

Lecture #16: Query Planning & Optimization II

15-445/645 Database Systems (Spring 2026)

<https://15445.courses.cs.cmu.edu/spring2026/>

Carnegie Mellon University

Andy Pavlo

1 Multi-Relation Query Plans

For Multi-Relation query plans, as number of joins increases, the number of alternative plans grows 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

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

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. During this search, a branch can be pruned early if its estimated cost already exceeds the lowest-cost plan found so far. Memoization prevents redundant exploration of identical plans. Enforcers are physical operators that ensure the properties of the output of a sub-plan or expression.

2 Data Statistics

Histograms

Real data is often skewed and is tricky to make assumptions about. However, storing every single value of a data set is expensive. One way to reduce the amount of memory used is by storing data in a *histogram* to group together values. An example of a graph with buckets is shown in Figure 1.

Another approach is to use a *equi-depth* histogram that varies the width of buckets so that the total number of occurrences for each bucket is roughly the same. An example is shown in Figure 2.

If certain keys occur frequently, one approach is to use an *end-biased* histogram that uses N-1 buckets to store the exact count for the most frequent keys and uses the last bucket to store the average frequency of all remaining values. An example is shown in Figure 3.

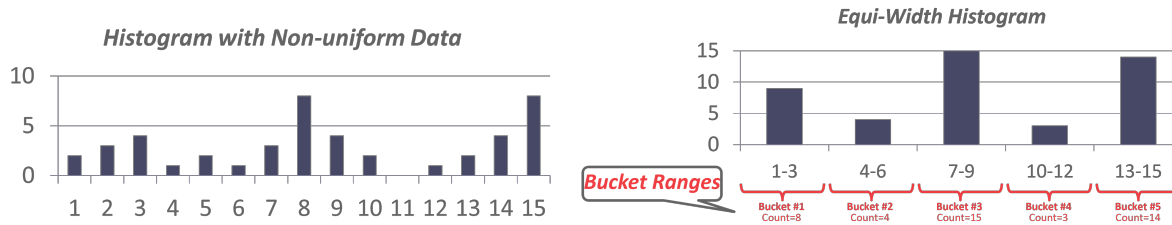


Figure 1: Equi-Width Histogram: The first figure shows the original frequency count of the entire data set. The second figure is an equi-width histogram that combines together the counts for adjacent keys to reduce the storage overhead.

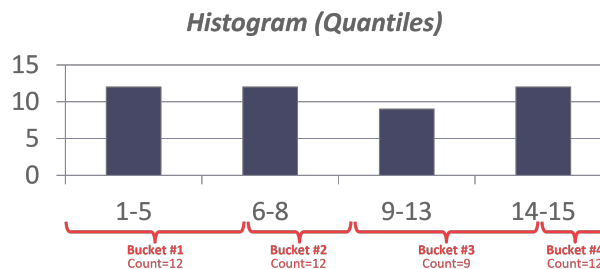


Figure 2: Equi-Depth Histogram – To ensure that each bucket has roughly the same number of counts, the histogram varies the range of each bucket.

In place of histograms, some systems may use *sketches* to generate approximate statistics about a data set, including Count-Min Sketch and HyperLogLog.

Sampling

DBMSs can use *sampling* to apply predicates to a subset of the table with a similar distribution (see Figure 4). There are two approaches to do this: maintaining read-only copy of the sample, or sampling real tables. The DBMS updates the sample whenever the amount of changes to the underlying table exceeds some threshold (e.g., 10% of the tuples).

3 Cost Estimations

DBMSs 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 both physical and logical cost estimates:

- **Physical costs:** CPU work, disk I/O, memory usage, and network communication.
- **Logical costs:** estimated output sizes (cardinalities) of operators, which are used to predict the cost of later operators in the plan.

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, DBMSs 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.

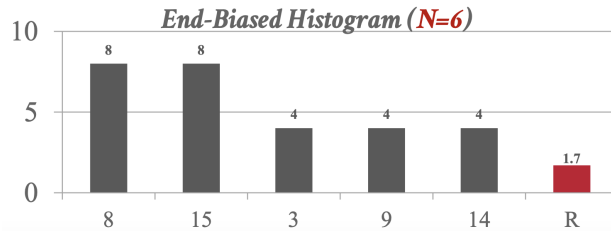


Figure 3: End-Biased Histogram – The exact counts for the most frequent keys are stored in the first N-1 buckets. The last bucket (R) stores the average frequency of all remaining values.

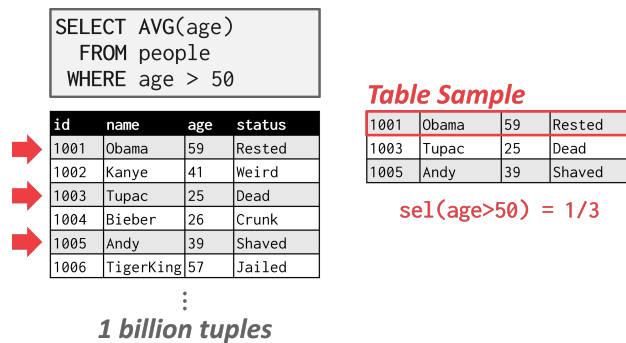


Figure 4: Sampling – Instead of using one billion values in the table to estimate selectivity, the DBMS can derive the selectivities for predicates from a subset of the original table.

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.

Selection Statistics

The selection cardinality can be used to determine the number of tuples that will be selected for a given input.

Equality predicates on unique keys are simple to estimate (see Figure 5). A more complex predicate is shown in Figure 6.

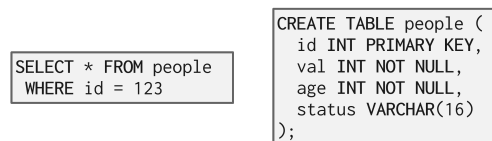


Figure 5: Simple Predicate Example – In this example, determining what index to use is easy because the query contains an equality predicate on a unique key.

```
SELECT * FROM people
WHERE val > 1000
```

```
SELECT * FROM people
WHERE age = 30
AND status = 'Lit'
AND age+id IN (1,2,3)
```

Figure 6: Complex Predicate Example – More complex predicates, such as range or conjunctions, are harder to estimate because the selection cardinalities of the predicates must be combined in non-trivial ways.

The *selectivity* (sel) of a predicate P is the fraction of tuples that qualify. The formula used to compute selectivity depends on the type of predicate. Selectivity for complex predicates is hard to estimate accurately which can pose a problem for certain systems. An example of a selectivity computation is shown in Figure 7.

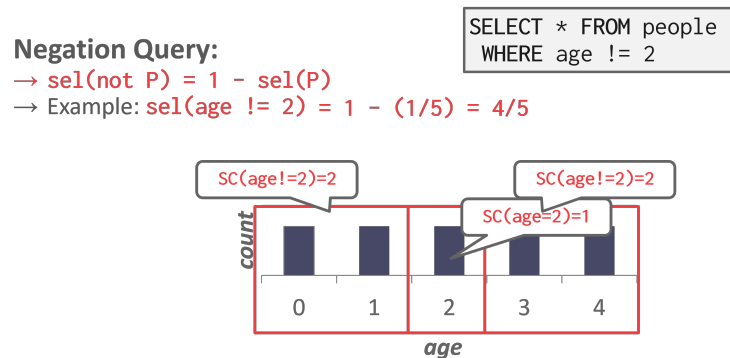


Figure 7: Selectivity of Negation Query Example – The selectivity of the negation query is computed by subtracting the selectivity of the positive query from 1. In the example, the answer comes out to be $\frac{4}{5}$ which is accurate.

Observe that the selectivity of a predicate is equivalent to the probability of that predicate. This allows probability rules to be applied in many selectivity computations. This is particularly useful when dealing with complex predicates. For example, if we assume that multiple predicates involved in a conjunction are *independent*, we can compute the total selectivity of the conjunction as the product of the selectivities of the individual predicates.

Selectivity Computation Assumptions

In computing the selection cardinality of predicates, the following three assumptions are used.

- **Uniform Data:** The distribution of values (except for the heavy hitters) is the same.
- **Independent Predicates:** The predicates on attributes are independent.
- **Containment Principle:** The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.

These assumptions are often not satisfied by real data. For example, *correlated attributes* break the assumption of independence of predicates.

Join Size Estimation

For a join between relations R and S , the goal is to determine how many tuples in S each tuple in R could match, assuming that every key in the inner relation exists in the outer table. We make the same three assumptions as above when estimating the join size.

In the general case where R and S share a non-primary-key attribute A , the estimated join cardinality is approximately $\frac{N_R \times N_S}{\max(V(A,S), V(A,R))}$.

However, estimation errors can easily propagate through query plans, as inaccuracies in one operator's estimate may cascade and amplify in subsequent operations, leading to poor overall cost estimates and suboptimal query plans.