggested Reading

Citation numbers below correspond to numbers in the References section.

[52] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers," in *Proceedings of the 13th International Conference on Very Large*

Data Bases, ser. VLDB '87, pp. 197–208, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987

[38] S. Chaudhuri *et al.*, "Including Group-By in Query Optimization," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94, pp. 354–366, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994

[70] C. Galindo-Legaria *et al.*, "Outerjoin Simplification and Reordering for Query Optimization," *ACM Trans. Database Syst.*, vol. 22, no. 1, Mar. 1997, pp. 43–74. DOI: 10.1145/244810.244812

[69] C. Galindo-Legaria *et al.*, "Orthogonal Optimization of Subqueries and Aggregation," SIGMOD '01, 2001, pp. 571–581. DOI: 10.1145/375663.375748

[62] M. Elhemali *et al.*, "Execution Strategies for SQL Subqueries," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07, pp. 993–1004, Beijing, China: Association for Computing Machinery, 2007. DOI: 10.1145/1247480.1247598

[153] T. Neumann *et al.*, *Unnesting Arbitrary Queries*, 2015

5

Cost Estimation

Cost estimation, i.e., the task of estimating the cost of executing a plan, is not unique to extensible optimizers. In fact, the cost estimation framework in most modern cost-based optimizers follows the approach of System R, which we outlined briefly in Section 1.2. In this section, we start with an overview of how cost estimation is used in an extensible query optimizer (Section 5.1). We then describe techniques for modeling the cost of an execution plan (Section 5.2). We next briefly review statistical summaries (aka *statistics*) used by the optimizer for cardinality estimation, focusing on *histograms*, one of the most widely used summaries in practice (Section 5.3). We also describe *sketches*, a newer class of statistical summaries, which have been gaining adoption in databases recently. We further give an overview of how DBMSs manage statistics over the lifetime of a database. We follow this with a description of techniques for cardinality estimation (Section 5.4). We conclude this section with a case study of how cost estimation works in Microsoft SQL Server (Section 5.5). In this section we do not discuss recent machine learning based techniques for cardinality estimation and cost estimation, which is an active area of exploration. We defer the discussion of those techniques to Section 7.

5.1 Cost Estimation Overview

The search algorithm of the optimizer uses cost estimation to provide an estimate of the efficiency of a given physical plan, which is used to compare the plan against other alternative plans for the same logical expression. Figure 5.1 shows the search and cost estimation modules in a query optimizer and their interaction. During the search for the best plan for a logical expression, the optimizer uses the cost estimation module to derive the cost of the plans and chooses the best one. The cost of a plan is estimated by combining the costs of individual operators in the plan. The cost of an operator is estimated using a *cost model*. The cost model for an operator (e.g., Hash Join) uses formulas to estimate the resources (e.g., CPU, I/O, memory) consumed by that operator. The cost model is operator specific. For example, the cost model for a Hash Join must capture (1) the cost of building the hash table, (2) the cost of probing the hash table, and (3) the cost of constructing the join result for matching rows. The most important parameters of the cost model of an operator include the *number of rows* or *cardinality* of its input relations and its output relation. Observe that the number of rows of an input relation of an operator is also the number of rows output by its child operator. Hence, the ability to accurately estimate the number of rows output by each operator, referred to as *cardinality estimation*, is crucial to ensure good accuracy of the cost estimation.

Cardinality estimation takes a logical query expression as input and uses *statistical summaries* of the data of the relations referenced by the logical expression. For example, the expression $Join(Join(R, S), T)$ with $R.a = S.b$ and $R.d = T.c$ references three relations, and information about the distribution of values in the columns $R.a$, $S.b$, $R.d$, and $T.c$ can be useful for estimating its cardinality. Hence database systems must collect and maintain these statistics over the lifetime of the database (see the Data Statistics module in Figure 5.1). The major kinds of statistical summaries, also referred to as *synopses*, that have been proposed for cardinality estimation include[1] *histograms* [111], *sketches* [192], and *samples*.

[1] *Wavelets* [131] have also been considered for cardinality estimation but they have not been adopted in practice by database systems.

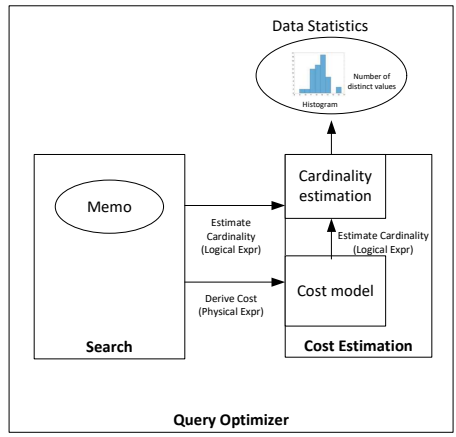


Figure 5.1: Search and cost estimation modules in a query optimizer. Cost estimation uses a cost model, which in turn uses a cardinality estimation module.

Finally, we note that the search algorithm may directly call the cardinality estimation module for a logical query expression to make decisions during its plan search as shown in Figure 5.1. For example, in Volcano/Cascades, the *promise* of a join transformation (see Section 2.2.3) may be based on whether the outer side has a larger (or smaller) estimated cardinality than the inner side.

5.2 Cost Model

The cost model estimates the cost of an individual physical operator in a plan, and then it combines the costs of all the operators in the plan to produce the total cost of the plan. There are several factors that determine the cost of executing a particular operator: (1) The amount of data it processes, i.e., data consumed from each of the operator's inputs. (2) The cost incurred by that operator for processing each unit of data and producing the output. (3) Runtime factors that can affect the cost of executing the operator. The amount of data processed by the operator is based on the number of rows from its inputs, i.e., *cardinality*, and the size of these rows. The cost of processing a unit of data includes I/O and CPU cost. The runtime factors may include the degree of parallelism (i.e., number of concurrent threads used to execute that

operator), the amount of memory available for an operator such as Hash Join or Sort, and the contents of the buffer pool.

Figure 5.2 shows an example query and its plan with a simplified cost model. Each operator in the plan is annotated with the size of intermediate result, i.e., cardinality, and its cost. Consider, for example, the Hash Join operator. It builds the hash table over 100 rows output by the Table Scan operator over the *store* table. It then probes the hash table with 1000 rows output by the Index Scan operator over the *store_sales* table. It outputs 250 rows, which are consumed by its parent Hash Aggregate operator. In our simplified cost model, inserting a row into the hash table costs 3, probing the hash table costs 1, and producing an output row costs 1. Therefore, the estimated cost of the Hash Join operator is 1550.

```
SELECT s_city, COUNT(*)
FROM store, store_sales
WHERE s_store_sk = ss_store_sk
AND s_state = 'CA' AND ss_year = 2000
AND ss_sales_price > 10
GROUP BY s_city
```

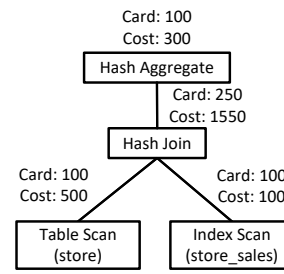


Figure 5.2: Query plan with cardinality and cost estimate

Finally, we note that the factors which impact the cost of an operator can be complex. For example, caching can significantly impact the CPU time of scan operators, such as Table Scan and Index Scan, because scanning the data is faster if the data or a part of the data is cached in the buffer pool memory compared to scanning all the data from the storage subsystem. The work by [55] proposes a technique to model the impact of caching on the cost of Index Scan by maintaining a random sample of rows from each table, which is used during query optimization time to obtain a list of record IDs (*RIDs*) of rows that qualify the selection conditions on that table. These RIDs are then used to probe the index at query execution time to estimate what fraction of data is cached. While modeling the effects of caching on the CPU time of a scan operator can improve accuracy of cost estimation, they are typically

not implemented in database systems due to several reasons. First, the contents of the buffer pool are difficult to predict when there are concurrent queries running in the database system. Second, the content of the buffer pool estimated at query optimization time may be different from that at query execution time. Finally, the overheads of the above estimation are non-trivial and can increase optimization time. Thus, modeling system-wide effects such as caching is an open problem in cost estimation, and most DBMSs do not model the effects of caching.

5.2.1 Cost model calibration

The cost of a physical operator depends on both the statistics of the operator, e.g., number and size of the rows for its input relations and output, as well as a cost formula with cost parameters, i.e., the *cost model*. For example, the CPU time estimate of the Table Scan operator needs to take into account the number of sequential I/Os performed by the operator, the CPU time cost of each sequential I/O, as well as other overhead of the operator, e.g., startup cost. A simple cost model might be $C_{CPU} = C_{IO} \cdot N_{IO} + C_0$, where C_{IO}, C_0 are the *cost parameters* and N_{IO} is the number of sequential I/Os. While the statistics of an operator is specific to the logical and physical properties of the operator, the cost model can generalize beyond a specific operator. Often, the cost formula itself is fixed for a type of physical operators, e.g., $C_{CPU} = C_{IO} \cdot N_{IO} + C_0$ for Table Scan, while the cost parameters can change based on the specific hardware, e.g., hard disk vs. SSD. Thus, the cost parameters need to be properly calibrated to accurately model the specific hardware where the database system runs.

One approach for calibrating cost model parameters, which is used in System R [178], is to execute a set of hand-crafted synthetic queries on the specific hardware and measure the resources consumed by operators in the plan. For example, assume we want to calibrate the parameters of our simple cost model for estimating the CPU time of sequential I/Os, i.e., $C_{CPU} = C_{IO} \cdot N_{IO} + C_0$. By executing a number of Table Scan operators, potentially on tables of different sizes, and measuring their actual execution statistics, i.e., the number of sequential I/Os performed (N_{IO}) and the CPU time (C_{CPU}), the cost parameters C_{IO}

and C_0 can be derived. Since the cost parameters are hardware specific, some commercial database systems will re-calibrate the cost parameters when running on a machine with a specification different from the one used to calibrate the parameters.

Finally, we note that there has been extensive research on cost model design and calibration over the years. The approaches vary, including developing more fine-grained cost models [125], using synthetically created databases and queries to improve the accuracy of calibration [60], automatically designing experiments to calibrate model parameters [197], and using machine learning for cost model calibration [199].

5.3 Statistics

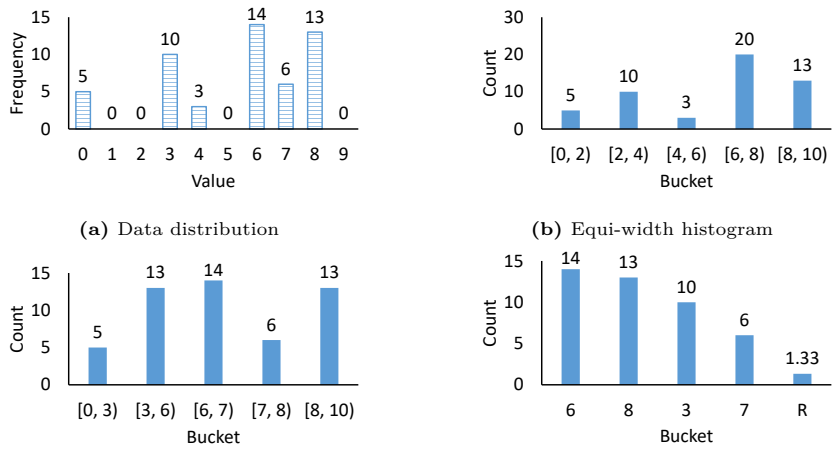
The major kinds of statistical summaries, also referred to as *synopses*, that have been proposed for use in cardinality estimation (CE) include *histograms* [111], *sketches* [192], and *samples*. The key requirements for these statistical summaries with respect to their use in CE are: (1) accuracy (2) efficiency (3) memory consumption (4) coverage i.e., applicability for different types of query expressions, and (5) cost of creation and maintenance. We refer the reader to [46] for a detailed survey on these techniques and their trade-offs in terms of the above requirements. In practice, by far the most popular techniques used for CE of selections and joins in commercial DBMSs are histograms, described in more details in Section 5.3.1. For distinct value estimation, the sketching technique of HyperLogLog [64, 92] is gaining traction in practice. Sampling is also used for cardinality estimation in some DBMSs, although its usage is much more limited compared to histograms. A survey and empirical comparison of various techniques for estimating the number of distinct values is presented in [90].

5.3.1 Histograms

Despite being error-prone, histograms are by far the most widely used techniques for cardinality estimation in practice because of their simplicity and efficiency [101]. A histogram is defined on a *set of columns*. The idea is to use a number of *buckets* and to store an aggregated summary

of the rows of that column set which fall into the corresponding bucket. These aggregates are then used for cardinality estimation of predicates involving that column set. The number of buckets to use in the histogram is an input parameter to histogram construction. Depending on how the buckets are defined and what aggregate summary is stored in each bucket, different kinds of histograms are possible. Below, we illustrate three kinds of commonly used histograms using an example.

Consider a column whose data distribution is shown in Figure 5.3a. In each case, we limit the the number of buckets in the histogram to 5.



(a) Data distribution

(b) Equi-width histogram

(c) Equi-depth histogram

(d) End-biased histogram

Figure 5.3: Data distribution and histograms

Equi-width histogram Figure 5.3b shows an example of an *equi-width* histogram. Each bucket represents a range of contiguous values of equal width, and the aggregate summary stored in the bucket is the number of rows whose value lies within that range. This histogram can be used to estimate the selectivity on the table with range and equality filters. For example, with the histogram in Figure 5.3b, we can estimate the number of rows with value 1 to be $5/2 = 2.5$ making the uniformity assumption (see Section 5.4.2). Although equi-width histograms are simple, they can suffer from poor accuracy of selectivity estimation when the data distribution across values is non-uniform, e.g., the number of

rows for each value within a bucket is very different and deviates from the uniformity assumption [164].

Equi-depth histogram Figure 5.3c shows an *equi-depth* (also called *equi-height*) histogram for the same data distribution, which tries to ensure that the height, i.e., the number of rows falling into each bucket, is the same. In our example, since there are 51 rows and 5 buckets, the buckets are chosen such that each bucket has approximately 10 rows. Thus, the bucket boundaries can be viewed as quantiles of the distribution. Unlike an equi-width histogram, the worst-case errors in selectivity estimates are much lower compared to equi-width histograms [164]. For example, with the equi-width histogram in Figure 5.3b, the error in estimating the cardinality of value 6 is $(20 - 14)/2 = 4$, while that with the equi-depth histogram in Figure 5.3c is 0.

End-biased histogram In an end-biased histogram with N buckets, $N - 1$ buckets are used to store the frequency of the values with the highest frequencies, and the remaining bucket stores the average frequency of all remaining values [103]. When estimating the selectivity of an equality predicate, the estimation is accurate if the value referenced in the predicate is one of the $N - 1$ values with the highest frequencies, whereas for other values the uniformity assumption must be made. End-biased histograms are also effective for estimating selectivity of equi-joins since the join size is often dominated by frequent values from one or both relations being joined. Figure 5.3d shows an end-biased histogram for the data distribution in our example, where the values with the 4 highest frequencies are stored in the first 4 buckets, and the 5^{th} bucket, labeled R in the figure, stores the average frequency of all remaining values. We observe that unlike equi-width and equi-depth histograms where buckets store value ranges in the sorted order of the column's domain, in end-biased histograms, column values of adjacent buckets are based on the frequencies of the corresponding values. End-biased histograms have been shown to perform well when the data distribution is skewed, e.g., there are a few high frequency values and many infrequent values [103]. In our example, the end-biased histogram has the lowest aggregated errors in cardinality estimation

for each value among the three histograms. We note that equi-depth histograms and end-biased histograms have been used in commercial DBMSs such as Oracle [159], IBM DB2 [99], Microsoft SQL Server 7.0 [34], and PostgreSQL [170]. More recent versions of Microsoft SQL Server use *Max-Diff* histograms, which we describe in Section 5.5.2.

Finally, there has been a large body of work on histograms for cardinality estimation [101]. A taxonomy for histograms is proposed in [168] to characterize the properties of different kinds of histograms.

Statistics on views Traditionally, optimizers restrict the statistics such as histograms to those built on base tables and derive the cardinality of an expression (e.g., one with joins) bottom-up. However, the errors due to such estimation can be large particularly when the expression is more complex, e.g., containing multiple joins. Statistics on views (SOVs) allow a user or DBA to create a histogram on the result of a relational expression [21, 71]. In the presence of SOVs, the optimizer is not limited to estimating cardinality using only histograms on base tables, but also has access to accurate pre-derived statistics on intermediate relations of the query, which can significantly improve accuracy of CE. The optimizer requires *view matching* technology (briefly described in Section 4.6.4) to identify relevant SOVs when trying to estimate the cardinality of a particular logical expression.

5.3.2 Sketches

A sketch [46] is a summary of the rows of a relation belonging to a domain $D = \{1, 2, ...M\}$, stored as a set of values of a much smaller domain $S = \{1, 2, ...w\}$ called *sketch counters*. Sketches are typically constructed in a single pass over streaming data, i.e., they need to examine each row exactly once. Hence, they can also be used in databases on stored (or intermediate) relations by streaming the data one row at a time. Sketches can be used to estimate properties of the data distribution of the input relation such as: (a) frequency of any given item, (b) heavy hitters, i.e., given a parameter k, finding all items that occur at least N/k times where N is the total number of rows, or (c) the number of distinct values in the relation. For example, the Count-Min sketch [47]

can be used for (a) and (b), and the Hyperloglog sketch [64] can be used for (c).

Since sketches can be used to efficiently estimate frequencies of values and number of distinct values, they are relevant to cardinality estimation. Specifically, they can be used for constructing histograms since they support the required operations such as computing the frequency of values in a range, the number of distinct values over a data distribution, or determining a heavy hitter in an efficient manner. Sketches have also been used for cardinality estimation involving joins (e.g., [104]).

One attractive property of sketches is that they can be configured to use only a small amount of memory (e.g., a few Kilobytes) even for summarizing large relations, while providing good accuracy. During sketch construction, a hash function is applied to each row, and the corresponding hash value is used to update sketch counter(s). The number of counters used can be increased to improve accuracy of estimation by repeating the above step with multiple independent hash functions. Estimating the property of interest (e.g., frequency of an item or number of distinct values) involves simple arithmetic operations on the sketch. Both the update and estimation cost are typically no worse than linear in the number of sketch counters used.

Another attractive property of most sketches is mergeability: For a dataset with multiple partitions, a sketch can be constructed for every partition independently, and the individual sketches computed can be combined without incurring loss in accuracy compared to building the sketch of the same size on the entire data [3]. This property makes parallel and distributed sketch computation practical, and sketches are therefore well-suited for data analytic scenarios where data partitioning is crucial to achieve scalability. For example, a common scenario in data analytics is when a new batch of data is loaded into a new partition, which is then added to the table. In such cases, the need to only compute the sketch on the data in the new partition saves significant computation and I/O since data in the existing partitions do not need to be accessed.

We now briefly review Count-Min and Hyperloglog sketches, which are widely used in database systems.

Count-Min sketch The Count-Min sketch is a space-efficient data structure for estimating the frequency of a given item (i.e., value) in a relation [47]. It has two parameters (ϵ, δ). The guarantee provided by the algorithm on the accuracy of the estimate is that the error is within a factor of ϵ with a probability $(1 - \delta)$. The algorithm uses these parameters to set $w = \lceil \frac{e}{\epsilon} \rceil$ and $d = \lceil \ln \frac{1}{\delta} \rceil$, where d is the number of independent hash functions used by the algorithm, and each hash function $h_j : \{1...n\} \rightarrow \{1...w\}$. The data structure used for Count-Min sketch is a two-dimensional array C, shown in Figure 5.4. In each cell the algorithm stores a counter (initialized to 0).

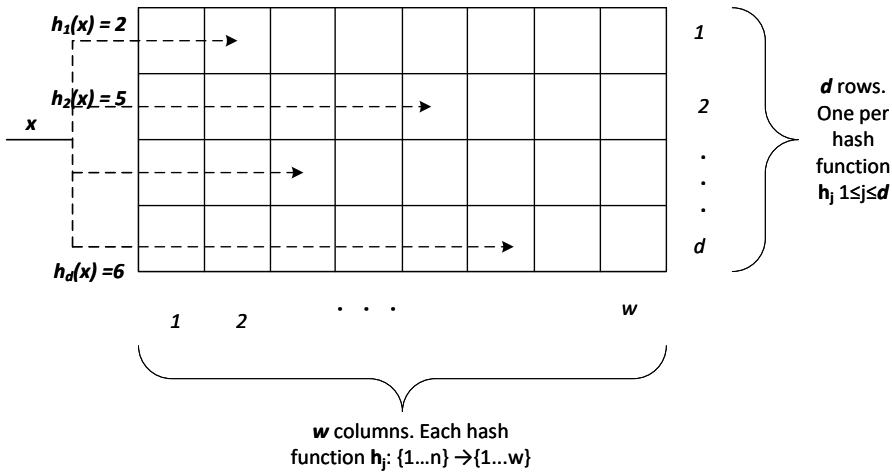


Figure 5.4: Count-Min sketch. Each item is mapped to one cell in the two-dimensional array of counters.

The sketch supports two methods: *increment(x)*, which is called when building the sketch on a column of the row with item x, and *estimate(x)*, which estimates the frequency of item x. In the *increment* method, $\forall j \in [1..d]$, the algorithm increments the array entry $C[j, h_j(x)]$. To find the estimated frequency of an element x, the algorithm returns $\hat{f}_x = min_{j \in [1..d]} C[j, h_j(x)]$. The intuition for using the smallest (i.e., min) value as the estimate of frequency is that it has the least "noise" due to collisions from other items in the data. Thus, the error is the smallest error among all the counters. The formal analysis of the accuracy of Count-Min sketch can be found in [48].

Hyperloglog The Hyperloglog (HLL) sketch is used to estimate the number of distinct values in a relation [64]. HLL uses a hash function that hashes a value in a domain D to the binary domain, $h : D \rightarrow \{0,1\}^L$, where L is the number of bits. The intuition behind HLL is that if the bit pattern $0^\rho 1$ is observed at the beginning of a hash value, then a good estimate of the number of distinct values is 2^ρ, assuming the hash function produces uniform hash values. This is because, if the bits are generated uniformly at random, then the probability that the number will start with 0 is $\frac{1}{2}$, the probability that it will start with 00 is $\frac{1}{4}$, and so on. Therefore, if we encounter a value starting with ρ 0s, then 2^ρ is a good estimate of the number of distinct values encountered in the data stream. However, this estimate has high variance, since a single value that happens to have a large number of leading 0s can lead to an erroneous estimate.

To reduce the large variance that such a single estimate has, a technique known as stochastic averaging [65] is used. Specifically, the input rows are logically divided into m partitions of roughly equal size, using the first p bits of the hash values, where $m = 2^p$. In each partition, the maximum number of leading zeros, after the initial p bits that are used to determine the partition, is measured independently. These numbers are kept in an array M, where $M[i]$ stores the maximum number of leading zeros plus one for the i^{th} partition. The above approach emulates the effect of m experiments and thereby significantly improves accuracy, while using only a *single* hash function and therefore does not increase the cost of hashing.

The algorithm (illustrated in Figure 5.5) initializes a collection of m counters, $M[1], ..., M[m]$, to $-\infty$. As the data from the relation is streamed by, for each value x, it computes $y = h(x)$. The figure shows four different values of $h(x)$ on the left hand side, each corresponding to a different combination of the first p bits. The first p bits ($p = 2$ in our example) of the hash value (say j) are used as an index into the array M. From the rest of the bits (say k), it computes $\rho(k) =$ the position of the leftmost 1-bit in k. The algorithm updates $M[j] = max(M[j], \rho(k))$. To compute the estimate of the number of distinct values, the algorithm takes the harmonic mean of the m estimations on the partitions as

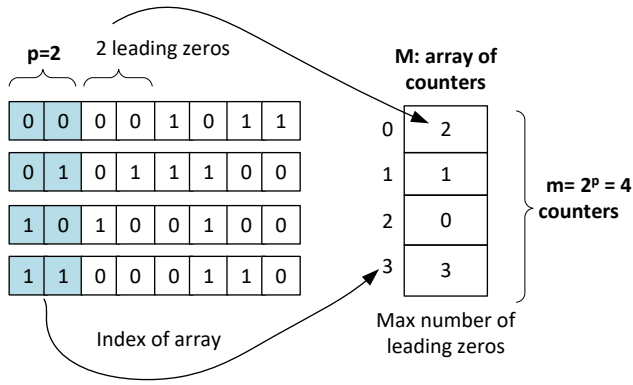


Figure 5.5: Illustration of HyperLogLog algorithm. $h(x)$ produces an 8-bit value. The first $p = 2$ bits of $h(x)$ are used to identify which counter to update. The counter records the maximum number of leading zeros in the remaining 6 bits of $h(x)$.

follows: $\frac{\alpha_m m^2}{\sum_{j=1}^{m} \frac{1}{2^{M[j]}}}$, where α_m is introduced to correct a bias present in the estimation. The details of the algorithm and analysis of its properties are found in [64].

As noted earlier, most sketches are mergeable, provided the hash functions used for all sketches are the same. For the Count-Min sketch, each cell $C[j, h_j(x)]$ of the merged Count-Min sketch is set to the *sum* of the counters of the corresponding cell $C[j, h_j(x)]$ of each of the sketches being merged. For a merged HLL sketch, its $M[j]$ value is set as the *maximum* value of the corresponding $M[j]$ counter across individual sketches.

We note that sketches such as Count-Min sketch and HLL discussed above can produce estimates of distinct values for frequency of values over the entire input. However, they do not have the ability to produce estimates *conditioned* on an arbitrary selection predicate (i.e., filter) over the same input. Developing sketches that can support selections is an active area of research, e.g., [67]. Finally, in database systems, sketches are also used in query execution. For example, DBMSs have recently added new built-in aggregate functions that return the approximate distinct count of an expression implemented using HLL e.g., [6, 143].

5.3.3 Sampling

There are two ways samples are used in query optimization. First, it is used for cardinality estimation. Second, samples are used for constructing histograms. We describes both use cases below.

Use of sampling for CE Unlike statistical summaries such as histograms, wavelets, and sketches, one of the key advantages of using the sample of a table for CE is its broad *coverage* of multiple kinds of predicates. Indeed, cardinality estimates for arbitrary predicates can be derived using a sample by evaluating the predicate on each row in the sample, and scaling the resulting count of qualified rows by a factor equal to the inverse sampling fraction. Thus, sampling can be used to estimate the cardinality of the predicates that can be difficult or impossible to estimate using histograms and sketches, such as LIKE predicates, predicates on user-defined functions (UDFs), and inequality predicates. Some DBMSs support sampling as a method for CE. However, sampling poses significant challenges as well. First, for large tables, samples can consume significantly more memory than histograms. Second, using uniform random samples of base tables for join size estimation can result in large error as shown in [35]. Intuitively, the problem arises from values that are very frequent in one relation but not the other. Thus, sampling from one relation without information about the frequency distribution of values in the other relation can result in large error. Generating a *random sample of the join result* can be expensive; see [35, 155] for such techniques and their limitations. Finally, CE using samples incurs a relatively high latency when compared to histograms since the predicate(s) need to be evaluated on each row in the sample. For these reasons, some commercial DBMSs provide the option of using samples on base tables for CE in limited cases, and it is not used by default. For example, Oracle's Dynamic Sampling feature [156] uses sampling to estimate cardinality when a table referenced in the query has no statistics. SAP Hana [176] supports user-specified hints (see Section 6.3) that direct the optimizer to use sampling for CE.

Use of sampling for histogram construction Sampling can be used for histogram construction, which is supported by all commercial DBMSs today. By using only a sample of the rows in the table, the cost of constructing the histogram can be sharply reduced at the expense of accuracy. One challenge with the use of sampling is that obtaining a uniform random sample of rows from a table requires a full scan of all rows in the table, thereby incurring a high I/O cost for large tables. Therefore, most DBMSs obtain a random sample of *blocks* (i.e., pages) of the table from the storage subsystem, and use all rows from that page as part of the sample. While this significantly reduces the I/O cost of sampling, block sampling does not yield a uniform random sample of the rows in the table. This is because rows within a single page may be correlated. For example, all the rows in a page may be from the records of one state because they were bulk inserted into the table. Techniques for mitigating the loss in accuracy while retaining most of the efficiency of block sampling are presented in [30, 34]. Finally, we note that efficient techniques for incrementally maintaining a uniform random sample of the rows in a table as the data changes due to updates are described in [74].

5.3.4 Statistics management

Thus far, we have introduced statistics on tables and views and discussed how they are used for cardinality estimation of a query. Two major challenges appear in managing statistics over the lifetime of the database. First, the DBMS needs to decide which statistics to create. Second, when the data changes, statistics need to be maintained (i.e., updated) to ensure that accuracy of CE does not degrade. Thus, in order to keep the cost of creating and maintaining statistics acceptable, it is important to be judicious in deciding which statistics to create.

Deciding which statistics to create Given a database, the space of single and multi-column statistics that are potentially relevant is large. Thus, it can be prohibitively expensive to create and maintain all potentially relevant statistics. By default, most commercial DBMSs create single-column statistics on each column referenced in the query

occurring in a selection or join predicate, GROUP BY clause, ORDER BY clause, etc. Furthermore, for efficiency of creation, particularly on large tables, they create statistics using a sample of rows in the table (e.g., Microsoft [146] and Oracle [160]). While this approach is efficient, a key limitation is that it fails to consider the space of multi-column statistics and statistics on views (see Section 5.3.1). Below we briefly describe two techniques proposed in the research literature for deciding which statistics to create in the presence of multi-column statistics and statistics on views. Intuitively, these techniques identify unimportant statistics from the large space of potentially relevant statistics by leveraging the fact that not all statistics are equally useful for a query. Consider the following Query 29:

Query 29

```
SELECT i_color, COUNT(*)
FROM item
WHERE i_category = 'Dress'
GROUP BY i_color
```

First, statistics creation can be limited to columns referenced in the query. In our example query, only statistics created on *item* table involving *i_category* and *i_color* columns can be used for CE, and therefore affect the cost of plans. Second, some statistics are equivalent with respect to a query. For example, if we create a single column statistics on *i_category* and another multi-column statistics on *(i_category, i_color)*, these two statistics will derive the same selectivity for $i\_category =' Dress'$ on the *item* table since the histograms on *i_category* would be the same.[2] Therefore, creating both statistics would be redundant for this query. Finally, while some statistics are syntactically relevant to a query, their impact on the plan quality for the query can be limited. Our example query involves accessing the *item* table followed by a group-by aggregate. If the cost of accessing the *item* table dominates the cost of the query plan, different choices of the physical operator for the group-by expression based on the distinct value estimate will have limited impact on the cost of the plan.

[2] If the statistics are created using a full scan of the data.

Prior work by Chaudhuri *et al.* [36] formulates the problem of statistics selection for a workload and proposes techniques to identify the set of essential statistics that will significantly impact the quality of the plans of the workload. The challenge is to quantify the impact of the statistics on the cost of a plan *without* creating the statistics. This work has two key ideas. First, they prioritize the statistics that will syntactically impact the expensive operators in a query plan. Second, they develop a technique called Magic Number Sensitivity Analysis (MNSA) to quantify the impact of a statistics on the plan cost *without actually creating* the statistics. Specifically, MNSA injects extreme selectivity values, close to 0 and 1, for predicates that use a statistic being considered for creation, and compares the two plans thus generated by the optimizer. If the plans are same, then the statistic is considered not important, and its creation can be avoided. For example, in the above query if injecting extreme selectivity values for the predicate $i\_category =' Dress'$ does not affect the plan chosen by the optimizer, then creating statistics on the column $i\_category$ can be skipped.

The problem of deciding which *statistics on views* (see Section 5.3) to create was studied in Bruno *et al.* [21]. They present a non-trivial generalization of the MNSA technique by identifying distributions of the join column on a table *after selections are applied* that can lead to "extreme" (i.e., smallest or largest) joins sizes. For example, consider the following Query 30:

Query 30

```
SELECT *
FROM store, store_sales
WHERE s_store_sk = ss_store_sk AND ss_price < 100
```

By default, when estimating cardinality of the above query, optimizers use the independence assumption and scale down each bucket of the histogram on the column ss_store_sk by the selectivity of the predicate $ss\_price < 100$. This modified histogram, along with the histogram on the column s_store_sk is used to estimate the join size as described in Section 5.4.2. However, instead of uniformly reducing the frequency of all tuples in the histogram ss_store_sk, the above technique adjusts the number of rows in each bucket such that the join

size is either the smallest or largest possible under the containment assumption (see Section 5.4.2). The difference in cost between these extreme cardinalities forms the basis for scoring the importance of a statistic on view, and this score is used to rank the candidate statistics on views to create.

Maintaining statistics Another challenge in statistics management is how to maintain existing statistics. When a table is updated, the underlying data distribution can change, and the statistics can become obsolete over time. Despite prior work on maintaining histograms upon data updates [75], in most commercial database systems, statistics on a column are reconstructed when a significant fraction of the rows in that column have been updated since the last time the statistics on that column were created. They maintain counters that accumulate the number of rows updated for each column. To keep the I/O overheads of such counting low, most DBMSs accumulate these counters in memory and only persist them when the database is checkpointed.

5.4 Cardinality Estimation

The cardinality estimation (*CE*) module in the query optimizer takes a logical query expression as input and returns an estimate of the number of rows output by that expression. In this section, we first describe the requirements and key challenges in CE and then describe the corresponding techniques.

5.4.1 Requirements and challenges

Efficiency The search algorithm of the query optimizer makes repeated calls to the cardinality estimation module during query optimization. As described in Section 2, in a query optimizer based on the Volcano/Cascades framework, for each group in the memo, the search algorithm calls the CE module to estimate the cardinality for that group. It is common for the CE module to be called 100s or 1000s of times during query optimization of a single query, particularly for large and complex queries. Therefore, CE needs to be efficient to limit the

total time and resources incurred by query optimization. To keep the CPU and memory cost of CE acceptable, the statistical information used for CE needs to be compact and the estimation techniques need to be computationally efficient.

Handling correlation and skew The cardinality of an expression depends on the data distribution, which can be skewed and correlated. For example, in Figure 5.2, the number of rows satisfying the predicate $s\_state =' CA'$ from the *store* table can be very different than a predicate $s\_state =' VA'$ due to the data skew that occurs commonly in real-world data sets. Hence CE must be able to adequately capture the effects of data skew. Moreover, if the expression requires satisfying a conjunction of two conditions, e.g., $ss\_year = 2000$ AND $ss\_sales\_price > 10$, then the effects of *correlation* between the data in the two columns may need to be captured to provide accurate CE. In this example, if most items sold in the year 2000 are priced less than 10, i.e., there is a strong *negative* correlation between the year and the specific price range of the items, then ignoring the correlation, i.e., assuming independence, may result in a significant overestimation of cardinality even if the CE of the individual predicates is accurate.

Handling complexity of SQL Even when the logical query expression for which we are estimating selectivity contains only join operators, the errors in join CE can propagate exponentially with the number of joins in the expression [102]. Furthermore, operators such as group-by, DISTINCT, and UNION require the optimizer to estimate the *number of distinct values* in a relation (either on base tables or intermediate relations). Accurately estimating cardinality of an expression with the constraints of limited statistical information and fast estimation makes CE challenging.

5.4.2 Key techniques

While the optimizer needs to estimate the cardinality for a variety of logical expressions, most prior work as well as emphasis in commercial DBMSs focuses on CE for three classes of logical expressions: selection

conditions (i.e., filters) on a single table, join size estimation, and distinct value estimation. These classes are important because they appear frequently in real-world queries. Moreover, these techniques can also be used as building blocks to estimate the cardinality of more complex expressions involving multiple joins, filters, group-bys, etc.

There is a rich body of literature on cardinality estimation techniques, which can be categorized by the kind of statistical summaries used on the base tables. In describing the techniques below we will assume the use of histograms and HyperLogLog sketches.

It is important to mention that due to the constraints noted earlier: limited statistical information about the data, need for fast estimation with limited resources, and complexity of the SQL language, CE techniques resort to various simplifying assumptions. Here, we discuss these assumptions and the implied limitations of CE techniques.

Selection conditions on a single table

The most common classes of selection conditions or filters are point predicates (e.g., $s\_state =' CA'$), IN clauses (e.g., s_state IN $('CA','VA')$), one-sided and two-sided range predicates (e.g., $ss\_sales\_price > 10$, ss_sales_price BETWEEN 5 AND 10), and predicates on string data with filters by prefix (e.g., $s\_manager$ LIKE $'ste\%'$) and substrings (e.g., $s\_manager$ LIKE $'\%ste\%'$). For cardinality estimation of selection conditions on a single table, the commonly used simplifying assumptions include *uniformity* and *independence*. We will explain these assumptions and how to use them in the presence of single selection condition and complex selection conditions as below.

Single selection condition Consider a single equality selection condition, e.g., $s\_state =' CA'$. In the absence of any statistical information about the data values in the column s_state, the CE module would need to make an assumption of how the values in the column are distributed. In such cases, query optimizers often make the *uniformity* assumption, i.e., they assume the data is uniformly distributed across all the values. For example, if the size of a table is $|T|$, and the number of distinct values in the column is n, then by uniformity, each value occurs in $|T|/n$

rows, i.e., the selectivity is estimated as $1/n$. Unfortunately, this could incur large errors in the presence of non-uniform data distribution in the column, e.g., CA could have many more stores than the average value across all states. When a histogram on the *s_state* column is available, depending on the type of histogram used, more accurate CE is generally possible (see Section 5.3.1), although *within* a bucket the uniformity assumption is still used across all distinct values that fall into that bucket.

Finally, we observe that for predicates on string columns that require estimating the number of matching substrings (e.g., *s_manager* LIKE $'\%ste\%'$), traditional histograms are insufficient. In this case, different data structures such as suffix trees [112] or tries are typically used for CE.

Conjunction of two or more selection conditions Consider the conjunction of two predicates $ss\_year = 2000$ AND $ss\_sales\_price > 10$. While the selectivities of the individual predicates can be estimated using the techniques described above, estimating the selectivity of the conjunction of two predicates is more challenging since the data in the two columns may be correlated. Thus, in the absence of data statistics beyond those available with single-column histograms, i.e., *ss_year* and *ss_sales_price*, the optimizer needs to make an assumption of how the data is correlated. In practice, multi-dimensional histograms are not commonly used due to the high costs of creation and maintenance. Therefore, a common assumption made by optimizers in such cases is *independence*. This implies that the selectivity of the conjunction of the two predicates is estimated as the product of the selectivity of the individual predicates as follows:

$$Sel(p_1 \wedge p_2) = Sel(p_1) \times Sel(p_2) \tag{5.1}$$

For example, in the *store_sales* table, if the selectivity $ss\_year = 2000$ is 0.1, and selectivity of $ss\_sales\_price > 10$ is 0.2, then using the *independence* assumption, the selectivity of their conjunction, i.e., items sold in year 2000 with a price more than 10, is estimated as 0.02.

Join size estimation

Join size estimation, especially for equi-joins and semi-joins, is another crucial task for query optimizers.

Estimating the cardinality of a join using histograms on the join columns involves three steps. First, the buckets of the histogram are *aligned* so that their boundaries agree, which might require splitting some buckets. Second, there is a per bucket estimation of join sizes. A common assumption made in the context of estimating join size is *containment* [178]. Let R and S be two relations, where each of them is grouped by the values of the join column(s) respectively, and R has a smaller number of groups than that of S. Then the containment assumes that for each group g_R of rows in R, it has a corresponding group g_S in S, where each row from g_R joins with the rows from g_S in S. Thus, the groups in S *contain* those from R.

Consider the example Query 31:

Query 31

```
SELECT s_store_sk
FROM store, store_sales
WHERE s_store_sk = ss_store_sk
```

Suppose a bucket in the histogram of *store* has 10 rows with 5 distinct values, and the corresponding bucket from *store_sales* has 200 rows with 2 distinct values. Then, each group of the bucket in *store_sales* (with $200/2 = 100$ rows each) is assumed to join with a group of rows from *store* (with $10/5 = 2$ rows each). Hence, the total number of rows joining between these buckets would be $2 \times 100 \times 2 = 400$ rows. More generally, if n_1 and n_2 are the frequencies of the two buckets, and d_1 and d_2 are the number of distinct values in each bucket, then the formula: $\frac{n_1 \times n_2}{max(d_1, d_2)}$ can be used to calculate the join size under the containment assumption. We observe that the output of joining two buckets can be viewed as a new bucket of the histogram of the resulting join, i.e., a histogram of an expression can be constructed from the histogram(s) of its input(s) for CE. In our example, the output bucket would contain 2 distinct values each occurring $200 = 100 \times 2$ times. The last step consists of aggregating the partial frequencies from joining each pair of buckets to get the estimated cardinality for the whole join.

Figure 5.6 illustrates the bottom-up propagation of histograms for the cardinality estimation of a join query with selections.

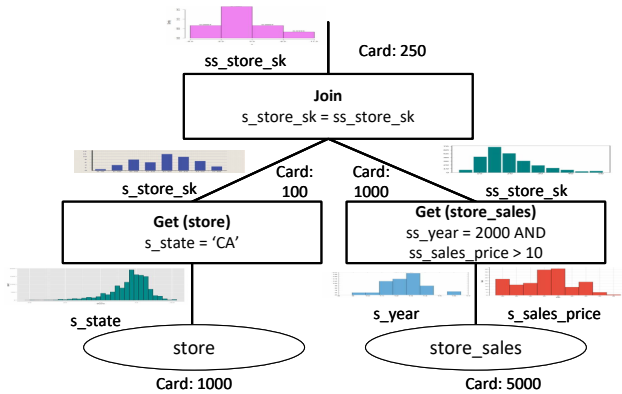


Figure 5.6: Example of join size estimation. To estimate the cardinality of the Join operator shown in the figure, the histograms of join columns *s_store_sk* and *ss_store_sk* are modified to reflect the impact of selection conditions. The CE module estimates the selectivity of the Join and also constructs a propagated histogram for the column *ss_store_sk* of the join expression.

In the presence of selection predicates, (e.g., suppose there is also a selection predicate $ss\_price < 100$ in Query 31), two variants for the above technique are possible. One approach is to estimate the selectivity of the other selection predicate (e.g., say it is 0.1), and scale down the frequency of each bucket in the histogram of the join column *ss_store_sk* by that selectivity. Then, the same procedure as above is performed but with the scaled-down histogram of *ss_store_sk*. The second approach is to first join the two histograms on *s_store_sk* and *ss_store_sk* as described above, and then apply the selectivity of the selection condition to the propagated histogram to scale down its frequency. We observe that these two methods can result in different cardinality estimates.

Distinct value estimation

When a logical query expression contains GROUP BY, DISTINCT, or UNION operators, estimating the number of distinct values in the column (or set of columns) of the expression becomes necessary. For

example, consider the following Query 32, where the cardinality of the logical expression of the query is the distinct number of values of the *ss_year* column:

Query 32

```
SELECT ss_year, count(*)
FROM store_sales
GROUP BY ss_year
```

One approach used by query optimizers for distinct value estimation is to track the number of distinct values in a column as part of the histogram, and use that information for CE of a group-by expression. Since computing the exact number of distinct values requires a full scan of the data and therefore can be expensive, DBMSs provide the option to use random sampling to estimate the number of distinct values. While sampling can substantially reduce the cost of histogram construction, it has been shown that any estimator that examines at most n rows of a table of size N must incur an expected ratio error of $\mathcal{O}(\sqrt{N/n})$ on some input [27]. Finally, we note that when the GROUP BY contains multiple columns, query optimizers leverage multi-column statistics stored in the histogram, i.e., number of distinct values for the combination of columns, for estimation. When such multi-column statistics are unavailable, they resort to assumptions on how the set of columns are correlated. For example, if the optimizer assumes independence among the columns, then it may estimate the total number of distinct values as $min(N, d_1 \times ...d_k)$, where N is the total number of rows in the relation, and d_i is the estimated number of distinct values in the i^{th} column of the group-by.

More recently, as discussed in Section 5.3.2, techniques based on *sketches*, specifically HyperLogLog (HLL) are gaining traction for their use in estimating the number of distinct values. HLL requires a full scan over each row in the relation. However, unlike sampling, they use only a small amount of memory (usually measured in KBs), and they are accurate, i.e., their relative accuracy (technically the standard error) is $\frac{1.04}{\sqrt{m}}$ where m is the memory used [64], computationally efficient, and mergeable, i.e., HLLs computed on each partition of the data can be combined easily to estimate the number of distinct values for the entire

relation. One drawback of HLLs with respect to CE is that the accuracy of estimation of number of distinct values can degrade in the presence of arbitrary selection conditions on the relation.

5.4.3 Status and limitations

Given the fundamental nature of the challenges in CE as noted earlier, it is not surprising that CE remains one of the biggest sources of error in cost estimation [29, 122]. Despite extensive research over the past few decades, the state-of-the-art in cardinality estimation in commercial database systems has not changed significantly over this period.

There have been empirical studies that attempt to quantify the impact of CE errors on plan quality, i.e., elapsed time and resource consumption of executing the plan. An empirical study [118, 119] of industrial-strength cardinality estimation techniques on synthetically generated select-project-join (SPJ) queries on the Internet Movie DataBase (IMDB) demonstrated that all estimators routinely produce large errors that lead to significantly sub-optimal ordering of joins. A more recent empirical study [116] quantifies the impact of cardinality estimation on plan quality in Microsoft SQL Server, an optimizer based on the Cascades framework. This study focuses not only on synthetic queries used in prior research, but also includes industry benchmarks such as TPC-H [167], TPC-DS [149], and DSB [57], as well as complex real-world queries with join, group-by, aggregation, and nested subqueries. The study shows that significant improvements in plan quality are possible across workloads if cardinality errors can be fixed. They find that these improvements hold even in the presence of query runtime techniques such as bitvector filtering [136] and adaptive joins [135] that are designed to mitigate the negative impact of CE errors on query performance.

The above studies suggest that despite advances in runtime query execution, techniques in DBMSs cardinality estimation remains an important problem that is worthy of attention.

5.5 Case Study: Cost Estimation in Microsoft SQL Server

5.5.1 Cost model

In Microsoft SQL Server, the cost of a query plan is a scalar that captures the estimated resource consumption such as CPU time, memory and disk I/O [145]. The cost refers to the estimated elapsed time, in seconds, that would be required to complete a query on a specific hardware configuration for which it was calibrated.

I/O and CPU cost The Microsoft SQL Server optimizer's cost model is a combination of the estimated CPU cost and estimated I/O cost. The I/O cost for an operator when reading (or writing) to a disk (or SSD) takes into consideration the number of pages that need to be read or written and whether the I/Os are *sequential*, e.g., for an Index Scan operator, or *random*, e.g., for Index Seek operator, as random I/Os can be slower than sequential I/Os. The CPU cost of an operator takes into account several factors such as: (a) the number of rows (both input and output) to be processed, (b) the columns used and their sizes (c) the kind of predicates used in a Filter or Join operator, e.g., equality predicate vs. range predicate vs. LIKE predicates. Different constants are used to model the cost of each kind of CPU operation on a row, which are then scaled by the number of rows to obtain the total cost.

Impact of memory and parallelism We observe that the CPU and I/O costs of an operator also depend on the resources available such as memory and the number of CPU threads used to execute the operator. The latter is also referred to as the degree of parallelism (DOP) (see Section 2.6.1). For example, for a Hash Aggregate operator, if the estimated number of entries in the hash table does not fit into the memory that is available to the operator, then the optimizer's cost model adds I/O cost for spilling rows to disk and reading them back to execute the operator. In Microsoft SQL Server, the cost of exchanging data across threads in a parallel plan is modeled by an Exchange operator which models different kinds of producer-consumer settings (see Section 2.6.1). Consider a serial, i.e., single threaded, Hash Aggregate

operator whose input is a *parallel* Table Scan operator. In this case, there are multiple *producer* threads and a single *consumer* thread. The Exchange operator gathers input rows into a buffer, which is then consumed by the single thread performing the aggregation. This additional cost incurred by the plan due to the overheads of parallelism is captured by the cost model of the Exchange operator. Other cases covered by the Exchange operator include single-producer multiple-consumers and multiple-producers multiple-consumers. We observe that while the total *CPU cost* of a plan with parallelism is higher than the CPU cost of the serial plan due to the above overheads, the *elapsed time* of the query plan could be lower since the work is done in parallel.

Tracking multiple costs Microsoft SQL Server estimates multiple different costs for each operator. For example, besides the total cost of the operator, i.e., cost to return all output rows, it also estimates the cost to return the *first row*. The latter is important for costing operators such as Nested Loops Join (NLJ). For example, consider when the inner side of the NLJ operator is an Index Seek operator. Each distinct value of the join column of the outer input to the NLJ operator is a new binding (i.e., argument) to the Index Seek operator, and hence the results need to be computed for each binding. However, if there is a duplicate value of the binding, then the results of the inner side can be reused by caching the result, and simply rewinding to the start of the result of the inner. The first row cost estimates the cost of performing the Index Seek for a new binding value, which can be different than the cost of returning the cached rows. Such first row cost tracking is also used for operators such as Spool and Table Valued Function.

Costing with row goals Microsoft SQL Server also takes into account *row goals* which can affect the total cost of an operator (and hence query plan). This is important for queries with a *TOP K* clause. Consider a query that requires TOP 1 row from a table that satisfies a given predicate. If a suitable index exists, a single matching row can be retrieved with a single I/O using an Index Seek operator. In contrast, using a Table Scan on the same table, many more pages may need to be scanned before the first matching row is found. However, when K

is large, using a Table Scan may be cheaper than using an Index Seek due to significantly higher cost of random seeks compared to a scan. When computing cost for an operator, the row goal is passed in as a parameter, and the total cost of the plan reflects the cheapest cost of obtaining the given row goal target.

5.5.2 Statistics

We give a brief overview of statistics used by Microsoft SQL Server for cardinality estimation. The application or the DBA can create a statistics object on one or more columns of a table. A statistics object consists of a histogram on the leading column of the statistics and a *density* vector (explained below). For example, the following command creates a multi-column statistics on columns (A, B) of table T:

```
CREATE STATISTICS statAB on T (A, B)
```

We explain the statistical information stored by Microsoft SQL Server using the example shown in Table 5.1.

Table 5.1: Frequency of each value in column

Value	Frequency
AL	7
AZ	8
CA	40
DE	12
MA	18
NY	33
RI	4
TX	37
VT	2
WA	15

Histogram Microsoft SQL Server creates a MaxDiff histogram on the leading column of the statistics. We illustrate this using data shown in Table 5.1. The corresponding MaxDiff histogram with 4 buckets is shown in Figure 5.7. The values shown in bold in the figure are actually stored in the histogram. Microsoft SQL Server associates with each bucket: (1) a value corresponding to the upper bound column value

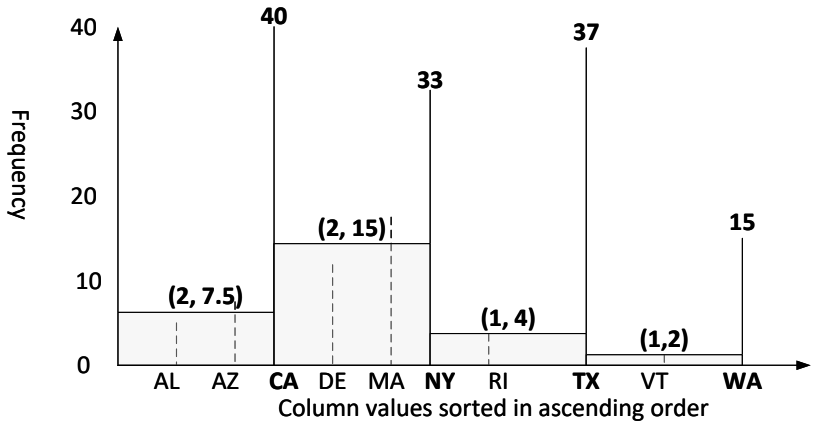


Figure 5.7: Example histogram for column shown in Table 5.1

represented by that bucket and its frequency. For example, in the first bucket, the upper bound column value is CA, with a frequency of 40. (2) The number of distinct values in the bucket, not including the upper bound column value, and the average number of rows per distinct value in that bucket. In the example, the first bucket has two values AL, AZ with a total frequency of 15, hence the values (2, 7.5) are stored. Observe that the column values AL and AZ are not stored, nor are their individual frequencies. The MaxDiff histogram creation algorithm chooses bucket boundaries where the difference in frequencies of adjacent values is maximized, while ensuring that the number of buckets does not exceed the given limit of buckets. Histograms in Microsoft SQL Server use a maximum of 200 buckets. This value is chosen to keep the cost of statistics loading as well as the cost of cardinality estimation using histograms acceptable.

Density The density for a set of columns is a single number $\frac{1}{d}$, where d is the number of distinct values in that set of columns. For the sequence of columns specified in a CREATE STATISTICS command, Microsoft SQL Server computes the density for each of the leading prefixes of the columns. For example, when creating a statistics object on columns (A, B) of table T, Microsoft SQL Server computes density information on (A) and (A, B). Density is used for estimating the number of distinct

values, which is necessary for CE of different types of logical expressions, e.g., group-by, DISTINCT, UNION.

Statistics management Microsoft SQL Server automatically creates and maintains statistics as data changes [146]. When a query executes and the statistics needed by the optimizer is not available, the database engine automatically creates statistics. Specifically, Microsoft SQL Server automatically creates a single-column statistics on any column referenced in the query for which statistics are not already available. For large tables, the statistics are created on a sample of the data from the table for efficiency, whereas small tables are inspected in their entirety. Additionally, statistics creation is performed asynchronously in a background thread so as to not delay the execution of the query that triggers the statistics creation. Once a statistics is created on a table, it bumps up the metadata version of the table. Hence, the next execution of the query triggers an invocation of the query optimizer which generates a plan using the newly created statistics. Statistics are updated as described in Section 5.3.4 by tracking the number of rows updated for the column(s) of the statistics, and refreshing when either the absolute number or fraction of rows exceeds certain thresholds.

5.5.3 Cardinality estimation

Microsoft SQL Server follows the techniques described in Section 5.4 for cardinality estimation. Starting with Microsoft SQL Server 7.0, the optimizer uses assumptions such as: (1) Independence: Data distributions on different columns are assumed to be independent of each other, unless correlation information (e.g., density) is available. (2) Uniformity: Within each histogram step, distinct values are evenly spread and every value has the same frequency. (3) Simple containment: For example, for an equality join between two tables, it factors in the predicates selectivity in each input histogram by scaling down each bucket, before joining the scaled-down histograms to estimate the join selectivity. For more details on CE in Microsoft SQL Server, we refer the reader to Microsoft [139].

Microsoft SQL Server made a major update to its cardinality estimator in 2014 [137] based on the aggregated benefit observed over several OLTP and data warehousing workloads. In particular, it made two major changes to the assumptions. First, instead of assuming independence, it assumes that data distributions on different columns are positively correlated. However, instead of assuming perfect correlation, it assumes a so-called "exponential back-off" model of correlation, which strikes a middle-ground while still keeping the overheads of statistics low. Specifically, when estimating the selectivity of a conjunction of predicates, the most selective predicate's selectivity is multiplied by the square root of the selectivity of the second most selective predicate, which is multiplied by the square root of the square root of the selectivity of the third most selective predicate, and so on, as shown in the formula below:

$$Sel(p_1 \wedge p_2 \wedge p_3) = Sel(p_1) \times Sel(p_2)^{1/2} \times Sel(p_3)^{1/4} \qquad (5.2)$$

Second, the simple containment assumption is modified to assume that the filter predicates on two tables being joined are not correlated with one another. With the new assumption, called *base containment*, the join selectivity is estimated with the original histograms, i.e., they are not scaled down as in simple containment.

While the new cardinality estimator in Microsoft SQL Server is more accurate in many cases, the earlier assumptions of independence and simple containment may work better for a specific query. Thus, Microsoft SQL Server provides a query hint for users to choose between the new cardinality estimator or the legacy estimator. We refer the reader to Section 6.3 for a broader discussion on query hints. They also provide more fine-grained hints that allow users to choose specific assumptions used for a query, e.g., use independence for filter predicates, use simple containment for join selectivity estimation. This allows users to mix and match the assumptions to improve plan quality in cases where the default assumptions of the optimizer do not produce sufficiently good plans.

5.6 Suggested Reading

Citation numbers below correspond to numbers in the References section.

[101] Y. Ioannidis, "The History of Histograms (abridged)," in *Proceedings 2003 VLDB Conference*, Elsevier, pp. 19–30, 2003

[46] G. Cormode *et al.*, "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches," *Foundations and Trends® in Databases*, vol. 4, no. 1–3, 2011, pp. 1–294

[92] S. Heule *et al.*, "Hyperloglog in Practice: Algorithmic Engineering of a state of the art Cardinality Estimation Algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 683–692, 2013

[119] V. Leis *et al.*, "Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark," *The VLDB Journal*, vol. 27, 2018, pp. 643–668

[116] K. Lee *et al.*, "Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, 2023, pp. 2871–2883

6

Plan Management

In previous sections, we have focused on how the query optimizer generates a plan for a given query. Since query optimization can be computationally expensive, the generated plans are cached for efficiency. In this section, we first discuss *plan caching* and their validation. We then describe different approaches to overcome the disadvantages of optimizer generated plans that are not performant. We next discuss query hints, which can influence the query optimizer in its plan choice. Finally, we cover the techniques used to optimize *parameterized queries* that are widely used in practice but for whom generating a plan for every query instance can be costly.

6.1 Plan Caching and Invalidation

The computational cost of generating a plan by the query optimizer can be significant. Since DBMSs need to support high query throughput, they need to manage the caching and reuse of plans while ensuring that the plan quality does not degrade when data gets updated. The overall workflow of plan caching and reuse within a DBMS is shown in Figure 6.1. Below, we briefly outline how DBMSs address two important challenges that arise due to the caching of plans.

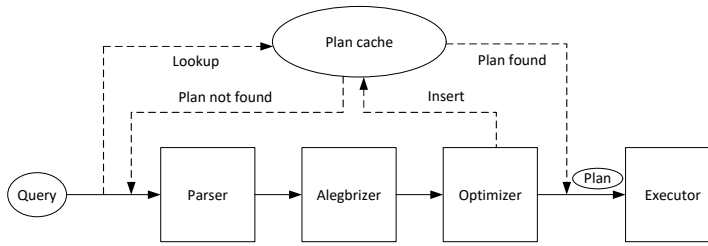


Figure 6.1: Plan caching and reuse

First, for applications that issue many different queries, the plan cache can consume a large amount of memory. Parameterized queries, where multiple query plans may be generated, further exacerbate the situation (see Section 6.4). Thus, the plan cache needs to be managed to handle memory pressure similar to other caches in the database systems such as the buffer pool. The eviction policy of the plans from the plan cache can take the following information into consideration: (1) The cost of re-generating the plan if it were to be evicted from the cache. The proxy used to approximate this cost is the CPU time taken to generate the plan. (2) Usage information of the plan, including the number of times the plan has been executed since it was generated.

Second, when the data is modified due to INSERT, UPDATE, or DELETE statements in the workload, the changes in data distribution may make the cached plan for the query to become sub-optimal. In general, efficiently identifying such cached sub-optimal plans is challenging. In practice, most DBMSs resort to simple heuristics to address this challenge. For example, for each table, they track the fraction of rows that have been updated since the last time statistics on that table were computed (see Section 5.3.4). If the fraction of rows updated exceeds a threshold (e.g., 0.1), the database system can rebuild the statistics on that table, and invalidate all cached plans that reference that table. Subsequently, when a query which references that table is processed, a plan is generated using the new statistics, and the newly generated plan is then cached. We note that a cached plan may also need to be invalidated due to DDL statements that change a table's metadata, such as the creation of a new index or the removal of a column.

6.2 Improving Sub-optimal Plans with Execution Feedback

The query optimizer chooses a plan based on a *model* of the cost of executing a plan. The cost model only approximately captures the *actual* cost of executing the plan, and the inputs to the cost model such as cardinality estimates may be inaccurate as well (see Section 5). The optimizer may also use heuristics, such as timeouts, during plan search for efficiency (see Section 2). Finally, at execution time, due to resource demands of other concurrently executing queries, the plan may not receive all the resources that the optimizer had assumed when generating the plan, e.g., insufficient memory for a Sort operator can lead to spill, which can increase execution time. Therefore, in practice, the optimizer sometimes may choose a plan that is sub-optimal, i.e., the elapsed time taken to execute the plan is too long. To deal with the issue of the optimizer picking a sub-optimal plan in terms of actual execution cost, database systems have developed a variety of mitigation techniques. We briefly review these techniques, ranging from simple heuristics that are relatively easy to implement to more involved solutions that require changes to the query execution engine.

Handling regressions due to plan change A special case of sub-optimality occurs when a plan is changed, and it executes much more inefficiently than in the past. Such a change in query plan can happen due to several reasons. One of the most common reasons is due to the update of statistics triggered by changes to the data (see Section 5.3). Other reasons include DDL statements on the table, e.g., creation of an index on columns referenced by the query. Such changes cause the DBMS to generate a new plan for the query. Whenever a new plan is used to execute the query, it is possible that the actual query performance can become worse after the plan change. In such cases, a simple strategy to remedy the regression is to revert to a *previously* used plan that is historically known to have better performance than the new plan, if that historical plan continues to be valid.

Microsoft SQL Server has developed such a technique called Automatic Plan Correction (*APRC*) [147]. The database system uses a persistent repository called Query Store [142] to cache historical plans

of a query and monitors the performance of the query to detect plan changes and regressions. Specifically, for each plan, it stores aggregate historical usage and performance counters such as number of executions, average/min/max/standard deviation of elapsed time and CPU time. Regression in a plan's performance is detected by running a statistical test on the plan's performance metrics in comparison to those of the last known plan for that query. If a plan regresses in performance, the database falls back to the last known plan that is still valid for the query.[1] After reverting to the fallback plan, APRC monitors the performance of the fallback plan. If it is no better than the plan that was deemed to have regressed, APRC forces the optimizer to generate a new plan again. Otherwise, the fallback plan is used to execute the query until the next time a new plan is generated, e.g., due to changes in statistics or database schema. Finally, we note that the above approach of using Query Store to collect historical information of actual resource usage of a plan can be used to also automatically mitigate other issues with plan quality. For example, when the amount of working memory to grant for operators such as Hash and Sort is insufficient, it can lead to spills, thereby increasing the cost of executing the query. Microsoft SQL Server's *memory grant feedback* feature [144] can automatically adjust memory grants to use for a query plan based on analyzing the estimated and actual memory usage in historical executions of the query.

Oracle database has a technique called SQL Plan Management (*SPM*) that aims to avoid query regressions [162]. It maintains a set of valid plans that the optimizer is allowed to use for a SQL statement, referred to as *baseline plans*. Whenever the optimizer generates a new plan for the query that is not in the baseline, SPM only adds the new plan to the existing set of baseline plans if the performance of the new plan is verified through execution to be comparable or better than the baselines. Thus, SPM ensures that only a plan whose performance is verified can be used to execute a query. For each query, the database selects a plan from the set of baseline plans to use for executing the query, usually the one with the lowest cost.

[1] A previously executed plan may no longer be valid if the database schema has changed, e.g., an index used in that plan was subsequently dropped.

Change of physical operators at runtime The Adaptive Join feature in Microsoft SQL Server [135] and Oracle DB [161] can switch from a Hash Join to Nested Loops Join at runtime. Specifically, Adaptive Join reads the input rows for the build side of the Hash Join. If the first input is fully consumed, and the number of rows is below a predetermined value chosen by the query optimizer, the Adaptive Join switches to using a Nested Loops Join operator. This strategy prevents a bad choice of the join operator by the optimizer by deferring the choice of Hash Join or Nested Loop Join until the first input is consumed. The overhead of this approach includes the extra memory and computational cost of buffering input rows.

Competition for access path selection One of the early work was the technique of plan competition, which was implemented in Oracle Rdb [7]. Plan competition consists of two phases. In the first phase, one or more index-based access paths are executed given a small, predetermined amount of time. If one of the access paths completes within the time, the corresponding plan is picked for execution; otherwise, we enter the second phase, where one among all available access methods (including Table Scan) is selected to execute the query. For example, if one of the access methods available is an Index Seek that retrieves rows matching the equality predicate *Country=@p1*, where the distribution of the *Country* column is skewed. For most countries, the Index Seek plan would complete within the first phase because it needs to retrieve only a few rows. If the Index Seek completes in the first phase, then the Table Scan access method does not need to be used. However, for those countries with large number of rows, the Index Seek plan may not complete within the predetermined amount of time. In such cases, in the second phase, one of the two plans is picked and executed. Observe that if the Table Scan operator is picked to run in the second phase, then the Index Seek operator would need to be canceled, and any rows it may have retrieved are discarded. Therefore, although redundant work may be incurred for the query, it can prevent the scenario where a poor plan executes for too long. In general, in the first phase, more than one access method may be executed concurrently. For example, a second Index Seek on a different index, or an Index Intersection of

two indexes may be executed. Finally, although we have introduced competition in the context of choosing between access paths, in principle, the idea of competition can be extended to sub-plans. However, this can substantially increase the overheads incurred due to wasted work, and hence is more challenging for adoption in practice.

Choose-Plan operator with runtime costing As the name suggests, the idea is to choose *at runtime* among a pre-determined set of plans and execute one of them. *Choose-Plan*, introduced in [45], is a "meta-operator" that contains the decision procedure to evaluate the cost of multiple children plans and pick the one with the lower cost at execution time, i.e., the plans are costed at runtime. Figure 6.2 shows an example plan with a Choose-Plan operator. Such an approach is beneficial when the costs of the children plans cannot be compared at query optimization time, e.g., because the parameter bindings are only known at runtime or the available memory at runtime is different than the optimizer had assumed. The concept of Choose-Plan was introduced in the context of the Volcano optimizer framework, and we refer the readers to [45] for more details on how to modify the dynamic programming based search algorithm to integrate the Choose-Plan meta-operator. We note that the decision of where to place the Choose-Plan operator(s) in a plan is important, and this problem requires further investigation.

Re-optimizing the query at runtime Another approach to improve sub-optimal plans is to *re-optimize* the query based on feedback obtained from partial execution of the query. A desirable goal in this approach is to not lose the work already done, which impacts the decision of when to re-optimize. The work by [105] only considers re-optimization in blocking operators, e.g., after the build of a Hash Join or after a Sort, where the results are already materialized. Therefore, no additional overheads of result materialization are incurred if a re-optimization were to be triggered. They also adopt a conservative strategy and only re-optimize the "remainder" of the query, i.e., the query optimizer must use the partial results obtained from execution so that the work done so far will not be wasted. Furthermore, since re-optimization can add significant overheads, they put in guard conditions for triggering

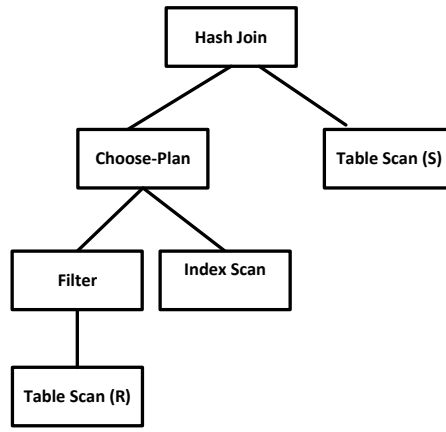


Figure 6.2: The Choose-Plan operator picks between Table Scan and Filter vs. Index Scan. At runtime, before query execution begins, one of the two children of Choose-Plan is picked, based on which one has a lower estimated cost.

re-optimization, including: (a) the difference between estimated and actual cardinality of the operator's inputs to exceed a pre-determined threshold, and (b) the estimated time taken to re-optimize the query is significantly smaller than the estimated time to run the remainder of the query using the current plan.

The adaptive query execution (AQE) feature in Databricks [5] collects statistics from completed pipelines to potentially change decisions such as which physical join operator to use, e.g., shuffle vs. broadcast join, or the degree of parallelism to use. They also consider the option of *canceling* a currently executing pipeline, and restarting it with a modified plan. For example, consider the case when there are two Table Scan operators that are the children of a join operator, and one Table Scan operator completes but the other is early in its execution, e.g., only 5% complete. In this case, AQE will cancel the ongoing Table Scan of the second table, and restart it with a newly added Bloom filter (see Section 4.6.2) that was computed during the processing of the first Table Scan, thereby potentially reducing the number of rows that need to be joined significantly.

The *progressive optimization* work [130], which was explored in IBM DB2, identifies a *validity range* for one or more operators in the plan

during query optimization. A validity range is a conservative range of cardinality values outside of which the operator is known to be sub-optimal in terms of the optimizer's estimated cost. During query optimization, each time an operator o is found to have lower cost than an alternative, the optimizer attempts to update the upper and lower values of the validity range of o. Specifically, they calculate cardinality values of the input(s) that could make the alternative cheaper than o, and use those cardinality values to update the bounds. Once validity ranges are computed, they introduce one or more CHECK operators into the plan. During query execution, the CHECK operator buffers rows of its child and counts the number of rows output by the child thus far. If the actual cardinality falls outside of the validity range of a CHECK operator, re-optimization is triggered. During re-optimization, the partial results from executing the original plan, which are buffered in the CHECK operator, are persisted into a temporary table and exposed to the query optimizer as a materialized view. During query re-optimization, the optimizer has additional cardinality information about operators that were executed. When the optimizer generates a new plan, it has the option to reuse the partial results of prior execution(s), but is not required to do so. For instance, if the materialized results are large, the optimizer may now find a lower cost plan, e.g., with a different join order, that does not use the partial results.

Fine-grained adaptive query processing This category of techniques executes different plans over fine-grained partitions of the data, and adapts the plan depending on feedback from execution. Due to the significant changes needed to the query execution engine to incorporate these techniques, we have not yet seen their adoption by database systems so far. Below, we discuss two example approaches from the literature. For a survey of adaptive query processing techniques we refer readers to [54].

Eddies [12] enables fine-grained run-time control over the plans executed for a query, using the *eddy* operator. They treat query execution as a process of routing rows through operators. The eddy operator consumes rows from the input relations, and routes rows to physical operators. The rows output from the physical operator are routed back

to the eddy. Changing the order in which a tuple is routed through the operators in effect changes the query plan used for that tuple. For example, for a join query, different join orders can be executed by changing the sequence in which a tuple is routed through the join operators. The eddy operator monitors the execution and makes the routing decisions for the tuples based on feedback about selectivity and execution cost obtained from execution so far. One requirement of the Eddies approach is that the physical operators need to be non-blocking and therefore they use operators such as symmetric hash join.

SkinnerDB [193] focuses on the problem of join ordering of plans. They divide query execution into fine-grained time slices, e.g., on the order of milliseconds or smaller. In each time slice, they execute a different join order on a batch of rows from the tables referenced in the query. The results from different executions are merged until the full result of the query is obtained. The execution progress of each join order is measured, which is an indicator of the efficiency of that join order. At the start of each time slice, the decision of which join order to pick is based on reinforcement learning [110], which balances between the goals of exploiting join orders that have worked well so far and exploring new join orders.

6.3 Influencing Plan Choice Using Hints

The DBA or the application developer may want to influence the query optimizer to select a different plan based on their knowledge of the application or the database. Such intervention is enabled in database systems by the mechanism of *hints*, which are directives available in the query language itself to enable the user to override the default behavior of the optimizer.

Although hints provide an important lever to influence plan quality, their usage is challenging. The impact of specifying a hint on query performance may only be determined by executing the query, making it a process of trial and error to select good hints. Furthermore, since the hints can become outdated as the data changes, new physical design structures are added, or the application characteristics evolve, query

hints also need to be managed over the lifetime of the query, e.g., with mechanisms such as APRC (see Section 6.2).

We now describe examples of query hints in Microsoft SQL Server (Section 6.3.1) and how hints can be implemented in a query optimizer based on the Volcano/Cascades framework (Section 6.3.2).

6.3.1 Hints in Microsoft SQL Server

Microsoft SQL Server provides a wide range of hints [140]. Below we sample a few classes of hints and provide illustrative examples.

Physical operator hints A common class of hints is to specify a physical operator for implementation of logical operators, such as Join, group-by, and Union, in the query. For example, a HASH JOIN hint specifies that only the Hash Join operator must be considered for any logical join operator in the query, unless Nested Loops Join is the only possible alternative, e.g., for non equi-join predicates. The user typically relies on their domain knowledge of the database, e.g., primary key - foreign key relationships that may exist among tables, data size and distribution information, to decide what hints to use. For instance, an experienced DBA may know that the original plan chosen by the optimizer is slow because of too many rows being sought by an Index Seek operator on the inner side of a Nested Loops Join operator, and hence may wish to use Hash Join instead. Hints can be specified by modifying the SQL query using the OPTION keyword as shown in example Query 33 below:

Query 33

```
SELECT c_name, SUM(o_price * o_qty)
FROM customer INNER JOIN orders ON c_id = o_cid
WHERE o_salesdate BETWEEN '2024-01-01' AND '2024-01-31'
GROUP BY c_name
OPTION (HASH JOIN)
```

Join order hints The FORCE ORDER hint tells the optimizer to use the join order specified in the query text. For example, Query 34 below will result in a plan which joins the *customer* and *orders* tables first and joins the resulting relation with the *lineitem* table:

Query 34
```
SELECT c_name, SUM(o_price * o_qty)
FROM customer INNER JOIN orders ON c_id = o_cid
INNER JOIN lineitem ON o_oid = l_oid
WHERE o_salesdate BETWEEN '2024-03-01' AND '2024-06-30'
AND l_shipdate BETWEEN '2024-03-01' AND '2024-06-30'
AND l_shipdate + 10 < l_receiptdate
GROUP BY c_name
OPTION (FORCE ORDER, HASH JOIN)
```

If the estimates of the cardinality of rows qualifying the selection predicates on the *lineitem* are significantly underestimated, the optimizer can pick a plan which joins the *orders* and *lineitem* table first, resulting in a sub-optimal plan that executes too long. In this case, the user can override the optimizer's choice and force a better join order as well as join method (HASH JOIN).

Runtime environment hints These hints allow the user to control resources consumed by the plan during execution. Commonly used examples include maximum degree of parallelism to use (MAXDOP <value>) and maximum memory grant (MAX_GRANT_PERCENT=<value>). Such hints can be valuable in practice since limiting the resource consumption of a plan helps control its performance impact on other concurrently executing queries.

Cardinality hints As noted in Section 5.4, the optimizer makes assumptions when estimating cardinality of expressions in the search. However, the specific assumption that works best for a given query depends on the query and the data distribution. Microsoft SQL Server supports hints that allow users to specify which assumption to use in cardinality estimation. For example, consider a query with two predicates, i.e., *o_orderdate>'2024-01-01' AND o_shipdate>'2024-01-31'*. Recall that by default the optimizer uses the independence assumption to estimate the cardinality for conjunctions of selection predicates. If the DBA knows that *o_orderdate* and *o_shipdate* columns are highly correlated, they can use the hint ASSUME_MIN_SELECTIVITY_FOR_FILTER_ESTIMATES, which tells the optimizer to estimate the selectivity of a conjunction of two

or more predicates as the selectivity of the most selective predicate. In effect, this hint forces the optimizer to assume full correlation between individual predicates. Note that cardinality hints apply to all predicates in the query.

USE PLAN hint The USE PLAN hint allows the user to specify a complete execution plan to be used for the query. The plan provided in the hint must be fully specified (in a pre-specified XML format supported by Microsoft SQL Server) including the join order, physical operators, and access methods to use. This hint provides the maximum degree of control since the user can specify all aspects of the physical plan. When a USE PLAN hint is provided, the query optimizer first validates the specified plan to ensure it is semantically equivalent to the query and that the access paths used are still valid (see Section 6.3.2). The DBMS executes the plan only if it passes the above validation.

6.3.2 Implementing hints

The implementation of hints vary depending on the type of hints. Some hints, such as runtime environment hints (e.g., degree of parallelism to use) or cardinality hints can be implemented by setting the corresponding configuration option in the optimizer. Physical operator hints and join order hints can be supported by disabling specific transformation rules (see Section 4). For example, the HASH JOIN hint is supported by disabling implementation rules that transform a Join to Nested Loops Join and Join to Merge Join. Similarly, a FORCE ORDER hint can be achieved by disabling rules that can change the join order, e.g., join commutativity and join associativity rules, thereby ensuring that the initial join order provided in the original query text is preserved in the final plan.

The USE PLAN hint needs to ensure that the plan specified in the hint (say P) is valid for the query. In Volcano/Cascades framework, it can be implemented by leveraging the optimizer's search algorithm to find a plan that satisfies the hint. Unfortunately, the original cost-based search algorithm, as described in Section 2, is no longer suitable for checking validity of a given plan, since that may incorrectly prune out

the required plan. Therefore, to check the validity, the optimizer disables cost-based pruning. It checks if any plan it has generated so far matches P. If so, the search terminates since it can be sure P is valid, as P was reached via a set of transformations from the input query. Although cost-based pruning is no longer available, the optimizer can leverage properties of P to eliminate certain transformations from consideration, thereby speeding up validation. For example, if the table R is accessed by an Index Scan in the plan specified by the hint, when validating the plan the optimizer will only consider the implementation rule that transforms the logical expression of Get(R) to Index Scan.

6.4 Optimizing Parameterized Queries

Parameterized queries are used extensively by applications. Parameterized queries are specified via a stored procedure or a prepare statement, and executed multiple times, with different parameter binding values. Consider the following Query 35 that defines a stored procedure named *custinfo* that returns customers below a given age and from a given state:

Query 35

```
CREATE PROCEDURE custinfo int @p1, nvarchar(32) @p2
AS
SELECT name, age, state
FROM customer
WHERE age < @p1 AND state = @p2
```

To execute the stored procedure, the application calls the EXECUTE statement with a binding for each parameter as the input. The code below shows two different invocations (also referred to as *instances*) of *custinfo* with different parameter binding combinations:

```
EXECUTE custinfo @p1 = 20, @p2 = 'Vermont';
EXECUTE custinfo @p1 = 50, @p2 = 'California';
```

6.4.1 Challenges in parametric query optimization

The straightforward approach of generating a plan for each instance of the parameterized query would achieve the best plan quality since the

optimizer can choose the plan that is best suited for the selectivity of the parameterized predicate(s) in the instance. However, this approach has two drawbacks. First, generating a plan for a query is computationally intensive as described in Section 6.1, and doing so for each instance of a parameterized query instance will be expensive and add latency to query execution. Second, it can be wasteful to generate a plan for each query instance because the optimizer's choice of plans for different parameter bindings may not change. In practice, the number of distinct plans generated by the optimizer is typically much fewer compared to the number of distinct parameter binding combinations.

Therefore, parametric query optimization (PQO) must strike a balance between optimizing query instances to generate new plans and providing good performance across different instances of the parameterized query using the generated plans. This leads to the following framework for PQO: (a) Identify a small set of plans to use for a parameterized query that could be cached and reused across multiple instances of the parameterized query. (b) For each incoming query instance, determine which plan among the set of cached plans is best suited for executing that instance.

6.4.2 Identifying a set of plans to cache

One line of work approaches the challenge of identifying a set of plans to cache by assuming that every point in the selectivity space is equally likely to be queried [49, 73, 98]. These techniques rely on assumptions of the optimizer's cost model, such as the cost is linear [73] or piece-wise linear [98] in the selectivities, or the cost is monotonic with selectivities [49]. One example of such an approach is the work on *anorexic* plan diagrams [49]. It aims to find the smallest subset of plans from the plan diagram [173] such that the sub-optimality between the optimal plan from the subset and the original optimal plan for every point in the selectivity space is bounded by a given factor. While the above guarantee on sub-optimality is desirable, this approach can incur prohibitive cost in identifying such plans. This is because generating a plan diagram requires creating a grid of points in the selectivity space and making one optimizer call for each grid point, which grows exponentially with the number of parameterized predicates in the query.

Another line of work only considers a portion of the selectivity space in the neighborhood of query instances that have executed historically. Based on the costs of plans generated for these queries, and the plan cost monotonicity assumption, they construct regions in the selectivity space where different plans are optimal. The shapes of such the regions constructed vary, e.g., one technique assumes the shape of the region to be a rectangle in the 2-d selectivity space [115] (or a hyper-rectangle in general), whereas another technique assumes the shape of the region to be an ellipse [18].

Microsoft SQL Server approaches the problem by partitioning the entire selectivity space into a grid, where each dimension corresponds to the selectivity of a parameterized predicate [138]. The creation of such a grid is done when the first query instance for that parameterized query is executed, and the boundary values are chosen based on the data distribution of the column corresponding to the parameterized predicate. When a query instance falling into a region (i.e., grid cell) is executed, if no plan already exists for that region, the DBMS generates a plan for that instance and caches it. Subsequent instances that fall into the same region reuse the cached plan. Oracle's approach also uses rectangular regions, however, they generate and expand these regions dynamically [115]. Their dynamic adjustment of plans is based on an assumption of the cost function. Specifically, if the same plan is generated for two points in the selectivity space, then they assume that the plan is optimal in all points in the rectangle in 2-d (or hyper-rectangle in higher dimensions) with those two points being on opposite corners. They use this assumption to generate such hyper-rectangular regions and associate one plan with it. Subsequently, any query instance falling within this hyper-rectangle reuses the same plan.

More recently, [194] proposed a *workload-driven* technique that leverages query logs, i.e., a set of instances of the parameterized query that has executed in the past, for deciding which set of plans to cache. In Microsoft SQL Server, such a query log can be obtained via the Query Store [141]. Specifically, from the set of plans that are picked by the query optimizer for query instances in the workload, they choose a subset of plans up to a pre-specified limit, which if cached would reduce the cost of the entire workload the most. This approach uses the ability

of the query optimizer to efficiently re-cost a plan P for a different selectivity [61]. An important characteristic of this approach is that it does not need to rely on simplifying assumptions of the optimizer's cost model.

6.4.3 Selecting the plan to execute for a query instance

Given a set of cached plans for a parameterized query, when a new query instance q is executed, the DBMS needs to select one of the cached plan to execute q. The straightforward approach of re-costing each cached plan for q and choosing the plan with the lowest cost is simply too expensive. This is because plan selection is on the critical path of query execution and therefore its latency must be low.

In Microsoft SQL Server [138] and Oracle [115], the cached plan corresponding to the rectangular region that the query instance q falls into is used to execute q. In [18], a distance function over the selectivities is defined between the query point q and each cached plan. The closest such plan or one whose distance is within a predefined threshold is selected.

Recently, the technique proposed in [194] uses a different approach where a classification model learned based on query logs (e.g., available for Microsoft SQL Server via the Query Store [141]) is used to find the cached plan most appropriate for an incoming query instance. This model uses features of a query that are inexpensive to compute, e.g., predicate selectivities. They use a decision tree based machine learning model [41] that has low overhead of inference with a tail latency less than 300 microseconds. The use of decision trees leads to rectangular partitioning of the selectivity space, but the partitioning is data-driven based on query logs rather than based on assumptions of the properties of the cost model.

6.5 Suggested Reading

Citation numbers below correspond to numbers in the References section.

[7] G. Antoshenkov, "Dynamic Query Optimization in Rdb/VMS," in *Proceedings of IEEE 9th International Conference on Data Engineering*, IEEE, pp. 538–547, 1993

[45] R. L. Cole *et al.*, "Optimization of Dynamic Query Evaluation Plans," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '94, pp. 150–160, Minneapolis, Minnesota, USA: Association for Computing Machinery, 1994. DOI: 10.1145/191839.191872

[130] V. Markl *et al.*, "Robust Query Processing through Progressive Optimization," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 659–670, 2004

[135] Microsoft, *Adaptive Joins in Microsoft SQL Server*, 2017. URL: https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing-details?view=sql-server-ver16

[161] Oracle, *The Optimizer In Oracle Database 19c*, 2024. URL: https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-optimizer-with-oracledb-19c-5324206.pdf

7

Open Problems

In this monograph, we have discussed many research ideas that have already influenced query optimizers of today's database systems. Despite decades of work, interest in query optimization research remains very strong. Given the breadth of query optimization topics and the large number of research papers, we are unable to summarize all relevant research. Instead, in this section we focus on a few selected topics and give a glimpse of some of the approaches that have been proposed.

7.1 Robust Query Processing

Query optimization suffers from inaccuracies in cardinality and cost estimations. Moreover, incrementally more accurate cardinality and cost estimations may not necessarily improve the quality of the plans. *Robust query processing* instead focuses on developing query optimization and query execution techniques to avoid poor query performance even in the presence of errors in cardinality and cost estimation. Leveraging runtime techniques to mitigate the inefficiency of query plans that we discussed in the previous section is one way to ensure robust behavior. In this section, we will mention a few other approaches that have been pursued. We recommend [88, 89] for additional details on this topic.

The *least expected cost* (LEC) optimization [44] takes the distribution of selectivities as input to capture cardinality errors and aims to choose a plan that minimizes the *expected cost* of the query. In contrast, the work in [13] proposes that the user, such as the application programmer or the query writer, should make explicit the extent to which they want to guard against uncertainty in selectivity estimations. For example, given a distribution of estimated costs for the plans as selectivities change, the user could specify that they prefer the plan that has a lowest median cost, or a more conservative user could specify the 90th percentile cost instead. In general, the user can specify what percentile value of the query cost distribution to look at in comparing costs of plan alternatives. The challenge in these approaches is to generate a distribution of selectivities and the corresponding cost in an efficient way that applies for the entire surface of SQL.

The Lookahead Information Passing (LIP) [205] technique focuses on reducing the variability in the performance of plans with sub-optimal join orders, induced by errors in cardinality estimations. Their approach is applicable for in-memory star schema databases and for the space of left-deep query plan trees, consisting of scan of the fact table followed by Hash Joins with dimension tables. As in many other database systems, they generate Bloom filters (see Section 4.6.2) for each dimension table containing selection predicates and push these Bloom filters to the scan of the fact table, thereby reducing the impact of sub-optimal join ordering. Their technique adaptively reorders the evaluation of the Bloom filters based on the execution feedback at runtime. One of the novel aspects of the LIP paper is their formalization of the notion of robustness as the difference in cost between the best and worst join orders normalized by the size of the fact table. Another example of robust query optimization is in the context of parametric query optimization, discussed in Section 6.4, that picks a single plan that minimizes the variance of costs over possible values of the parameters [33].

Research ideas have been developed to make key physical operators resilient to errors in cardinality and cost estimations. SmoothScan [20] adaptively switches between sequential Table Scan and Index Scan. Likewise, G-Join [81] unifies and thus generalizes the design of Nested Loops Join, Sort-Merge Join, and Hash Join.

A different approach to robustness is through algorithms for answering relational queries that are able to avoid generation of large intermediate relations that the execution plans generated by System-R or Volcano/Cascades style optimizers may be susceptible to. For acyclic project-join queries, Yannakakis's algorithm [203] ensures this goal by careful use of semi-join reduction and the algorithm has a running time polynomial in the size of input relations and the output size. However, for project-join queries that are not necessarily acyclic in structure, Yannakakis's algorithm does not apply. For a general project-join query, a tight worst-case bound on the size of the output (AGM bound) is proposed by [11]. The worst-case optimal join (WCOJ) [154] proposes an algorithm which has a running time polynomial in the size of the AGM bound. Like Yannakakis's algorithm, the above algorithm avoids generation of large intermediate relations. Since the publication of that paper, many algorithmic variants have been published and there has been work in adapting the algorithm to reduce the dependence of these algorithms on building many precomputed structures. An example of the latter line of work is [66]. We have only seen limited adoption of these algorithms as despite attractive theoretical properties, in many situations they perform worse than traditional execution plans.

7.2 Query Result Caching

Reusing results of previous queries to improve performance of subsequent queries has been extensively studied. Such repeated queries frequently arise, e.g., when a report is published, it may be pulled by many applications, including business intelligent dashboards. Recent papers (AWS Redshift [174, 177]) reported significant repetitiveness at the level of table scans, as well as at the overall query level, including parametric queries with exactly the same parameter bindings.

We now discuss a couple of examples of how result caching can be used. In the Oracle database system, the results of executing the query are stored in a result set cache [158]. If the exact same query repeats, and the underlying data in the tables referenced by the query have not been modified, then the query is answered using the cached results. Another technique (used in AWS Redshift [177]) is *predicate caching*.

When a query plan executes a Scan operator on a table containing a predicate (e.g., the predicate $l\_discount = 0.1$), the DBMS caches the predicate as well as a compact representation of the row ranges that qualify the predicate. During scan, if the predicate in the scan matches one of the predicates in the cache, the Scan operator uses the row ranges to retrieve the qualifying rows. In their system, all modified rows are inserted into a new buffer and given higher row numbers and hence the cached row ranges do not need to be invalidated when data is inserted, deleted, or updated. Finally, we note that there is a large body of work on semantic caching. Some references on this topic include [32, 51, 87]. However, some of the techniques proposed in the literature incur significant overhead in leveraging cached data.

7.3 Feedback-driven Statistics

There have been several research papers that aim to exploit execution feedback by incorporating it into the statistics used by the query optimizers with the goal of improving the accuracy of statistics as well as to reduce the cost of building and maintaining statistics. Earliest work in this direction [40] proposes modeling the cardinality estimates by curve-fitting functions with actual cardinality from query execution. In contrast, [2] proposes building a histogram with minimal upfront costs by initializing it from a uniform distribution. Then, they continuously refine the histogram with query execution feedback. The work by [22] develops a multi-dimensional histogram structure that is amenable to incorporating cardinality feedback to model non-uniform data distributions while leveraging efficient data structures. LEO [186] corrects the errors in cardinality estimates with adjustments derived from execution feedback.

It is important to note that the availability of actual cardinality from execution feedback is limited to those expressions that *occur in the plan*. However, cardinality of expressions that are *not* part of the plan may be crucial as well in finding a higher quality plan. Consider a simple example of a Table Scan operator containing the conjunction of two predicates ($l\_tax = 0.04$ *AND* $l\_discount = 0.08$). The cardinality obtained from executing this operator is the cardinality of the conjunction of

predicates. However, if actual cardinality could additionally be obtained for the individual predicates *l_tax = 0.04*, *l_discount = 0.08*, it could potentially lead to a better plan containing an Index Scan using an index on column *l_tax* (or *l_discount*) or an index intersection involving both indexes. Techniques have been proposed for collecting cardinality of expressions that are not part of the plan with low overhead by introducing small changes to the query execution plan [37].

7.4 Leveraging Machine Learning for Query Optimization

The topic of leveraging machine learning (ML) for query optimization has experienced an explosive growth of interest with many research papers in recent conferences.

Since cardinality and cost estimation are prediction problems, they are natural candidates for application of ML techniques. Models are built to learn the joint data distribution across columns and tables in the database, which are then used to estimate the cardinality of any query expressions [95, 201, 202, 207]. A comparative and empirical study of some of the ML-based CE techniques can be found in [108, 196]. Techniques such as [121, 151] aim to make ML models for cardinality estimation resilient to errors when the workload or data changes. Application of ML to improve cost estimation has been studied as well [129, 191, 204]. The work by [182] compares the use of different ML approaches to cost estimation, and considers how a learned cost model can be integrated into a Cascades based query optimizer. There has been recent work on pre-trained cost models that may be adapted with little or no training for unseen databases [94, 124]. The traditional query optimization framework discussed in this monograph can directly make use of cardinality and cost estimators improved through ML techniques.

We now discuss ML inspired techniques that influence the search component of the optimizer. Bao [127] is a learning component that is used as a "value-add" layer on top of the traditional query optimizer. It leverages query hints (Section 6.3) to guide the optimizer to generate alternatives to the default plan. The set of hints considered consists of only plan-level Boolean hints (e.g., Hash Join hint). Bao uses reinforcement learning and learns a model to choose the best hint from the set

of hints for a given query. Bao generates a modified query decorated with its selection of hints, executes the plan generated by the optimizer for that modified query, and observes the execution time (reward). Over time, Bao improves the plan quality with its learning and outperforms the traditional query optimizer it uses. Bao has the benefit of easy integration with the traditional optimization framework but its ability to steer the optimizer is limited by the set of hints. Furthermore, hints are global for the query and thus lacks fine-grained influence on altering the query plans. Like Bao, Lero [206] leverages a traditional query optimizer and uses an approach based on pairwise learning-to-rank paradigm. Lero has two components. The first component (Plan Explorer) uses the cardinality injection API to alter cardinality of subexpressions for the query, which leads the optimizer to generate plans potentially distinct from its default choice. The second component (Plan Comparator) is a binary classifier that identifies the better plan from a pair of plans. An advantage of this approach is that since the task of comparing two plans is simpler than predicting the execution cost of a plan, the model can be trained at lower cost and achieve higher accuracy.

In contrast to Bao and Lero, Neo [128] and Balsa [200] take responsibilities of generating the query plans for the scope of single block Select Project Join queries with aggregates. They use reinforcement learning to learn a cost function and use that to guide its search for a plan. Neo bootstraps its repertoire of plans and their costs by leveraging a traditional query optimizer. In contrast, Balsa bootstraps using a rudimentary cost model and does not use the search component of an existing optimizer. This is because Balsa targets scenarios where there may not be an existing query optimizer, e.g., for a new execution engine. The rudimentary cost model steers Balsa away from very expensive plans as candidates. In addition, Balsa uses timeout to ensure that the plans it picks for execution do not run for too long.

Despite much research, the efficacy of the proposed ML based optimization techniques are yet to be fully understood. While the proposed models are generally more accurate for cardinality estimation as well as cost estimation and generates better query plans, the techniques suffer from high overheads of training and inference, generation of surprisingly

poor query plans from time to time, high cost of updating models as data changes, and difficulties in interpretability and debuggability [196].

7.5 Other Research Topics in Query Optimization

There are many topics in query optimization that we did not cover in this monograph and in this section, we reference some of these areas. Multi-query optimization [175, 179] takes advantage of commonality across queries. In this scenario, the input is a set of queries and the output is a DAG, i.e., a set of plans that share physical operators. Thus, the goal is to generate such a DAG that can answer multiple queries by taking advantage of common subexpressions. In data-analytic queries, aggregate functions that are *window functions*, i.e., generating aggregation on a sliding window of ordered data, are important for dashboards, and ensuring that computation is reused across sliding windows is important. Streaming systems bring their unique challenges in query planning. For example, while streaming queries may share a number of common subexpressions, their data may change at different paces. Changing plans for such streaming queries has the added challenge that queries are stateful. Some of the techniques for optimizing streaming queries leverage operator reordering like selection push-down or pull-up, as discussed in Section 4. We refer interested readers to the survey article by [96] on this topic.

7.6 The Big Questions

As we end with this monograph, we want to leave the readers with our thoughts on some of the key opportunities to significantly improve the state-of-the-art in query optimization.

Most query engines today use variants of the query optimization framework described in this monograph. At the same time, these engines complement query optimization with runtime decisions to improve the performance of execution. However, we do not have a deeper understanding of what decisions are best deferred to runtime and what decisions should be made during the query optimization phase.

On a related note, leveraging execution feedback has been researched extensively. Some of the learning could be applied during the execution of the current query (e.g., adaptive operators) but perhaps some of the feedback is best aggregated offline to further fine tune the optimizer, potentially with the aid of advanced analytic models. So far, we have only early work on the latter but a breakthrough in this area could have broad impact.

Despite all the successes query optimizers have experienced, we still do not know how to appropriately tune the resources devoted by the query optimizer to match the impact of potential improvements on the execution costs of the queries. Ideally, the optimizer should invest substantial time only if the payoffs in plan quality will be large.

A broad area that has been largely ignored is the development of realistic benchmarks for query optimizers. We have benchmarks to stress test join orders but we need benchmarks with realistic data as well as complex queries that exercise the richness of transformation rules. Even though commercial database systems differ in their execution engines and two query optimizers on two different engines cannot be compared, a common set of queries and data sets will allow each engine to test regression and measure improvement. In case of Open Source database systems such as PostgreSQL, such benchmarks can accelerate improvement in its optimizer technology. On a related note, technology for testing and debugging the optimizers have seen little evolution over the years, and this should be a priority as well.

Two frameworks in query optimization have so far stood the test of time. One of them is the framework of System R with search, cardinality estimation, and cost estimation as three key components. The extensible query optimizer search framework pioneered by Volcano/Cascades has also enjoyed broad adoption. It may be worthwhile to revisit both these frameworks. Combinatorial search as well as ML, especially any foundation model like technology that can be universally applied may lead to new directions.

Finally, query optimizers traditionally have been stand-alone systems that take into account characteristics of data through statistics as the only additional information beyond the submitted query. We should ask ourselves what could be gained by offering applications a direct

way to interact with and influence the optimizer [29]. Specifically, the optimizer could offer APIs to the application to *inject cardinalities* for selected expressions as well as *specification of correlation between filter expressions*, thereby overriding default selectivity estimation. On the search side, *plan directives* provide more fine-grained and powerful constraints (e.g., [23]) on the structure and operators in the final plan than what is possible with query hints in today's optimizers (Section 6.3). For example, one may wish to constrain that the final plan must include a join between *customer* and *orders* table as the lowest-level join, but impose no constraints on the join order of the remaining relations in the query. The directives may also include *operator preference* to guard against erroneous cardinality estimation, e.g., prefer Table Scan unless the estimated cost of Index Seek is at least 50% lower. This may favor more robust plans at the cost of sacrificing performance. It is not necessary that the application builder explicitly provides such directives. It is far more likely that the application programmer writes code to leverage these interfaces that also takes into account execution feedback from historical information, e.g., Query Store in SQL Server [141], Automatic Workload Repository in Oracle DB [157].

Acknowledgements

We thank Anshuman Dutt, Wentao Wu, and Christian König, our colleagues at the Data Systems Group at Microsoft Research, who carefully read several parts of this monograph, and gave us detailed feedback. Our sincere thanks to the anonymous reviewers whose feedback greatly helped improve this monograph. Joe Hellerstein provided thoughtful suggestions and detailed comments on an early draft, and also shared with us feedback from students in his graduate class at UC Berkeley. We are grateful to Jignesh Patel for his comments on a draft of the monograph, and for sharing feedback from students in his graduate class at Carnegie Mellon University. We are also highly appreciative of Nicolas Bruno, Cesar Galindo Legaria, and Vassilis Papadimos from the Azure Data team at Microsoft for sharing with us their knowledge and insights on the Volcano/Cascades frameworks, as well as the Microsoft SQL Server and Fabric Data Warehouse query optimizers. Finally, we sincerely thank our spouses for their patience and support.

Appendix

A

Access Methods

We provide a brief overview of how the query execution engine in relational databases can access data stored in the base tables. The data in base tables can be physically organized using different persistent (i.e., on-disk) data structures. Some of the most commonly used structures are heaps, B-trees indexes [14], and columnstore indexes [189]. We use the examples of heaps and B+-tree indexes to introduce the important physical operators.[1] We note that there are other aspects of access methods on base tables that are not discussed below, e.g., partitioning, but are also relevant for query optimization.

Heap and B+-tree index A heap is an unordered collection of all records in the table. Each record (row) has a DBMS generated row id, and stores values for each column in the table. Rows are organized into pages and stored on disk. B+-trees are n-ary tree based data structures that organize the data ordered by the *key columns* of the index. Further, a B+-tree index can either be a clustered index or non-clustered index. In a clustered index, the leaf pages of the index contain the entire record

[1] In contrast to a B-tree, in a B+-tree, leaf pages contain a pointer to the next leaf page in index order, thereby enabling more efficient scans of a range of values.

(i.e., values of all columns of the table), whereas in a non-clustered index, the leaf pages only contain the key columns and the record id. Besides key columns, a non-clustered index may optionally contain additional *include* columns. In this case, each row in the leaf page of the index contains the key columns as well as the include columns. Observe that the B+-tree supports search (i.e., lookups or range scans) only over the key columns, and not the include columns. In contrast to heaps, B+-Tree indexes can greatly speed up the retrieval of the data, especially when only a selective subset of the data is needed to answer the query.

We use the same example table and query from Section 4.1 to describe the physical operators for accessing data in heaps and B+-tree indexes. Consider a table $S(id, a, b, c)$ with four columns, where id is the *primary key* of table S. Consider the following query Q_1:

```
SELECT S.a, S.b
FROM S
WHERE S.a > 10 AND S.b = 20
```

Table Scan Since the rows of a heap are unordered, a heap provides no ability to lookup any individual record. Thus, the only physical operator allowed on a heap is the Table Scan operator. Table Scan takes a table as an argument and returns all rows from the table. In our example, Table Scan of S returns all rows in S. Observe that when the table is large, the Table Scan operator can be expensive since it needs to fetch all pages of S from storage, including rows and columns that are not needed to answer the query.

Index Seek and Key Lookup When the query contains an equality predicate on any prefix of the key columns in the index, the Index Seek operator can be used to retrieve all rows satisfying the predicate. For example, consider a B+-tree index I_b built on table S with b as the key column. Then for Q_1, instead of scanning the full table, invoking Index Seek (I_b, $S.b = 20$) will find and retrieve row ids of all rows in the table satisfying the predicate. A special case of Index Seek is a Key Lookup operator which is used when the index is defined on a primary key or unique column of the table. In a Key Lookup, either 0 or 1 record is returned, whereas in an Index Seek, 0 or more rows

can be returned. Note that an invocation of the Index Seek operator performs a random I/O to access the data page containing the matching records. While an Index Seek is often used for identifying rows satisfying a selection predicates (e.g., $S.b = 10$), it can also be combined with a Nested Loops Join operator to efficiently support a join between two relations. Specifically, for each row from the outer relation, a Nested Loops Join operator can use an Index Seek on the inner side relation of the join if the key column of the index is the join column.

Index Scan When the query contains a range predicate, a B+-tree index enables efficient range scans using the Index Scan operator. Consider a B+-tree index I_a built on table S with a as the key column. Since the predicate $S.a > 10$ needs to retrieve a range of values, invoking Index Scan (I_a, $S.a > 10$) will retrieve row ids of all rows satisfying the range predicate on column a. We observe that for the Index Scan operators the predicate is an optional argument. If no predicate is specified, Index Scan returns all rows from the leaf pages of the index. Unlike Table Scan where the rows returned are unordered, these rows from Index Scan will appear in the order of the key columns of the index. Thus, the usefulness of Index Scan goes beyond its ability to retrieve rows since it can benefit other operators such as Merge Join and Stream Aggregate, which require their inputs to be sorted. B+-tree indexes containing include columns can be very effective in answering a query when all columns required to answer the query are available in the index, whether as part of key or include columns. For example, consider an index $I_a(b)$ where the key column is a, and the include column is b. Observe that the query Q_1 can be answered using an Index Scan $I_a(b)$ with $S.a > 10$ followed by a Filter operator that can apply the predicate $S.b = 20$. Since the column b is available in the index, we can avoid a Key Lookup into the clustered index to obtain column b.

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. Row-stores: How Different are They Really?" In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, pp. 967–980, Vancouver, Canada: Association for Computing Machinery, 2008. DOI: 10.1145/1376616.1376712.

[2] A. Aboulnaga and S. Chaudhuri, "Self-Tuning Histograms: Building Histograms without Looking at Data," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '99, pp. 181–192, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1999. DOI: 10.1145/304182.304198.

[3] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, "Mergeable Summaries," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 4, 2013, pp. 1–28.

[4] J. Aguilar-Saborit, R. Ramakrishnan, K. Srinivasan, K. Bocksrocker, I. Alagiannis, M. Sankara, M. Shafiei, J. Blakeley, G. Dasarathy, S. Dash, *et al.*, "POLARIS: the Distributed SQL Engine in Azure Synapse," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020, pp. 3204–3216.

[5] M. X. et al., "Adaptive and Robust Query Execution for Lakehouses at Scale," *Proceedings of the VLDB Endowment*, vol. 17, 2024.

[6] Amazon, *AWS: COUNT function*, 2024. URL: https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html.

[7] G. Antoshenkov, "Dynamic Query Optimization in Rdb/VMS," in *Proceedings of IEEE 9th International Conference on Data Engineering*, IEEE, pp. 538–547, 1993.

[8] P. M. Aoki, "Implementation of Extended Indexes in POSTGRES," in *ACM SIGIR Forum*, ACM New York, NY, USA, vol. 25, pp. 2–9, 1991.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pp. 1383–1394, 2015.

[10] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, *et al.*, "Amazon Redshift Re-invented," in *Proceedings of the 2022 International Conference on Management of Data*, pp. 2205–2217, 2022.

[11] A. Atserias, M. Grohe, and D. Marx, "Size Bounds and Query Plans for Relational Joins," in *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pp. 739–748, 2008.

[12] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 261–272, 2000.

[13] B. Babcock and S. Chaudhuri, "Towards a Robust Query Optimizer: A Principled and Practical Approach," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05, pp. 119–130, Baltimore, Maryland: Association for Computing Machinery, 2005. DOI: 10.1145/1066157.1066172.

[14] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pp. 107–141, 1970.

[15] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache Calcite: A Foundational Framework for Optimized Query Processing over Heterogeneous Data Sources," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 221–230, 2018.

[16] S. Bellamkonda, R. Ahmed, A. Witkowski, A. Amor, M. Zait, and C.-C. Lin, "Enhanced Subquery Optimizations in Oracle," *Proc. VLDB Endow.*, vol. 2, no. 2, Aug. 2009, pp. 1366–1377. DOI: 10.14778/1687553.1687563.

[17] P. A. Bernstein and D.-M. W. Chiu, "Using Semi-Joins to Solve Relational Queries," *J. ACM*, vol. 28, no. 1, Jan. 1981, pp. 25–40. DOI: 10.1145/322234.322238.

[18] P. Bizarro, N. Bruno, and D. J. DeWitt, "Progressive Parametric Query Optimization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 4, 2009, pp. 582–594. DOI: 10.1109/TKDE.2008.160.

[19] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution.," in *Cidr*, vol. 5, pp. 225–237, 2005.

[20] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser, "Smooth Scan: Statistics-oblivious Access Paths," in *2015 IEEE 31st International Conference on Data Engineering*, IEEE, pp. 315–326, 2015.

[21] N. Bruno and S. Chaudhuri, "Exploiting Statistics on Query Expressions for Optimization," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 263–274, 2002.

[22] N. Bruno, S. Chaudhuri, and L. Gravano, "STHoles: A Multidimensional Workload-Aware Histogram," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pp. 211–222, 2001.

[23] N. Bruno, S. Chaudhuri, and R. Ramamurthy, "Power Hints for Query Optimization," in *2009 IEEE 25th International Conference on Data Engineering*, pp. 469–480, 2009. DOI: 10.1109/ICDE.2009.68.

[24] N. Bruno, C. Galindo-Legaria, M. Joshi, E. Calvo Vargas, K. Mahapatra, S. Ravindran, G. Chen, E. Cervantes Juárez, and B. Sezgin, "Unified Query Optimization in the Fabric Data Warehouse," in *Companion of the 2024 International Conference on Management of Data*, pp. 18–30, 2024.

[25] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[26] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg, "The EXODUS Extensible DBMS project: An Overview," 1988.

[27] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya, "Towards Estimation Error Guarantees for Distinct Values," in *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 268–279, 2000.

[28] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems," in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 34–43, 1998.

[29] S. Chaudhuri, "Query Optimizers: Time to Rethink the Contract?" In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 961–968, 2009.

[30] S. Chaudhuri, G. Das, and U. Srivastava, "Effective use of Block-level Sampling in Statistics Estimation," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 287–298, 2004.

[31] S. Chaudhuri, U. Dayal, and V. Narasayya, "An Overview of Business Intelligence Technology," *Communications of the ACM*, vol. 54, no. 8, 2011, pp. 88–98.

[32] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing Queries with Materialized Views," in *Proceedings of the Eleventh International Conference on Data Engineering*, IEEE, pp. 190–200, 1995.

[33] S. Chaudhuri, H. Lee, and V. R. Narasayya, "Variance aware Optimization of Parameterized Queries," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 531–542, 2010.

[34] S. Chaudhuri, R. Motwani, and V. Narasayya, "Random Sampling for Histogram Construction: How Much is Enough?" *ACM SIGMOD Record*, vol. 27, no. 2, 1998, pp. 436–447.

[35] S. Chaudhuri, R. Motwani, and V. Narasayya, "On Random Sampling over Joins," *ACM SIGMOD Record*, vol. 28, no. 2, 1999, pp. 263–274.

[36] S. Chaudhuri and V. Narasayya, "Automating Statistics Management for Query Optimizers," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 1, 2001, pp. 7–20.

[37] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, "A Pay-as-You-Go Framework for Query Execution Feedback," *Proc. VLDB Endow.*, vol. 1, no. 1, Aug. 2008, pp. 1141–1152. DOI: 10.14778/1453856.1453977.

[38] S. Chaudhuri and K. Shim, "Including Group-By in Query Optimization," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94, pp. 354–366, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

[39] S. Chaudhuri and K. Shim, "Optimization of Queries with User-defined Predicates," *ACM Transactions on Database Systems (TODS)*, vol. 24, no. 2, 1999, pp. 177–228.

[40] C. M. Chen and N. Roussopoulos, "Adaptive Selectivity Estimation using Query Feedback," in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pp. 161–172, 1994.

[41] T. Chen and C. Guestrin, "Xgboost: A Scalable Tree Boosting System," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, 2016.

[42] A. Cheung, A. Solar-Lezama, and S. Madden, "Optimizing database-backed applications with query synthesis," *ACM SIGPLAN Notices*, vol. 48, no. 6, 2013, pp. 3–14.

[43] R. Chirkova, J. Yang, *et al.*, "Materialized Views," *Foundations and Trends® in Databases*, vol. 4, no. 4, 2011, pp. 295–405.

[44] F. Chu, J. Y. Halpern, and P. Seshadri, "Least Expected Cost Query Optimization: An Exercise in Utility," in *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 138–147, 1999.

[45] R. L. Cole and G. Graefe, "Optimization of Dynamic Query Evaluation Plans," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '94, pp. 150–160, Minneapolis, Minnesota, USA: Association for Computing Machinery, 1994. DOI: 10.1145/191839.191872.

[46] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, *et al.*, "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches," *Foundations and Trends® in Databases*, vol. 4, no. 1–3, 2011, pp. 1–294.

[47] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: the Count-Min Sketch and its Applications," *Journal of Algorithms*, vol. 55, no. 1, 2005, pp. 58–75.

[48] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: the Count-Min Sketch and its Applications," *Journal of Algorithms*, vol. 55, no. 1, 2005, pp. 58–75.

[49] H. D., P. N. Darera, and J. R. Haritsa, "On the Production of Anorexic Plan Diagrams," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07, pp. 1081–1092, Vienna, Austria: VLDB Endowment, 2007.

[50] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, *et al.*, "The Snowflake Elastic Data Warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 215–226, 2016.

[51] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan, *et al.*, "Semantic Data Caching and Replacement," in *VLDB*, vol. 96, pp. 330–341, 1996.

[52] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers," in *Proceedings of the 13th International Conference on Very Large Data Bases*, ser. VLDB '87, pp. 197–208, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987.

[53] K. Delaney, *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*. Microsoft Press, 2007.

[54] A. Deshpande, Z. Ives, and V. Raman, "Adaptive Query Processing," *Foundations and Trends® in Databases*, vol. 1, no. 1, 2007, pp. 1–140. DOI: 10.1561/1900000001.

[55] D. J. DeWitt and R. Ramamurthy, "Buffer Pool Aware Query Optimization," in *Proceedings of the 2005 CIDR Conference*, pp. 961–968, 2009.

[56] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "An Evaluation of Non-Equijoin Algorithms," in *Proceedings of the 17th International Conference on Very Large Data Bases*, ser. VLDB '91, pp. 443–452, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991.

[57] B. Ding, S. Chaudhuri, J. Gehrke, and V. Narasayya, "DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems," *Proc. VLDB Endow.*, vol. 14, no. 13, Sep. 2021, pp. 3376–3388. DOI: 10.14778/3484224.3484234.

[58] B. Ding, S. Chaudhuri, and V. Narasayya, "Bitvector-Aware Query Optimization for Decision Support Queries," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, pp. 2011–2026, Portland, OR, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3318464.3389769.

[59] B. Ding, V. Narasayya, and C. Surajit, *Errata and Updates to the Foundations and Trends in Databases article: Extensible Query Optimizers in Practice*, 2024. URL: https://www.microsoft.com/en-us/research/project/extensible-query-optimizers-in-practice-errata-and-updates/.

[60] W. Du, R. Krishnamurthy, and M.-C. Shan, "Query Optimization in a Heterogeneous DBMS," in *VLDB*, vol. 92, pp. 277–291, 1992.

[61] A. Dutt, V. Narasayya, and S. Chaudhuri, "Leveraging Re-Costing for Online Optimization of Parameterized Queries with Guarantees," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, pp. 1539–1554, Chicago, Illinois, USA: Association for Computing Machinery, 2017. DOI: 10.1145/3035918.3064040.

[62] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi, "Execution Strategies for SQL Subqueries," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07, pp. 993–1004, Beijing, China: Association for Computing Machinery, 2007. DOI: 10.1145/1247480.1247598.

[63] R. Fagin, "Normal Forms and Relational Database Operators," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79, pp. 153–160, Boston, Massachusetts: Association for Computing Machinery, 1979. DOI: 10.1145/582095.582120.

[64] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the Analysis of a Near-Optimal Cardinality Estimation Algorithm," in *Discrete Mathematics and Theoretical Computer Science*, Discrete Mathematics and Theoretical Computer Science, pp. 137–156, 2007.

[65] P. Flajolet and G. N. Martin, "Probabilistic Counting Algorithms for Data Base Applications," *Journal of computer and system sciences*, vol. 31, no. 2, 1985, pp. 182–209.

[66] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, "Adopting Worst-case Optimal Joins in Relational Database Systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020, pp. 1891–1904.

[67] M. Freitag and T. Neumann, "Every Row Counts: Combining Sketches and Sampling for Accurate Group-by Result Estimates," *ratio*, vol. 1, 2019, pp. 1–39.

[68] C. Galindo-Legaria and A. Rosenthal, "How to Extend a Conventional Optimizer to Handle One- and Two-sided Outerjoin," in *[1992] Eighth International Conference on Data Engineering*, pp. 402–409, 1992. DOI: 10.1109/ICDE.1992.213169.

[69] C. Galindo-Legaria and M. Joshi, "Orthogonal Optimization of Subqueries and Aggregation," SIGMOD '01, 2001, pp. 571–581. DOI: 10.1145/375663.375748.

[70] C. Galindo-Legaria and A. Rosenthal, "Outerjoin Simplification and Reordering for Query Optimization," *ACM Trans. Database Syst.*, vol. 22, no. 1, Mar. 1997, pp. 43–74. DOI: 10.1145/244810.244812.

[71] C. A. Galindo-Legaria, M. M. Joshi, F. Waas, and M.-C. Wu, "Statistics on Views," in *Proceedings 2003 VLDB Conference*, Elsevier, pp. 952–962, 2003.

[72] C. A. Galindo-Legaria, T. Grabs, S. Gukal, S. Herbert, A. Surna, S. Wang, W. Yu, P. Zabback, and S. Zhang, "Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server," in *2008 IEEE 24th International Conference on Data Engineering*, pp. 1190–1199, 2008. DOI: 10.1109/ICDE.2008.4497528.

[73] S. Ganguly, "Design and Analysis of Parametric Query Optimization Algorithms," in *VLDB*, vol. 98, pp. 228–238, 1998.

[74] P. B. Gibbons, Y. Matias, and V. Poosala, "Fast Incremental Maintenance of Approximate Histograms," in *VLDB*, vol. 97, pp. 466–475, 1997.

[75] P. B. Gibbons, Y. Matias, and V. Poosala, "Fast Incremental Maintenance of Approximate Histograms," *ACM Transactions on Database Systems (TODS)*, vol. 27, no. 3, 2002, pp. 261–298.

[76] J. Goldstein and P.-Å. Larson, "Optimizing Queries using Materialized Views: a Practical, Scalable Solution," *ACM SIGMOD Record*, vol. 30, no. 2, 2001, pp. 331–342.

[77] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys (CSUR)*, vol. 25, no. 2, 1993, pp. 73–169.

[78] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Comput. Surv.*, vol. 25, no. 2, Jun. 1993, pp. 73–169. DOI: 10.1145/152610.152611.

[79] G. Graefe, "Volcano - An Extensible and Parallel Query Evaluation System," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, 1994, pp. 120–135.

[80] G. Graefe, "The Cascades Framework for Query Optimization," *IEEE Data Eng. Bull.*, vol. 18, no. 3, 1995, pp. 19–29.

[81] G. Graefe, "New Algorithms for Join and Grouping Operations," *Computer Science-Research and Development*, vol. 27, 2012, pp. 3–27.

[82] G. Graefe and W. McKenna, "The Volcano Optimizer Generator," Colorado Univ at Boulder Dept of Computer Science, Tech. Rep., 1991.

[83] G. Graefe and W. J. McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search," in *Proceedings of IEEE 9th international conference on data engineering*, pp. 209–218, 1993.

[84] A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications*. MIT press, 1999.

[85] A. Gupta, I. S. Mumick, *et al.*, "Maintenance of Materialized Views: Problems, Techniques, and Applications," *IEEE Data Eng. Bull.*, vol. 18, no. 2, 1995, pp. 3–18.

[86] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible Query Processing in Starburst," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '89, pp. 377–388, Portland, Oregon, USA: Association for Computing Machinery, 1989. DOI: 10.1145/67544.66962.

[87] A. Y. Halevy, "Answering Queries using Views: A Survey," *The VLDB Journal*, vol. 10, 2001, pp. 270–294.

[88] J. Haritsa, "Robust query processing: A survey," *Foundations and Trends® in Databases*, vol. 15, no. 1, 2024, pp. 1–114. URL: https://nowpublishers.com/article/Details/DBS-089.

[89] J. R. Haritsa, "Robust Query Processing: Mission Possible," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, IEEE, pp. 2072–2075, 2019.

[90] H. Harmouch and F. Naumann, "Cardinality Estimation: An Experimental Survey," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, 2017, pp. 499–512.

[91] J. M. Hellerstein and M. Stonebraker, "Predicate Migration: Optimizing Queries with Expensive Predicates," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pp. 267–276, 1993.

[92] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in Practice: Algorithmic Engineering of a state of the art Cardinality Estimation Algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 683–692, 2013.

[93] E. Hewitt, *Cassandra: the Definitive Guide*. O'Reilly Media, Inc., 2010.

[94] B. Hilprecht and C. Binnig, "Zero-shot Cost Models for Out-of-the-box Learned Cost Prediction," *arXiv preprint arXiv:2201.00561*, 2022.

[95] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: Learn from Data, not from Queries!" *arXiv preprint arXiv:1909.00607*, 2019.

[96] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A Catalog of Stream Processing Optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, 2014, pp. 1–34.

[97] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major Technical Advancements in Apache Hive," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1235–1246, 2014.

[98] A. Hulgeri and S. Sudarshan, "Parametric Query Optimization for Linear and Piecewise Linear Cost Functions," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, Elsevier, pp. 167–178, 2002.

[99] IBM, *DB2 for z/OS: Histogram statistics*, 2024. URL: https://www.ibm.com/docs/en/db2-for-zos/12?topic=statistics-histogram.

[100] Y. E. Ioannidis and Y. Kang, "Randomized Algorithms for Optimizing Large Join Queries," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90, pp. 312–321, Atlantic City, New Jersey, USA: Association for Computing Machinery, 1990. DOI: 10.1145/93597.98740.

[101] Y. Ioannidis, "The History of Histograms (abridged)," in *Proceedings 2003 VLDB Conference*, Elsevier, pp. 19–30, 2003.

[102] Y. E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, pp. 268–277, 1991.

[103] Y. E. Ioannidis and V. Poosala, "Balancing Histogram Optimality and Practicality for Query Result Size Estimation," *Acm Sigmod Record*, vol. 24, no. 2, 1995, pp. 233–244.

[104] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin, "COMPASS: Online Sketch-based Query Optimization for In-memory Databases," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 804–816, 2021.

[105] N. Kabra and D. J. DeWitt, "Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pp. 106–117, 1998.

[106] S. Kandula, L. Orr, and S. Chaudhuri, "Pushing data-induced predicates through joins in big-data clusters," *Proceedings of the VLDB Endowment*, vol. 13, no. 3, 2019, pp. 252–265.

[107] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about Compiled and Vectorized Queries but were afraid to ask," *Proceedings of the VLDB Endowment*, vol. 11, no. 13, 2018, pp. 2209–2222.

[108] K. Kim, J. Jung, I. Seo, W.-S. Han, K. Choi, and J. Chong, "Learned Cardinality Estimation: An In-Depth Study," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, pp. 1214–1227, Philadelphia, PA, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3514221.3526154.

[109] W. Kim, "On Optimizing an SQL-like Nested Query," *ACM Trans. Database Syst.*, vol. 7, no. 3, Sep. 1982, pp. 443–469. DOI: 10.1145/319732.319745.

[110] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," in *European conference on machine learning*, Springer, pp. 282–293, 2006.

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

[112] P. Krishnan, J. S. Vitter, and B. Iyer, "Estimating Alphanumeric Selectivity in the Presence of Wildcards," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pp. 282–293, 1996.

[113] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou, "SQL Server Column Store Indexes," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11, pp. 1177–1184, Athens, Greece: Association for Computing Machinery, 2011. DOI: 10.1145/1989323.1989448.

[114] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, IEEE, pp. 75–86, 2004.

[115] A. W. Lee and M. Zait, "Closing the Query Processing Loop in Oracle 11g," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, 2008, pp. 1368–1378.

[116] K. Lee, A. Dutt, V. Narasayya, and S. Chaudhuri, "Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, 2023, pp. 2871–2883.

[117] M. K. Lee, J. C. Freytag, and G. M. Lohman, "Implementing an Interpreter for Functional Rules in a Query Optimizer.," in *VLDB*, pp. 218–229, 1988.

[118] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How Good Are Query Optimizers, Really?" *Proc. VLDB Endow.*, vol. 9, no. 3, Nov. 2015, pp. 204–215. DOI: 10.14778/2850583.2850594.

[119] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark," *The VLDB Journal*, vol. 27, 2018, pp. 643–668.

[120] W. Lemahieu, S. vanden Broucke, and B. Baesens, *Principles of Database Management: the Practical Guide to Storing, Managing and Analyzing Big and Small Data*. Cambridge University Press, 2018.

[121] B. Li, Y. Lu, and S. Kandula, "Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts," in *Proceedings of the 2022 International Conference on Management of Data*, pp. 1920–1933, 2022.

[122] G. Lohman, *Is Query Optimization a "Solved" Problem?* 2014. URL: https://wp.sigmod.org/?p=1075.

[123] G. M. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives," *ACM SIGMOD Record*, vol. 17, no. 3, 1988, pp. 18–27.

[124] Y. Lu, S. Kandula, A. C. König, and S. Chaudhuri, "Pre-training Summarization Models of Structured Datasets for Cardinality Estimation," *Proceedings of the VLDB Endowment*, vol. 15, no. 3, 2021, pp. 414–426.

[125] L. F. Mackert and G. M. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," in *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pp. 84–95, 1986.

[126] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for local queries," in *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pp. 84–95, 1986.

[127] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making Learned Query Optimization Practical," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21, pp. 1275–1288, Virtual Event, China: Association for Computing Machinery, 2021. DOI: 10.1145/3448016.3452838.

[128] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A Learned Query Optimizer," *arXiv preprint arXiv:1904.03711*, 2019.

[129] R. Marcus and O. Papaemmanouil, "Plan-structured Deep Neural Network Models for Query Performance Prediction," *arXiv preprint arXiv:1902.00132*, 2019.

[130] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust Query Processing through Progressive Optimization," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 659–670, 2004.

[131] Y. Matias, J. S. Vitter, and M. Wang, "Wavelet-based Histograms for Selectivity Estimation," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pp. 448–459, 1998.

[132] W. J. McKenna, *Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator*. University of Colorado at Boulder, 1993.

[133] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, H. Ahmadi, D. Delorey, S. Min, *et al.*, "Dremel: A Decade of Interactive SQL Analysis at Web Scale," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020, pp. 3461–3472.

[134] J. Melton and A. R. Simon, *SQL: 1999: Understanding Relational Language Components*. Elsevier, 2001.

[135] Microsoft, *Adaptive Joins in Microsoft SQL Server*, 2017. URL: https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing-details?view=sql-server-ver16.

[136] Microsoft, *Intro to Query Execution Bitmap Filters*, 2019. URL: https://techcommunity.microsoft.com/t5/sql-server-blog/intro-to-query-execution-bitmap-filters/ba-p/383175.

[137] Microsoft, *Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator*, 2021. URL: https://learn.microsoft.com/en-us/previous-versions/dn673537(v=msdn.10)?redirectedfrom=MSDN.

[138] Microsoft, *Parameter Sensitive Plan optimization in Microsoft SQL Server*, 2022. URL: https://learn.microsoft.com/en-us/sql/relational-databases/performance/parameter-sensitive-plan-optimization?view=sql-server-ver16.

[139] Microsoft, *Cardinality Estimation (SQL Server)*, 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver16.

[140] Microsoft, *Hints (Transact-SQL) - Query*, 2023. URL: https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver16.

[141] Microsoft, *Microsoft SQL Server Query Store*, 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store?view=sql-server-ver16.

[142] Microsoft, *Microsoft SQL Server Query Store*, 2024. URL: https://learn.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store?view=sql-server-ver16.

[143] Microsoft, *Microsoft SQL server: Approximate Count Distinct*, 2024. URL: https://learn.microsoft.com/en-us/sql/t-sql/functions/approx-count-distinct-transact-sql?view=sql-server-ver16.

[144] Microsoft, *Microsoft SQL Server: Memory grant feedback*, 2024. URL: https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing-memory-grant-feedback?view=sql-server-ver16.

[145] Microsoft, *Microsoft SQL Server: Query Processing Architecture Guide*, 2024. URL: https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver16.

[146] Microsoft, *Statistics: Microsoft SQL Server*, 2024. URL: https://learn.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-ver16.

[147] Microsoft, *Automatic Plan Correction in Microsoft SQL Server*. URL: https://learn.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning?view=sql-server-ver16.

[148] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan, "Magic is Relevant," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90, pp. 247–258, Atlantic City, New Jersey, USA: Association for Computing Machinery, 1990. DOI: 10.1145/93597.98734.

[149] R. O. Nambiar and M. Poess, "The Making of TPC-DS," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06, pp. 1049–1058, Seoul, Korea: VLDB Endowment, 2006.

[150] V. Narasayya and S. Chaudhuri, "Cloud Data Services: Workloads, Architectures and Multi-Tenancy," *Foundations and Trends® in Databases*, vol. 10, no. 1, 2021, pp. 1–107. DOI: 10.1561/1900000060.

[151] P. Negi, Z. Wu, A. Kipf, N. Tatbul, R. Marcus, S. Madden, T. Kraska, and M. Alizadeh, "Robust Query Driven Cardinality Estimation under Changing Workloads," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, 2023, pp. 1520–1533.

[152] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, 2011, pp. 539–550.

[153] T. Neumann and A. Kemper, *Unnesting Arbitrary Queries*, 2015.

[154] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case Optimal Join Algorithms," *Journal of the ACM (JACM)*, vol. 65, no. 3, 2018, pp. 1–40.

[155] F. Olken, "Random Sampling from Databases," Ph.D. dissertation, Citeseer, 1993.

[156] Oracle, *Oracle Dynamic Sampling*, 2020. URL: https://blogs.oracle.com/optimizer/post/dynamic-sampling-and-its-impact-on-the-optimizer.

[157] Oracle, *Oracle Automatic Workload Repository*, 2023. URL: https://docs.oracle.com/en/database/oracle/oracle-database/23/tgdba/awr-report-ui.html.

[158] Oracle, *Oracle Result Set Caching*, 2024. URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdbc/statement-and-resultset-caching.html#GUID-5D1A9E2F-F191-4FCF-994C-C1D5B143FC4F.

[159] Oracle, *Oracle SQL Tuning Guide: Histograms*, 2024. URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/histograms.html#GUID-BE10EBFC-FEFC-4530-90DF-1443D9AD9B64.

[160] Oracle, *Statistics Best Practices: Oracle*, 2024. URL: https://www.oracle.com/docs/tech/database/technical-brief-bp-for-stats-gather-19c.pdf.

[161] Oracle, *The Optimizer In Oracle Database 19c*, 2024. URL: https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-optimizer-with-oracledb-19c-5324206.pdf.

[162] Oracle, *SQL Plan Management in Oracle database*. URL: https://docs.oracle.com/en-us/iaas/database-management/doc/use-spm-manage-sql-execution-plans.html.

[163] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten, "The Complexity of Transformation-Based Join Enumeration," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, pp. 306–315, 1997.

[164] G. Piatetsky-Shapiro and C. Connell, "Accurate Estimation of the Number of Tuples Satisfying a Condition," *ACM Sigmod Record*, vol. 14, no. 2, 1984, pp. 256–276.

[165] H. Pirahesh, T. Leung, and W. Hasan, "A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS," in *Proceedings 13th International Conference on Data Engineering*, pp. 391–400, 1997. DOI: 10.1109/ICDE.1997.581945.

[166] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/Rule based Query Rewrite Optimization in Starburst," *ACM Sigmod Record*, vol. 21, no. 2, 1992, pp. 39–48.

[167] M. Poess and C. Floyd, "New TPC Benchmarks for Decision Support and Web Commerce," *SIGMOD Rec.*, vol. 29, no. 4, Dec. 2000, pp. 64–71. DOI: 10.1145/369275.369291.

[168] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *ACM Sigmod Record*, vol. 25, no. 2, 1996, pp. 294–305.

[169] *Postgres Query Optimizer*, 2023. URL: https://github.com/postgres/postgres/tree/master/src/backend/optimizer.

[170] *PostgreSQL pg_stats*, 2024. URL: https://www.postgresql.org/docs/current/view-pg-stats.html.

[171] *PostgreSQL: Genetic Algorithms*, 2024. URL: https://www.postgresql.org/docs/current/geqo-pg-intro.html.

[172] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham, "Froid: Optimization of Imperative Programs in a Relational Database," *Proc. VLDB Endow.*, vol. 11, no. 4, Dec. 2017, pp. 432–444. DOI: 10.1145/3186728.3164140.

[173] N. Reddy and J. R. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, pp. 1228–1239, Trondheim, Norway: VLDB Endowment, 2005.

[174] A. van Renen, D. Horn, P. Pfeil, K. Vaidya, W. Dong, M. Narayanaswamy, Z. Liu, G. Saxena, A. Kipf, and T. Kraska, "Why TPC is not enough: An Analysis of the Amazon Redshift fleet," *Proceedings of the VLDB Endowment*, vol. 17, no. 11, 2024, pp. 3694–3706.

[175] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 249–260, 2000.

[176] SAP, *CREATE STATISTICS Statement in SAP Hana*, 2024. URL: https://help.sap.com/docs/SAP_HANA_PLATFORM/4fe29514fd584807ac9f2a04f6754767/20d5252d7519101493f5e662a6cda4d4.html.

[177] T. Schmidt, A. Kipf, D. Horn, G. Saxena, and T. Kraska, "Predicate caching: Query-driven Secondary Indexing for Cloud Data Warehouses," 2024.

[178] 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 *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79, pp. 23–34, Boston, Massachusetts: Association for Computing Machinery, 1979. DOI: 10.1145/582095.582099.

[179] T. K. Sellis, "Multiple-query Optimization," *ACM Transactions on Database Systems (TODS)*, vol. 13, no. 1, 1988, pp. 23–52.

[180] P. Seshadri, H. Pirahesh, and T. Leung, "Complex Query Decorrelation," in *Proceedings of the Twelfth International Conference on Data Engineering*, pp. 450–458, 1996. DOI: 10.1109/ICDE.1996.492194.

[181] S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. DeWitt, and C. Galindo-Legaria, "Query Optimization in Microsoft SQL Server PDW," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 767–776, 2012.

[182] T. Siddiqui, A. Jindal, S. Qiao, H. Patel, and W. Le, "Cost Models for Big Data Query Processing: Learning, Retrofitting, and our Findings," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 99–113, 2020.

[183] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, *et al.*, "Orca: a Modular Query Optimizer Architecture for Big Data," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 337–348, 2014.

[184] SQLShack, *SQL Server Trivial Execution Plans*, 2021. URL: https://www.sqlshack.com/sql-server-trivial-execution-plans/.

[185] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and Randomized Optimization for the Join Ordering Problem," *The VLDB journal*, vol. 6, 1997, pp. 191–208.

[186] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, "LEO - DB2's LEarning Optimizer," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01, pp. 19–28, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[187] M. Stonebraker and L. A. Rowe, "The Design of Postgres," *ACM Sigmod Record*, vol. 15, no. 2, 1986, pp. 340–355.

[188] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et al.*, "C-store: a Column-oriented DBMS," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, Trondheim, Norway: VLDB Endowment, 2005.

[189] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et al.*, "C-store: a Column-oriented DBMS," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, 2018, pp. 491–518.

[190] *Substrait*, 2024. URL: https://github.com/substrait-io/substrait.

[191] J. Sun and G. Li, "An End-to-end Learning-based Cost Estimator," *arXiv preprint arXiv:1906.02560*, 2019.

[192] N. Thaper, S. Guha, P. Indyk, and N. Koudas, "Dynamic Multidimensional Histograms," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 428–439, 2002.

[193] I. Trummer, J. Wang, Z. Wei, D. Maram, S. Moseley, S. Jo, J. Antonakakis, and A. Rayabhari, "Skinnerdb: Regret-bounded Query Evaluation via Reinforcement Learning," *ACM Transactions on Database Systems (TODS)*, vol. 46, no. 3, 2021, pp. 1–45.

[194] K. Vaidya, A. Dutt, V. Narasayya, and S. Chaudhuri, "Leveraging Query Logs and Machine Learning for Parametric Query Optimization," *Proc. VLDB Endow.*, vol. 15, no. 3, Nov. 2021, pp. 401–413. DOI: 10.14778/3494124.3494126.

[195] F. M. Waas and J. M. Hellerstein, "Parallelizing Extensible Query Optimizers," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09, pp. 871–878, Providence, Rhode Island, USA: Association for Computing Machinery, 2009. DOI: 10.1145/1559845.1559938.

[196] X. Wang, C. Qu, W. Wu, J. Wang, and Q. Zhou, "Are We Ready for Learned Cardinality Estimation?" *Proc. VLDB Endow.*, vol. 14, no. 9, May 2021, pp. 1640–1654. DOI: 10.14778/3461535.3461552.

[197] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton, "Predicting Query Execution Time: are Optimizer Cost Models Really Unusable?" In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, IEEE, pp. 1081–1092, 2013.

[198] W. P. Yan and P.-Å. Larson, "Eager Aggregation and Lazy Aggregation," in *Proceedings of the 21th International Conference on Very Large Data Bases*, ser. VLDB '95, pp. 345–357, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995.

[199] J. Yang, S. Wu, D. Zhang, J. Dai, F. Li, and G. Chen, "Rethinking Learned Cost Models: Why Start from Scratch?" *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, 2023, pp. 1–27.

[200] Z. Yang, W.-L. Chiang, S. Luan, G. Mittal, M. Luo, and I. Stoica, "Balsa: Learning a Query Optimizer Without Expert Demonstrations," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, pp. 931–944, Philadelphia, PA, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3514221.3517885.

[201] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica, "NeuroCard: One Cardinality Estimator for all Tables," *arXiv preprint arXiv:2006.08109*, 2020.

[202] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica, "Deep Unsupervised Cardinality Estimation," *arXiv preprint arXiv:1905.04278*, 2019.

[203] M. Yannakakis, "Algorithms for Acyclic Database Schemes," in *VLDB*, vol. 81, pp. 82–94, 1981.

[204] Y. Zhao, G. Cong, J. Shi, and C. Miao, "Queryformer: A Tree Transformer Model for Query Plan Representation," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, 2022, pp. 1658–1670.

[205] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel, "Looking ahead makes Query Plans Robust: Making the initial case with In-memory Star Schema Data Warehouse Workloads," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, 2017, pp. 889–900.

[206] R. Zhu, W. Chen, B. Ding, X. Chen, A. Pfadler, Z. Wu, and J. Zhou, "Lero: A Learning-to-rank Query Optimizer," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, 2023, pp. 1466–1479.

[207] R. Zhu, Z. Wu, Y. Han, K. Zeng, A. Pfadler, Z. Qian, J. Zhou, and B. Cui, "FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation," *arXiv preprint arXiv:2011.09022*, 2020.