Lecture #02: IBM System R

15-799 Special Topics in Databases: Query Optimization (Spring 2025)

https://15799.courses.cs.cmu.edu/spring2025/

Carnegie Mellon University
Prepared By: Anna Lee

1 Background

In the late 1960s, early DBMSs required developers to write queries using **procedural** code. The programmer acted as a *navigator* by specifying the access paths and execution ordering according to the current database contents. If database contents change, the programmer must rewrite their query code for correctness and/or efficiency.

The **relational model** enabled programmers to use a high-level **declarative** language for accessing and modifying the database. This allowed the DBMS, not the programmer, to decide how to execute queries, meaning queries did not have to be rewritten all the time.

Some early relational DBMS implementations are shown below:

- **Peterlee Relational Test Vehicle**: IBM Research (UK)
- **System R**: IBM Research (San Jose)
- **INGRES**: U.C. Berkeley
- **Oracle**: Larry Ellison
- **Mimer**: Uppsala University

Because the relational model's declarative API requires the DBMS to find the best way to execute a query, all of these systems eventually had to implement a query optimizer.

Throughout the semester, we will discuss different strategies for query optimization, such as:

- **Heuristics**: INGRES (1970s), Oracle (until mid 1990s)
- **Heuristics + Cost-based Join Search**: System R (1970s), early IBM DB2
- **Stratified Search**: IBM STARBURST (late 1980s), now IBM DB2 + Oracle
- **Unified Search**: Volcano/Cascades in 1990s, now MSSQL + Greenplum
- **Randomized Search**: Academics in 1980s, current Postgres

These notes focus on the earliest strategies: **heuristic-based optimization** and **cost-based search**.

2 Heuristics

Heuristic-based optimization defines static rules that transform logical operators into a physical plan without using a cost model. Because the optimizer does not have a cost model for comparing query plans, the rules are derived from domain knowledge to describe transformations that are known to be better in general (e.g., predicate pushdown).

Most new database systems today begin with a heuristic-based optimizer because it is easier to implement and debug. This approach also works reasonably well for simple queries. However, many implementations rely on magic constants to make decisions and cannot control the order in which transformations are applied. As a result, they struggle when operators have complex inter-dependencies.

2.1 Relational Algebra Equivalences

Relational algebra equivalency rules provide a way to reason about the correctness of individual transformations. We say that two relational algebra expressions are **equivalent** if they generate the same set of tuples. Using these equivalences, the DBMS can manipulate and transform a query while ensuring that the output is still correct. We will focus on equivalences for selection (σ) and join ($\bowtie$).

Selection (σ)

The following are some useful equivalences commonly used in transformations:

- We can move a filter in the query plan, so long as the predicate remains in scope of any relations it references.
- We can break complex predicates into predicates for each conjunctive clause. This can be expressed with the following equivalence:

$$\sigma_{p1 \wedge p2 \wedge \cdots \wedge pn}(R) = \sigma_{p1}(\sigma_{p2}(\ldots \sigma_{pn}(R)))$$

- We can simplify complex predicates (e.g., by precomputing expressions with a constant value). The following are examples of potential simplifications:
  - `X=Y AND Y=3` $\rightarrow$ `X=3 AND Y=3`
  - `X=1+1` $\rightarrow$ `X=2`
  - `X=YEAR('1/15/2025')` $\rightarrow$ `X=2025`

Join ($\bowtie$)

Joins are **commutative**, meaning we can swap the sides of the join:

$$R \bowtie S = S \bowtie R$$

Joins are also **associative**, meaning nested joins can occur in any order:

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

These two properties mean that the number of equivalent join orderings for an n-way binary join is $O((n-1)!\ C_{n-1}) \approx O((n-1)!\ 4^n\ n^{-3/2})$ (where C_n is the nth Catalan number). A query optimizer cannot exhaustively look at every join ordering, so it must employ clever tricks to cut down on its search space.

2.2 Logical Query Optimization

Consider the following query, which retrieves the names of people that appear on the DJ Mooshoo Tribute mixtape, ordered by their artist ID:

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
 ORDER BY ARTIST.ID
```

The following are some examples of rules that a heuristic-based optimizer might apply.

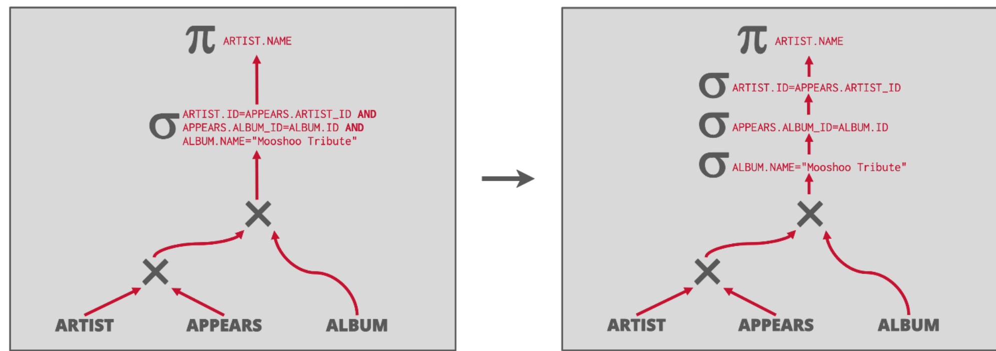

Figure 1: Split Conjunctive Predicates — The complex predicate is decomposed into three conjuncts that can be pushed down independently.

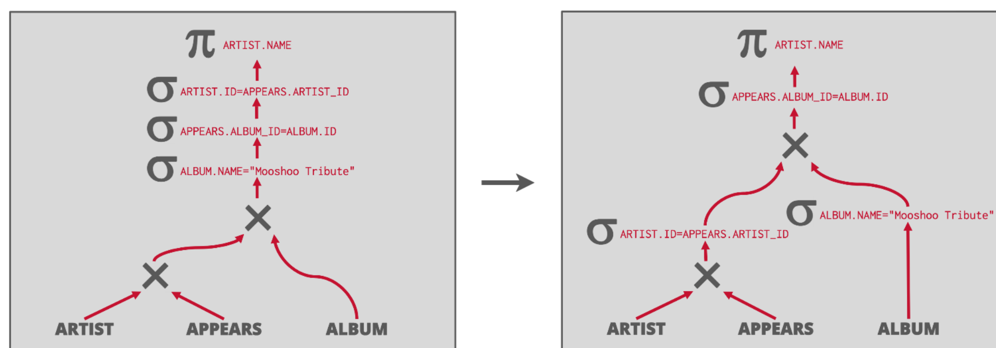

Figure 2: Predicate Pushdown — Each predicate is pushed down as far as possible. Notice that the filter `ARTIST.ID=APPEARS.ARTIST_ID`, for example, is not pushed past the Cartesian product joining the two relations it references.

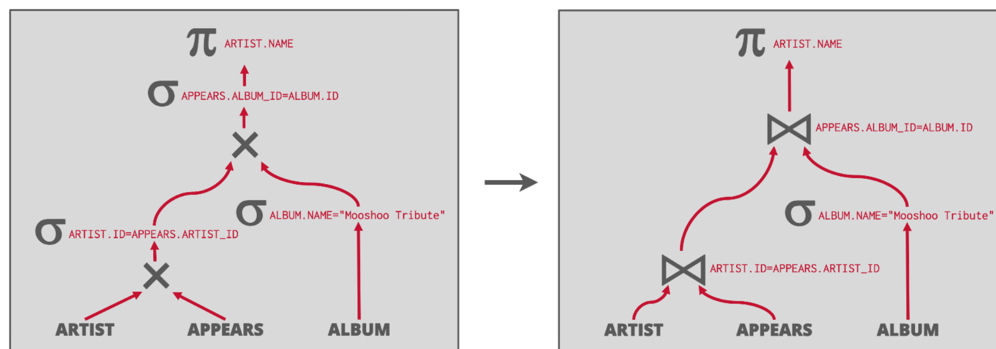

Figure 3: Replace Cartesian Products — If an appropriate filter sits above a Cartesian product, we can replace it with an inner join.

- **Split Conjunctive Predicates**: Decompose predicates into their simplest forms to make it easier for the optimizer to move them around (Figure 1).
- **Predicate Pushdown**: Move the predicate to the lowest possible point in the plan after Cartesian products (Figure 2).
- **Replace Cartesian Products**: Replace all Cartesian products with inner joins using the join predicates (Figure 3).

- **Projection Pushdown**: Eliminate redundant attributes before pipeline breakers to reduce materialization cost. The efficacy of this rule depends on the implementation of the system.

Since these transformations use relational algebra equivalences, they are guaranteed to produce correct queries. Also notice that none of these rules require a cost model — they are automatically applied when conditions are met.

2.3 Example: INGRES Optimizer

So far, our example query has been optimized to use two inner join operators (Figure 3). Next, we will cover one way to determine and execute join order without a cost model or statistics.

To implement joins and select a join ordering, INGRES used the following algorithm:

1. **Decompose into single-value queries.** Decompose the complex query into *single-value queries*, and direct the outputs of the inner queries into temporary tables. A **single-value query** is one that accesses only one "real" relation, and potentially a temporary table.

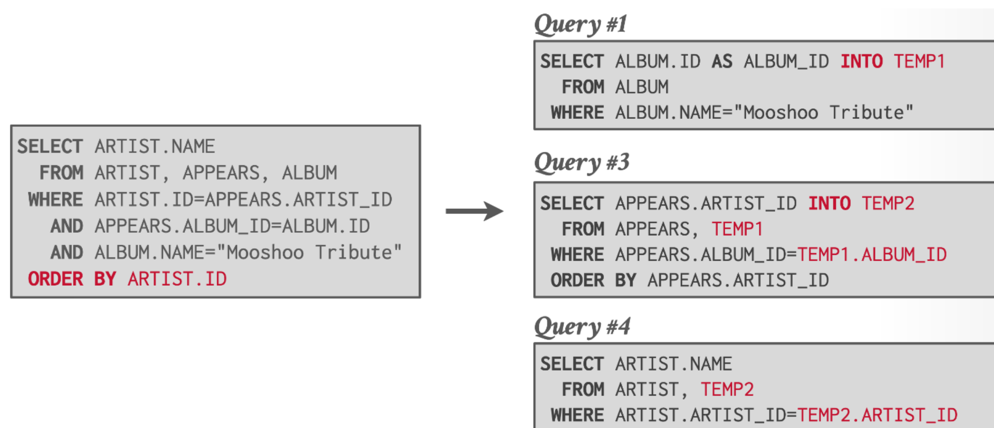

Figure 4: Decomposition into single-value queries. Each query references one relation in the `FROM` clause, and potentially one temporary relation.

2. **Substitute the values into the queries.** Run the queries in order of their dependencies, substituting values that reference temporary tables.

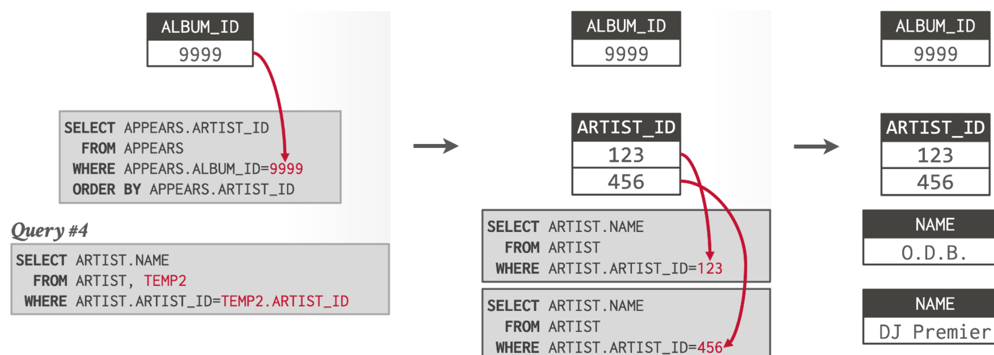

Figure 5: Substitution into dependent queries. A single-value query is run for each value it depends on — for example, query 4 is run twice, once for each artist ID produced by query 3.

This algorithm provides a way to implement joins, but cannot control their order. Controlling join order can greatly improve the speed of a query, but an optimizer needs a cost model to evaluate potential orderings.

3 Heuristics + Cost-Based Search

Heuristic-based optimization with **cost-based search** first uses static rules to perform initial logical-to-logical transformations, and then uses a cost model to find the best logical-to-physical transformations.

Physical query optimization transforms a query plan's logical operators to physical operators. The optimizer must specify its execution strategy for the operators in the query plan. It must decide which join algorithms to use, what access paths to select, and when to materialize data while going from one operator to the next. The optimizer relies on a *cost model* to evaluate potential plans.

3.1 Example: System R Optimizer

The System R optimizer breaks queries into blocks and optimizes each query block individually [1]. A **query block** comprises a single `SELECT`, a `FROM` potentially referencing multiple tables, and a `WHERE` tree representing the predicates. Nested queries are treated and optimized as separate queries.

For each query block, the optimizer generates logical operators using the heuristics defined in Section 2. It then generates the set of physical operators that implement them. It uses a greedy search to find the best access path, and a dynamic programming approach to find the best join ordering.

Single Relation Queries

Query blocks that only reference a single relation are relatively easy to optimize because because they are *sargable*. A **sargable** predicate is one that can be expressed as a SARG, or search argument, meaning it can be looked up in an index[1]. For single relation query blocks, the optimizer just needs to decide what the best access method is — sequential scan or index scan — using a cost model.

Cost Model System R's **cost model** measures the cost of an access method by the following formula:

$$\texttt{COST} = \texttt{Page Fetches} + w(\texttt{RSI Calls})$$

This equation takes into account the expected amount of I/O ("page fetches") and computational cost ("RSI calls"), where w is an adjustable weighting factor between I/O and CPU.

To estimate these values, the DBMS keeps track of statistics about each of the relations and uses *selectivity factors*. A **selectivity factor** of a predicate is a rough estimate of the expected fraction of tuples that will satisfy it. Selectivity factors are computed using formulae depending on the predicate, and by making assumptions about the distribution of the data. For example, the formula for the predicate `COLUMN=VALUE` assumes an even distribution of the data, and assumes the selectivity is $\frac{1}{10}$ if there is no index on the column.

The DBMS cuts down on its search space by only evaluating certain data orderings. An **interesting order** is one that is required by a query block's `GROUP BY` or `ORDER BY` clauses. The optimizer compares the best access method that orders the data against the cost of the best access method that does not order the data *and* ordering the data as required. If there is no required ordering, then the optimizer selects the cheapest access method.

[1] Formally, a sargable predicate is defined as a predicate that can be expressed as a column, a comparison operator, and a value.

Multi Relation Queries

If the query block access multiple relations, the optimizer must determine the best ordering to join the relations. System R uses a **bottom-up** approach by iteratively building a search tree using the following steps:

1. Choose the best access paths to each relation.
2. Enumerate all join orderings for 1-relation plans using the best access path found in Step 1.
3. For each subsequent path, determine the best way to join the result of an $(n-1)$-relation plan as the outer relation to the nth relation.

Since this algorithm is a bottom-up approach, there is no need to maintain history, as the previous levels of the tree implicitly remember the path taken.

Recall that exhaustively searching all possible join orderings would be exponentially slow. System R cuts down on its search space by delaying or discarding plan choices — for example, only searching left-deep trees. This prunes the search tree so that it can find a reasonably fast query plan.

Join costs are estimated based on the number of tuples processed in each of the relations, and again, interesting orders are used to evaluate the cost of a naturally correctly ordered output against having to sort at the end of the plan.

Nested Queries

Since System R optimizes each query block individually, nested queries are treated as separate queries. Similar to how INGRES implemented joins, System R's optimizer executes an inner query before it plans an outer query. It then either substitutes values into the outer query or materializes the results into a temporary table.

Heuristic + cost-based search optimizers can usually find a reasonable plan without having to perform an exhaustive search, but suffer from the same issues as the heuristic-only approach. Additionally, cutting down on the search space by examining only left-deep trees might not produce optimal results, and the cost model must take into account the physical properties of the data, such as its sort order.

References

[1] 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*, SIGMOD '79, pages 23–34, 1979. doi: 10.1145/582095.582099.