Lecture #16: Query Planning & Optimization II

15-445/645 Database Systems (Fall 2026)
https://15445.courses.cs.cmu.edu/fall2026/
Carnegie Mellon University
Jignesh Patel

1 Overview

A DBMS's query optimizer takes a logical query plan as input and generates the physical plan that it estimates has the lowest *cost*. The quality of its plans depends mostly on three factors: the **transformations** it applies (covered in the previous lecture), the **search algorithm** it uses to explore equivalent plans, and the **cost model** it uses to compare them. This lecture covers the last two.

2 Search Algorithms

There are two high-level approaches for searching the space of possible plans. A **bottom-up** (forward chaining) optimizer starts with nothing and iteratively assembles building blocks into a query plan (e.g., System R, Starburst). A **top-down** (backward chaining) optimizer starts with the outcome that the query wants and transforms it into equivalent sub-plans to find the optimal plan that reaches that goal (e.g., Volcano, Cascades). The examples below both use the following query:

```
SELECT ARTIST.NAME FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID = APPEARS.ARTIST_ID AND APPEARS.ALBUM_ID = ALBUM.ID
   AND ALBUM.NAME = 'Andy''s OG Remix'
 ORDER BY ARTIST.ID
```

Bottom-Up Optimization

A bottom-up optimizer first applies static rules (e.g., predicate pushdown), and then uses dynamic programming to determine the best join order with a divide-and-conquer search. This is the approach used by IBM System R, DB2, MySQL, DuckDB, Postgres, and most other open-source DBMSs. The System R optimizer breaks the query into blocks, generates the logical operators for each block, and then generates the physical operators that implement each logical operator (all combinations of join algorithms and access paths). For the example query, it works in three steps:

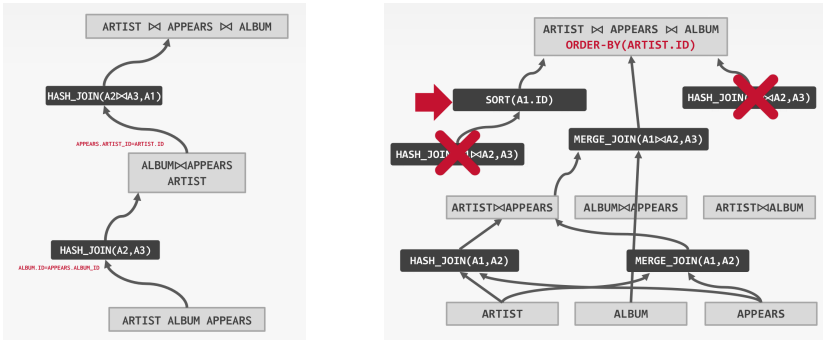
1. **Choose the best access path for each table**: sequential scans for `ARTIST` and `APPEARS`, and an index lookup on `NAME` for `ALBUM`.
2. **Enumerate all possible join orderings**, including orderings that require a Cartesian product (e.g., `ARTIST` $\times$ `ALBUM` $\bowtie$ `APPEARS`).
3. **Determine the join ordering with the lowest cost.** Starting from the base tables, the optimizer considers every physical join algorithm for each pair of tables, keeps only the cheapest operator for each intermediate result, and repeats this for the next level of joins. Figure 1a shows the cheapest plan.

The query has an `ORDER BY` on `ARTIST.ID`, but the logical plans do not contain any sorting properties. The optimizer therefore keeps track of the best plans both with and without sorted output, and then checks whether adding a sort operator at the end is cheaper.

Top-Down Optimization

A top-down optimizer starts with a logical plan of what we want the query to be, and then performs a branch-and-bound search that converts logical operators into physical operators. It keeps track of the global best plan during the search, and it treats the physical properties of data (e.g., sort order) as first-class entities. Examples include Microsoft SQL Server, Greenplum, CockroachDB, and Apache Calcite.

The optimizer invokes rules to create new nodes: **logical→logical** rules (e.g., `JOIN(A,B)` to `JOIN(B,A)`) and **logical→physical** rules (e.g., `JOIN(A,B)` to `HASH_JOIN(A,B)`). *Enforcer* rules require the input to an operator to have certain properties, such as being sorted, and add an operator (e.g., a sort) when a sub-plan does not provide them. The optimizer prunes any branch whose cost already exceeds that of the best complete plan found so far, and it memoizes the best plan for each sub-expression (see Figure 1b).



(a) System R **(b)** Top-down

Figure 1: Search Algorithms – Light boxes are logical operators and dark boxes are physical operators. (a) System R hash joins `ALBUM` (`A2`) with `APPEARS` (`A3`), and then hash joins the result with `ARTIST` (`A1`). (b) The root requires output sorted on `ARTIST.ID`, and a sort-merge join gives the best plan so far. A hash join below the root cannot provide the sort order, and a hash join under a `SORT` enforcer costs more, so both are pruned. Here, `A2` is `APPEARS` and `A3` is `ALBUM`.

3 Cost Estimation

Earlier lectures gave cost formulas for operator algorithms (e.g., hash join and sort-merge join), but these formulas depend on the size of each operator's input. Except for the base tables, an operator's input is another operator's output, so the optimizer must also estimate the size of each operator's output. This is hard because the output of each operator depends on its input.

The DBMS uses a *cost model* to predict the behavior of a query plan given a database state. This internal cost only needs to let the DBMS compare one plan with another, and the DBMS must derive it because running every possible plan is too expensive. A cost model can use **physical costs** (CPU cycles, I/O, cache misses, RAM consumption, network messages), which depend heavily on the hardware, or **logical costs** (the estimated output size of each operator), which are independent of the operator algorithm.

For example, Postgres combines CPU and I/O costs that are weighted by "magic" constant factors. The defaults assume a disk-resident database without a lot of memory: processing a tuple in memory is 400× faster than reading a tuple from disk, and sequential I/O is 4× faster than random I/O. The Postgres documentation warns that there is no well-defined method for determining ideal values for these constants.

4 Data Statistics

To estimate the size of operator outputs, the DBMS maintains *statistical summaries*. These are auxiliary data structures that the DBMS populates by scanning the database. Choosing a summary involves trade-offs between accuracy, efficiency, memory consumption, coverage, and creation and maintenance costs.

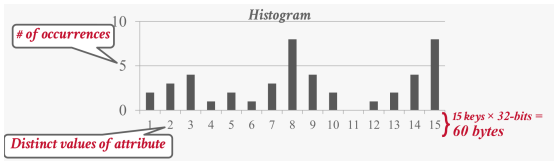
Most DBMSs store statistics in their internal catalog. They update them through periodic background tasks (e.g., Postgres's autovacuum), maintenance schedules (e.g., Oracle), modification thresholds, or manual invocation (e.g., `ANALYZE` or `UPDATE STATISTICS`). Most DBMSs create single-column statistics for each column, and they can also track statistics for groups of columns together. Some systems (e.g., SQL Server) build these automatically for columns that are used together in an index; otherwise, a human must specify the columns. These are also called *column group statistics* (DB2) or *extended statistics* (Oracle).

There are four main ways to summarize the data in a column: **histograms** (most common), **sketches** (increasing usage), **sampling** (rare), and **machine learning models** that learn the selectivity of predicates and correlations between tables (experimental and very rare).

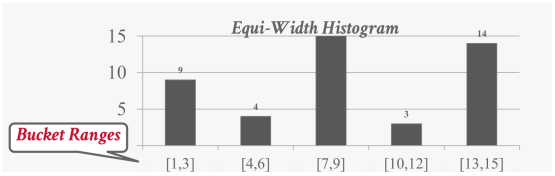
Histograms

A *histogram* approximates the distribution of values in a column. Real data is often skewed, so the DBMS cannot assume that every value is equally common. The simplest histogram stores the count of every distinct value (Figure 2a), but this needs space proportional to the number of distinct values. To reduce this overhead, the DBMS groups values into *buckets*:

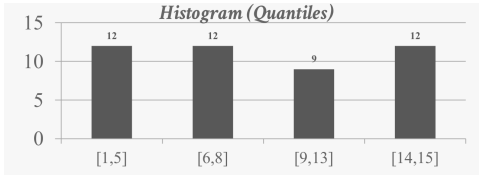
- **Equi-Width**: All buckets cover the same number of values (Figure 2b).
- **Equi-Depth**: The bucket widths vary so that each bucket has roughly the same total count (Figure 2c). These have better worst-case and average error than equi-width histograms.
- **End-Biased**: $N - 1$ buckets store the exact counts of the most frequent keys, and the last bucket (R) stores the average frequency of all remaining values (Figure 2d).



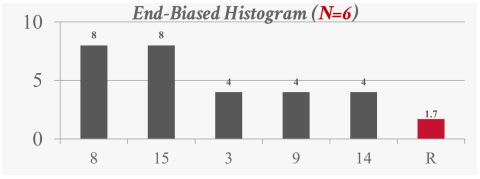
(a) Exact counts



(b) Equi-width



(c) Equi-depth



(d) End-biased

Figure 2: Histograms – (a) 45 tuples with 15 distinct values. (b) Bucket counts of 9, 4, 15, 3, and 14. (c) Four buckets with roughly equal counts. (d) The last bucket stores the average frequency of the remaining ten values, $17/10 = 1.7$.

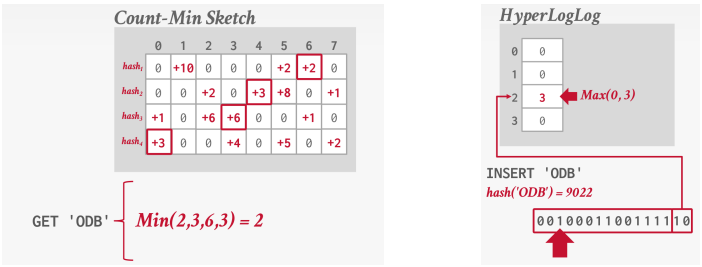
Sketches

Maintaining exact statistics is expensive and slow, so the DBMS can use probabilistic data structures called *sketches* to generate error-bounded estimates. Examples include Count-Min Sketch (frequent items), Hy-

perLogLog (count distinct), and t-digest (quantiles). Open-source implementations include Apache DataSketches and Google ZetaSketch.

Count-Min Sketch approximates the frequency of elements using d hash functions and an array of counters with d rows and w columns, one row per hash function. To insert x, the DBMS increments the counter at column $hash_i(x) \bmod w$ in every row i. To look up x, it returns the minimum of those d counters (Figure 3a). Collisions only add to counters, so the sketch can overestimate a frequency but never underestimate it. Adding rows and columns improves accuracy at the cost of space.

HyperLogLog approximates the number of distinct elements using $m = 2^b$ counters. To insert an element, the DBMS hashes it and uses the last b bits of the hash to pick a counter. It then sets that counter to the maximum of its current value and the position of the leftmost 1-bit in the remaining bits (Figure 3b). To estimate the number of distinct elements, it computes the harmonic mean across the counters and corrects the result with a constant fudge factor.



(a) Count-Min Sketch **(b)** HyperLogLog

Figure 3: Sketches – (a) Looking up `ODB` returns $\min(2, 3, 6, 3) = 2$. (b) The last two bits of $hash(\texttt{ODB}) = 9022$ select counter 2, and the leftmost 1-bit of the remaining bits is at position 3, so the counter becomes $\max(0, 3) = 3$.

Sampling

The DBMS can also estimate the selectivity of a predicate by executing it on a random sample of the table (Figure 4). The number of tuples to examine depends on the size of the table. The DBMS can either maintain a read-only copy of the sample, which it periodically refreshes, or sample the real table, which may read multiple versions of the same logical tuple. It updates its samples when the number of changes to the table exceeds some threshold.

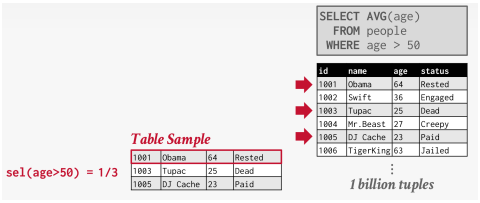


Figure 4: Sampling – One of the three sampled tuples satisfies `age > 50`, so the estimated selectivity is $1/3$.

5 Cardinality Estimation

Cardinality estimation predicts the number of tuples that an operator (e.g., a filter or a join) will produce. The core of a cost model must support three kinds of estimates: selection conditions, join sizes, and distinct

values.

For each relation R, the DBMS maintains statistics to approximate N_R, the number of tuples in R, and $V(A, R)$, the number of distinct values of attribute A. From these, it derives the *selection cardinality* $SC(A, R) = N_R/V(A, R)$, which is the average number of tuples with a given value of A.

Single Selection Conditions

The *selectivity* (sel) of a predicate P is the fraction of tuples that qualify. The examples below use a `people` table with 45 tuples and the equi-depth histogram on `age` from Figure 2c.

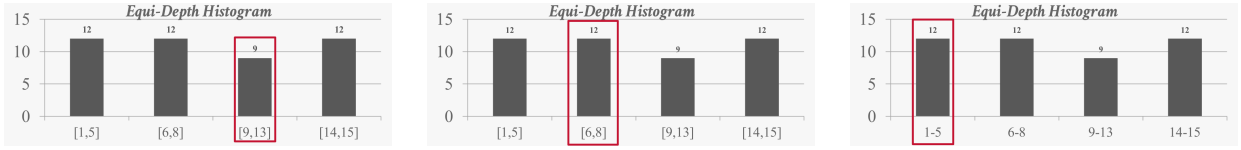
Equality Predicate: $\text{sel}(A = c) = \#\text{ occurrences of } c \,/\, |R|$. If the DBMS knows the exact counts, then $\text{sel}(\texttt{age} = 9) = 4/45 \approx 0.088$. With only the equi-depth histogram, it knows that the bucket $[9, 13]$ has 9 tuples across 5 values, so it assumes that each value occurs $9/5 = 1.8$ times. The estimate is then $\text{sel}(\texttt{age} = 9) \approx 1.8/45 \approx 0.04$. This relies on **Assumption #1: Uniform Data**, which states that the distribution of values (except for the heavy hitters) is the same within a histogram bucket.

Range Predicate: $\text{sel}(A \geq a)$ is the number of tuples with $A > a$ plus the number with $A = a$, divided by $|R|$. For $\texttt{age} \geq 7$, all of the tuples in the buckets $[9, 13]$ and $[14, 15]$ qualify. The bucket $[6, 8]$ has 12 tuples across 3 values, and two of those values qualify. Assuming a continuous distribution of values within the bucket,

$$\text{sel}(\texttt{age} \geq 7) \approx \frac{(9 + 12) + (2 \times (12/3))}{45} = \frac{29}{45} \approx 0.644$$

Negation Query: $\text{sel}(\text{not } P) = 1 - \text{sel}(P)$. The bucket $[1, 5]$ has 12 tuples across 5 values, so

$$\text{sel}(\texttt{age} \neq 2) \approx 1 - \frac{12/5}{45} = 1 - \frac{2.4}{45} \approx 1 - 0.053 \approx 0.95$$



(a) $\texttt{age} = 9$ **(b)** $\texttt{age} \geq 7$ **(c)** $\texttt{age} \neq 2$

Figure 5: Selectivity Estimation – The highlighted bucket is the one that the DBMS must split under the uniform data assumption.

Observe that the selectivity of a predicate is equivalent to the probability that a random tuple satisfies it. This allows the DBMS to apply the rules of probability to complex predicates.

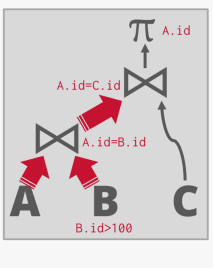
Multiple Selection Conditions

Consider `WHERE age = 2 AND name LIKE 'A%'`, where $\text{sel}(\texttt{age} = 2) \approx 0.053$ and $\text{sel}(\texttt{name LIKE 'A\%'}) \approx 0.1$. To combine them, the DBMS relies on **Assumption #2: Independent Predicates**, which states that the selectivity of a conjunction of predicates is the product of their individual selectivities.

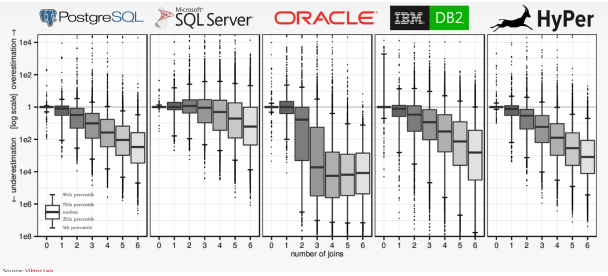
Conjunction: $\text{sel}(P_1 \wedge P_2) = \text{sel}(P_1) \times \text{sel}(P_2)$, so the example's selectivity is $0.053 \times 0.1 \approx 0.0053$. Since this product shrinks quickly as predicates are added, some systems diminish the weight of each additional predicate to reduce underestimation.

Disjunction: $\text{sel}(P_1 \vee P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1) \times \text{sel}(P_2)$, so the selectivity of `age = 2 OR name LIKE 'A%'` is $0.053 + 0.1 - (0.053 \times 0.1) \approx 0.1477$. This again assumes that the predicates are independent.

Correlated Attributes: The independence assumption often fails for real data. Consider a database of automobiles with 10 makes and 100 models, and the predicate `make = 'Honda' AND model = 'Accord'`. With the independence and uniformity assumptions, the estimated selectivity is $1/10 \times 1/100 = 0.001$. However, only Honda makes Accords, so the real selectivity is $1/100 = 0.01$, which is $10\times$ higher. Multi-column statistics help the DBMS avoid this kind of error.



(a) Plan



(b) Estimator quality (source: Viktor Leis)

Figure 6: Estimation Errors – (a) Each join's estimate builds on the estimates below it. (b) The ratio of estimated to actual cardinality (log scale) by number of joins, with one panel per DBMS. Values below 1 are underestimates.

Join Size Estimation

For a join of R and S, the DBMS needs to estimate how many tuples of S each tuple of R will match. It relies on **Assumption #3: Containment Principle**, which states that the domains of the join keys overlap such that each key in the inner relation also exists in the outer table.

Consider the general case where R and S share an attribute A that is not a primary key for either table. Each tuple of R matches about $N_S/V(A,S)$ tuples of S, and each tuple of S matches about $N_R/V(A,R)$ tuples of R. The DBMS uses the smaller of the two resulting estimates:

$$\text{estSize} \approx \frac{N_R \times N_S}{\max(V(A,S), V(A,R))}$$

Error Propagation

The optimizer computes each operator's cardinality from the estimated cardinalities of its inputs, so errors propagate up the plan. Consider a query that joins tables `A`, `B`, and `C` on `id` and filters on `B.id > 100` (Figure 6a). The estimate for the filtered `B` ($|B| \times \text{sel}(\texttt{B.id} > 100)$) feeds into the estimate for $A \bowtie B$, which in turn feeds into the estimate for $(A \bowtie B) \bowtie C$. If the filter's selectivity is off by $10\times$, both join estimates inherit that error, so errors compound as the number of joins grows.

In practice, these errors are large. A study by Leis et al.[1] measured the cardinality estimates of PostgreSQL, HyPer, and three major commercial DBMSs on queries over the real-world IMDB dataset. For every system, the estimates get worse as the number of joins increases, and they are usually underestimates that can be off by several orders of magnitude (Figure 6b).

[1] `https://www.vldb.org/pvldb/vol9/p204-leis.pdf`