

# Lecture #14: Query Planning & Optimization II

15-445/645 Database Systems (Fall 2021)

<https://15445.courses.cs.cmu.edu/fall2021/>

Carnegie Mellon University

Lin Ma

## 1 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, 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.

### 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 1). A more complex predicate is shown in Figure 2.

The *selectivity* (sel) of a predicate  $P$  is the fraction of tuples that qualify. The formula used to compute selective 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 3.

```
SELECT * FROM people
WHERE id = 123
```

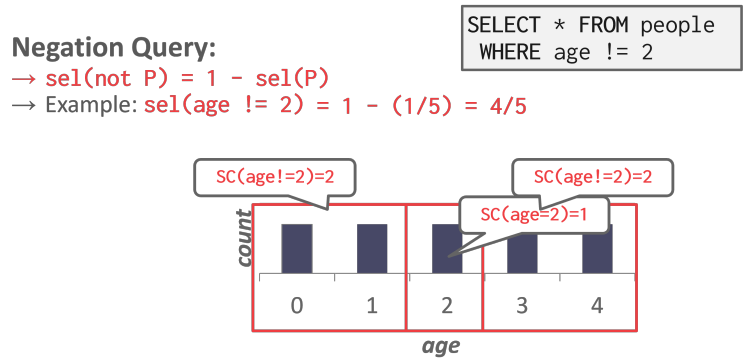
```
CREATE TABLE people (
  id INT PRIMARY KEY,
  val INT NOT NULL,
  age INT NOT NULL,
  status VARCHAR(16)
);
```

**Figure 1: 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 2: 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.



**Figure 3: 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**

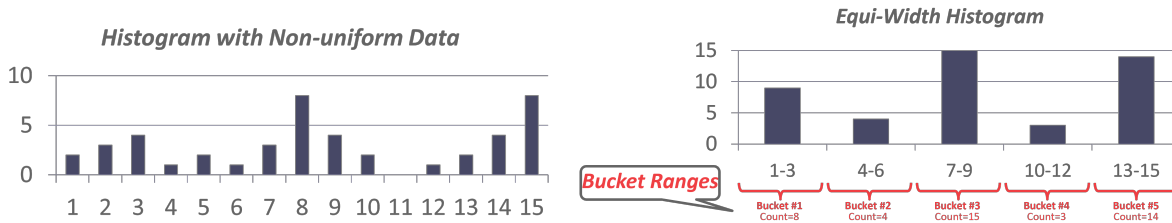
In computing the selection cardinality of predicates, the following three assumptions are used.

- **Uniform Data:** The distribution of values (except for the heavy hitters) is the same.
- **Independent Predicates:** The predicates on attributes are independent.
- **Inclusion 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.

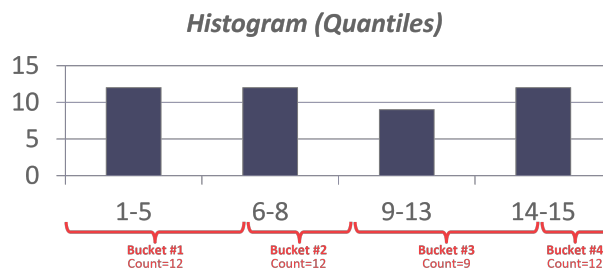
## 2 Histograms

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 by storing data in a *histogram* to group together values. An example of a graph with buckets is shown in Figure 4.



**Figure 4: 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.

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 5.

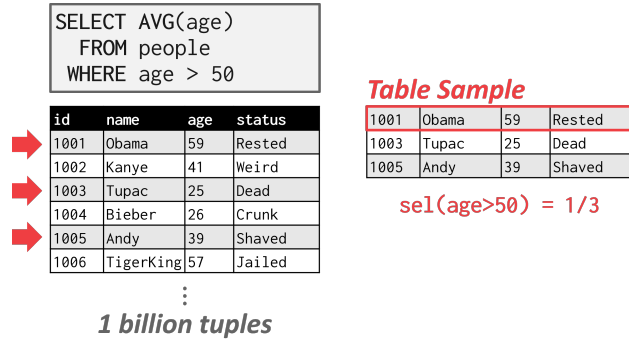


**Figure 5: Equi-Depth Histogram** – To ensure that each bucket has roughly the same number of counts, the histogram varies the range of each bucket.

In place of histograms, some systems may use *sketches* to generate approximate statistics about a data set.

## 3 Sampling

DBMS's can use *sampling* to apply predicates to a smaller copy of the table with a similar distribution (see Figure 6). The DBMS updates the sample whenever the amount of changes to the underlying table exceeds some threshold (e.g., 10% of the tuples).



**Figure 6: 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.

## 4 Plan Enumeration

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.

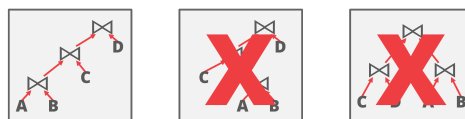
### Single-Relation Query Plans

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.

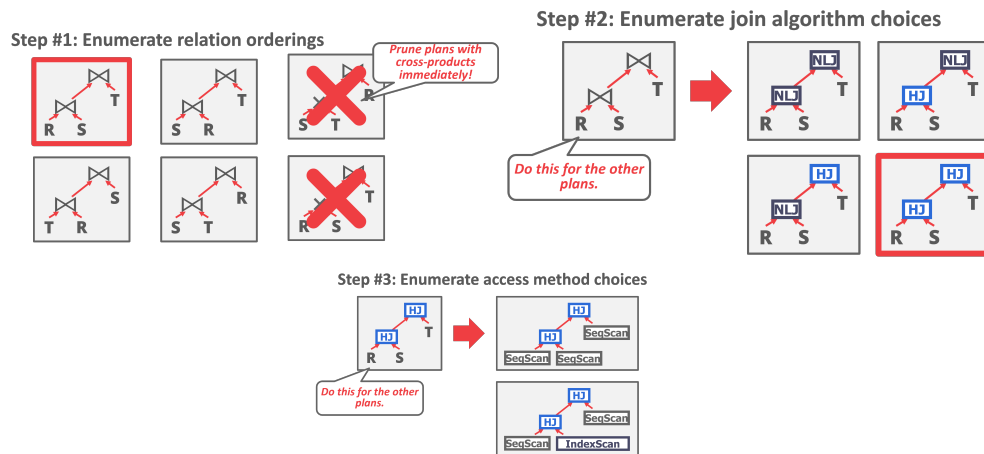
### Multi-Relation Query Plans

As the number of joins increases, the number of alternative plans grows rapidly. To deal with this, we need to restrict the search space. **IBM System R** made the fundamental decision to only consider left-deep join trees (see Figure 7). This is because left-deep join trees are better suited for the pipeline model since the the DBMS does not need to materialize the outputs of the join operators. If the DBMS’s optimizer only considers left-deep trees, then it will reduce the amount of memory that the search processes uses and potentially reduce the search time. Most modern DBMSs do not make this restriction during optimization.



**Figure 7: System R Optimizer** – The first cost-based query optimizer in **IBM System R** only considered left-deep join trees.

To make query plans, the DBMS must first enumerate the orderings, then the plans for each operator, followed by the access paths for each table. See Figure 8 for an example. *Dynamic programming* can be used to reduce the number of cost estimations.



**Figure 8: Candidate Plans Example** – The first step is to enumerate all relation orderings. Any orderings with cross-products or non-left deep joins can be pruned. In the second step, all join algorithm choices (e.g. nested loop join, hash join, sort-merge join) are enumerated. In step three, the access methods are enumerated to find the cheapest path.