Lecture #15: Query Planning & Optimization

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Overview

Because SQL is declarative, the query only tells the DBMS what to compute, but not how to compute it. Thus, the DBMS needs to translate a SQL statement into an executable query plan. But there are different ways to execute each operator in a query plan (e.g., join algorithms) and there will be differences in performance among these plans. The job of the DBMS's optimizer is to pick an optimal plan for any given query.

The first implementation of a query optimizer was IBM System R and was designed in the 1970s. Prior to this, people did not believe that a DBMS could ever construct a query plan better than a human. Many concepts and design decisions from the System R optimizer are still in use today.

Query optimization is the most difficult part of building a DBMS. Some systems have attempted to apply machine learning to improve the accuracy and efficiency of optimizers, but no major DBMS currently deploys an optimizer based on this technique.

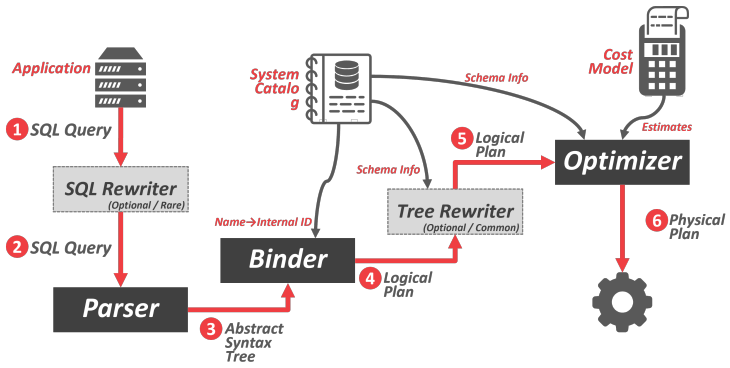

Figure 1: Architecture Overview – The application connected to the database system and sends a SQL query, which may be rewritten to a different format. The SQL string is parsed into tokens that make up the syntax tree. The binder converts named objects in the syntax tree to internal identifiers by consulting the system catalog. The binder emits a logical plan which may be fed to a tree rewriter for additional schema info. The logical plan is given to the optimizer which selects the most efficient procedure to execute the plan.

Logical vs. Physical Plans

The optimizer generates a mapping of a *logical algebra expression* to the optimal equivalent physical algebra expression. The logical plan is roughly equivalent to the relational algebra expressions in the query.

Physical operators define a specific execution strategy using an access path for the different operators in the query plan. Physical plans may depend on the physical format of the data that is processed (i.e. sorting, compression).

There does not always exist a one-to-one mapping from logical to physical plans.

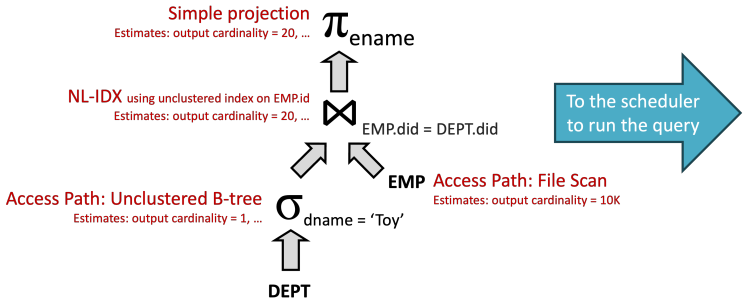

Figure 2: Physical plan as an annotated relational algebra tree: – You could view them as a relational algebra tree that is annotated with the information needed for the physical implementation.

2 Relational Algebra Equivalence

Much of query optimization relies on the underlying concept that the high level properties of relational algebra are preserved across equivalent expressions. Two relational algebra expressions are *equivalent* if they generate the same set of tuples.

This technique of transforming the underlying relational algebra representation of a logical plan is known as *query rewriting*.

One example of relational algebra equivalence is *predicate pushdown*, in which a predicate is applied in a different position of the sequence to avoid unnecessary work. Figure 3 shows an example of predicate pushdown.

3 Types of Query Optimization

There are two high-level strategies for query optimization.

1. The first approach is to use static rules, or *heuristics*. Heuristics match portions of the query with known patterns to assemble a plan. These rules transform the query to remove inefficiencies. Although these rules may require consultation of the catalog to understand the structure of the data, they never need to examine the data itself.

2. An alternative approach is to use *cost-based search* to read the data and estimate the cost of executing equivalent plans. The cost model chooses the plan with the lowest cost.

4 Logical Query Optimization

Transform a logical plan into an equivalent logical plan using pattern matching rules.

Some selection optimizations include:

- Predicate Pushdown - Perform filters as early as possible.
- Reorder predicates so that the DBMS applies the most selective one first.
- Split Conjunctive Predicates - Breakup a complex predicate and pushing it down.
- Replace Cartesian Products with Joins - Transform into inner joins using the join predicates.

An example of predicate pushdown is shown in Figure 3.

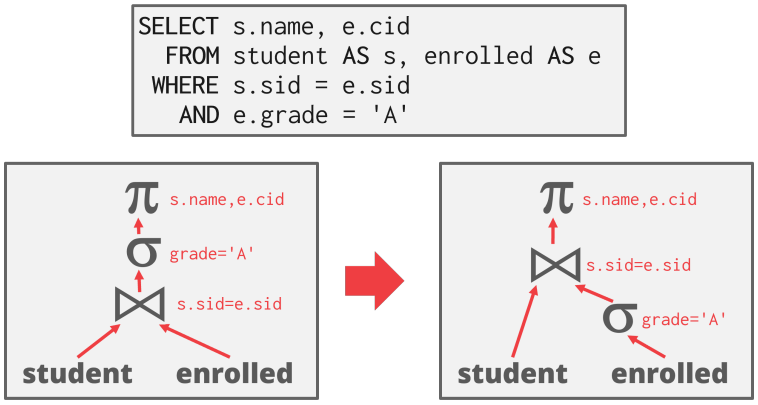

Figure 3: Predicate Pushdown: – Instead of performing the filter after the join, the filter can be applied earlier in order to pass fewer elements into the filter.

Some projection optimizations include:

- Perform projections as early as possible to create smaller tuples and reduce intermediate results *(projection pushdown)*.
- Project out all attributes except the ones requested or requires.
- Optimization from projection pushdown depends on number of columns.

An example of projection pushdown in shown in Figure 4.

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```

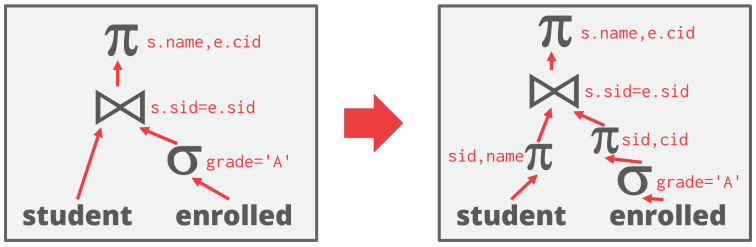

Figure 4: Projection Pushdown – Since the query only asks for the student name and ID, the DBMS can remove all columns except for those two before applying the join.

5 Cost-Based Query Optimization

After performing rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs. It then chooses the best plan for the query after exhausting all plans or some timeout.

6 Cost Estimations

DBMS's use cost models to estimate the cost of executing a plan. These models evaluate equivalent plans for a query to help the DBMS select the most optimal one.

The cost of a query depends on several underlying metrics split between physical and logical costs, including:

- **CPU:** small cost, but tough to estimate.
- **Disk I/O:** the number of block transfers.
- **Memory:** the amount of DRAM used.
- **Network:** the number of messages sent.

Exhaustive enumeration of all valid plans for a query is much too slow for an optimizer to perform. For joins alone, which are commutative and associative, there are 4^n different orderings of every n-way join. Optimizers must limit their search space in order to work efficiently.

To approximate costs of queries, DBMS's maintain internal *statistics* about tables, attributes, and indexes in their internal catalogs. Different systems maintain these statistics in different ways. Most systems attempt to avoid on-the-fly computation by maintaining an internal table of statistics. These internal tables may then be updated in the background.

For each relation R, the DBMS maintains the following information:

- N_R: Number of tuples in R
- $V(A, R)$: Number of distinct values of attribute A

With the information listed above, the optimizer can derive the *selection cardinality* $SC(A, R)$ statistic. The selection cardinality is the average number of records with a value for an attribute A given $\frac{N_R}{V(A,R)}$. Note that this assumes data uniformity. This assumption is often incorrect, but it simplifies the optimization process.

7 Search Termination

Optimizers must decide when to stop exploring new plans. Common termination criteria include:

- Wall-clock Time: Stop after running for a fixed duration.
- Cost Threshold: Stop when a plan below a cost threshold is found.
- Exhaustion: Stop when no more plan enumerations remain.
- Transformation Count: Stop after a fixed number of rule applications.

8 Access Path Transformation

The optimizer selects access methods that minimize data retrieval cost from base relations and may reorder predicate evaluations.

Key cost factors include:

- Predicate selectivity
- Data structures (e.g., B+Tree vs. hash index)
- Sort order of table or index
- Data features (e.g., `INCLUDE`, zone maps)
- Compression and encoding

9 Single-Relation Query Plans

For single-relation query plans, the biggest obstacle is choosing the best access method (i.e., sequential scan, binary search, index scan, etc.) Most new database systems just use heuristics, instead of a sophisticated cost model, to pick an access method.

For OLTP queries, this is especially easy because they are *sargable* (Search Argument Able), which means that there exists a best index that can be selected for the query. This can also be implemented with simple heuristics.

10 Multi-Relation Query Plans

For Multi-Relation query plans, as number of joins increases, the number of alternative plans grow rapidly. Consequently, it is important to restrict the search space so as to be able to find the optimal plan in a reasonable amount of time. There are two ways to approach this search problem:

- **Generative / Bottom-up**: Start with nothing and then build up the plan to get to the outcome that you want. Examples: IBM System R, DB2, MySQL, Postgres, most open-source DBMSs.
- **Transformation / Top-down**: Start with the outcome that you want, and then work down the tree to find the optimal plan that gets you to that goal. Examples: MSSQL, Greenplum, CockroachDB,

Volcano

11 Bottom-up optimization example - System R

Use static rules to perform initial optimization. Then use dynamic programming to determine the best join order for tables using a divide-and conquer search method. Most open source database systems do this.

- Break query up into blocks and generate the logical operators for each block
- For each logical operator, generate a set of physical operators that implement it
- Then, iteratively construct a "left-deep" tree that minimizes the estimated amount of work to execute the plan

12 Top-down optimization example - Volcano

The Volcano optimizer begins with an initial logical query plan. It then explores the plan space by applying transformation rules (logical-to-logical) and implementation rules (logical-to-physical), recursively finding equivalent expressions and optimal implementations for sub-plans. Memoization prevents redundant exploration of identical plans.

- Keep track of global best plan during search.
- Treat physical properties of data as first-class entities during planning.

$\sigma_{P1}(\sigma_{P2}(R)) \equiv \sigma_{P2}(\sigma_{P1}(R))$ (σ commutativity)

$\sigma_{P1 \wedge P2 \ldots \wedge Pn}(R) \equiv \sigma_{P1}(\sigma_{P2}(\ldots \sigma_{Pn}(R)))$ (cascading σ)

$\prod_{a1}(R) \equiv \prod_{a1}(\prod_{a2}(\ldots \prod_{ak}(R)\ldots))$, $a_i \subseteq a_{i+1}$ (cascading $\prod$)

$R \bowtie S \equiv S \bowtie R$ (join commutativity)

$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (join associativity)

$\sigma_P(R \times S) \equiv (R \bowtie_P S)$, if P is a join predicate

$\sigma_P(R \times S) \equiv \sigma_{P1}(\sigma_{P2}(R) \bowtie_{P4} \sigma_{P3}(S))$, where $P = p1 \wedge p2 \wedge p3 \wedge p4$

$\prod_{A1,A2,\ldots An}(\sigma_P(R)) \equiv \prod_{A1,A2,\ldots An}(\sigma_P(\prod_{A1,\ldots An,\ B1,\ldots\ BM}R))$, where B1 … BM are columns in P

…

Figure 5: Example Logical Rules – The set of rules in the system defines the search space the optimizer operates on.

13 Nested Sub-Queries

The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.

- Re-write the query by de-correlating and / or flattening nested subqueries. An example of this is shown in Figure 6.
- Decompose the nested query and store the result to a temporary table. An example of this is shown in Figure 7.

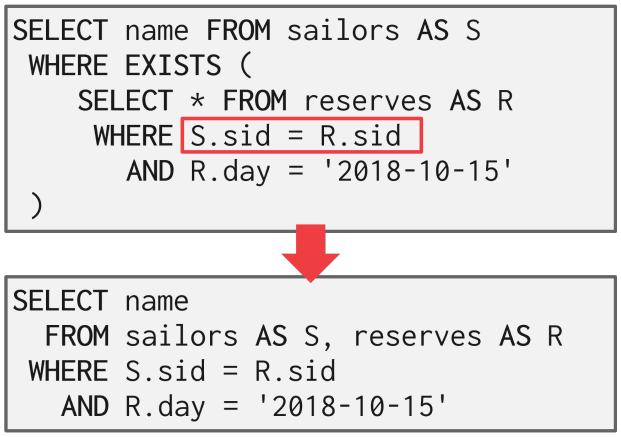

Figure 6: Subquery Optimization - Rewriting The former query can be rewritten as the latter query by rewriting the subquery as a `JOIN`. Removing a level of nesting in this way effectively *flattens* the query.

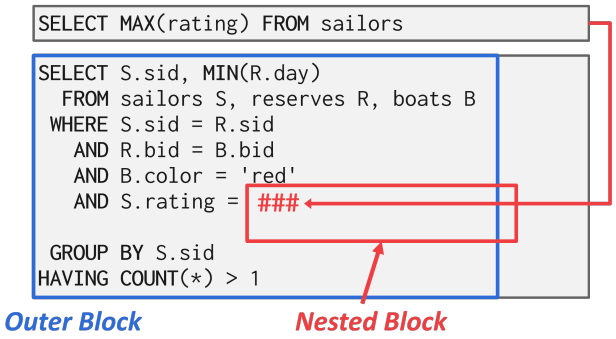

Figure 7: Subquery Optimization - Decomposition – For complex queries with subqueries, the DBMS optimizer may break up the original query into blocks and focus on optimizing each individual block at a a time. In this example, the optimizer decomposes a query with a nested aggregation by pulling the nested query out into its own query, and subsequently using this result to realize the logic of the original query.

14 Expression Rewriting

An optimizer transforms a query's expression (e.g. WHERE/ON clause predicates) into a minimal set of expressions.

- Search for expressions that match a pattern.
- When a match is found, rewrite the expression.
- Halt if there are no more rules that match.

Some examples of expression rewriting

- Impossible predicates as shown in Figure 8.
- Merging predicate as shown in Figure 9.

Figure 8: Impossible Predicates – Evaluate the expression if possible at optimization time.

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 100
    OR val BETWEEN 50 AND 150;
```

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 150;
```

Figure 9: Merging Predicates – The `WHERE` predicate in query 1 has redundancy as what it is searching for is any value between 1 and 150. Query 2 shows the more succinct way to express request in query 1.