Lecture #2: The System R Optimizer

15-721 Advanced Database Systems (Fall 2024)

https://www.cs.cmu.edu/~15721-f24/

Carnegie Mellon University

Note authored by: Suhas Thalanki

1 Introduction

Database systems have continually evolved in terms of how data is stored and retrieved. SQL (Structured Query Language) was one of the first languages developed to express queries for data retrieval.

System R was a Database Management System (DBMS) built at IBM and it utilized SQL for its operations. Created primarily for research purposes, System R played a critical role in creating a query planner and optimizing the query plan, which was the first of its kind.

SQL is a non-procedural/declarative, high-level query language in which users specify the data to be obtained but not how to obtain data. System R determines both the access path to the tuples and the order of the joins in case of multi-table queries. The optimizer determines the "total access cost" of all the possible choices and chooses one with the least cost.

This results in a few advantages:

1.1 Data Independence

Data independence refers to the separation of user applications from the underlying data representations. Previously, this relationship was tightly coupled, but with data independence, the database schema is divided into logical schema, physical schema, and the actual storage of data on disk. This separation allows for complete independence between these different layers.

With this capability, users can focus solely on the application logic, while the Database Management System (DBMS) gains the flexibility and responsibility to optimize data layout and adjust it as workloads evolve. Additionally, this approach enables the creation of an *external schema* through the use of "views," introducing an additional layer of abstraction that defines what data users can access and see.

1.2 Multi-Relation Query Planning

In the past, programmers played the role of optimizers, tasked with crafting a physical execution plan, which involved making decisions about which files to open, which indices to utilize, and other low-level details. However, modern DBMSs now receive a declarative, high-level specification of *what* data to retrieve, rather than *how* to retrieve it. This shift reduces the technical expertise required to interact with databases, thereby lowering the barrier to entry and facilitating more widespread adoption.

2 Stages of processing SQL Statement

There are 4 stages of processing that a SQL Statement undergoes (especially the System R).

2.1 Parsing

The parser processes the SQL query and checks for correct syntax. The **query block** includes the SELECT list, FROM list, and WHERE clause fields. The parser verifies the validity of the provided elements, such as table names, columns, and commands. Then it returns a result indicating either success or failure based on the correctness of these elements.

2.2 Rewrite/Unnest

This process involves transforming complex SQL queries into simpler, equivalent forms. It involves rewriting into more efficient joins or other constructs and flattening nested sub-queries, particularly correlated sub-queries, into joins or other operations that can be more efficiently processed by the database engine.

2.3 Optimize Query

The optimizer is invoked after the preceding stages have been completed. Query optimization involves the following actions:

- Accumulates the names of tables and columns referenced in the query, along with statistics about the referenced relations (which will be discussed shortly) and the available access paths for each. These details are crucial for the subsequent step of access path selection.
- The optimizer evaluates all possible access paths and selects the optimal one. There are a few ways to do this:
  - **Bottom-Up Approach**: The optimizer evaluates all possible access paths and selects the plan with the lowest total cost. This is how the **System R** optimizer selects an access path.
  - **Top-Down Approach**: The optimizer begins with the desired outcome and then works its way down the tree to identify the optimal plan to achieve that goal. This results in having a working plan without enumerating the entire search space, at the cost of not having a globally optimal plan.

A notable advantage of the top-down approach is that it can time out after a specified duration (e.g., 5 minutes) and still provide the best plan found so far. In contrast, the bottom-up approach requires processing all possible plans before delivering its final result, as it enumerates all potential plans exhaustively.

These plans can have different structures. Examples of different plan structures include:

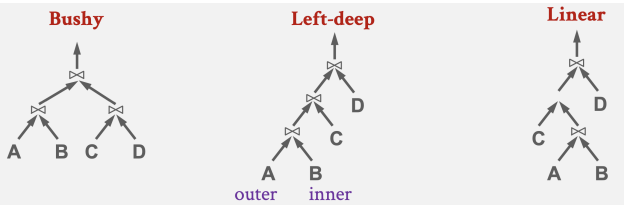


Figure 1: Different Plan Structures.

- Bushy Plan
- **Left Deep Plan** - System R uses this approach. This version builds an efficient pipeline.
- Linear Plan

2.4 Execution

Query is executed using the access path generated by the OPTIMIZER. Once it's generated, it calls the System R *Research Storage System(RSS)*, via *Internal Storage Interface(RSI)*.

3 Optimizer Overview

3.1 Search Space

System R restricts its Search Space to just left-deep plans. This results in a reduction of the search space from $n!$ to $n2^{n-1}$.

3.2 Enumeration Strategy

System R employs an incremental algorithm to explore the search space. The process involves the following steps:

1. **Find the best 1-relation plan for each relation**: This step identifies the optimal way to access a single table, which may involve the use of indices.
2. **Join the best 1-relation plan with another relation**: The algorithm then finds the best way to join the selected 1-relation plan as the outer relation to another relation.
3. **N-pass strategy**: For each subsequent pass, the algorithm determines the best way to join the result of an $n-1$ relation plan as the outer relation to the n^{th} relation.

An interesting observation is that it does not need to remember anything at a previous level explicitly as it's being remembered implicitly by the nature of a Bottom-Up Approach.

For each subset of relations, retain (for each column of interest):

- Cheapest plan overall to join that subset
- Cheapest plan for each *interesting order* of the tuples

where an *interesting order* is a tuple order specified by columns/attributes that benefit from having an intermediate result sorted by that column. Thus columns that are part of an join predicate, or appear in a GROUP BY clause, or appear in the (prefix of) an ORDER BY clause are considered "interesting."

If an interesting order is found to be cheaper than an uninteresting one, the latter is discarded. The algorithm then considers all combinations of single-access plans and selects the combination with the lowest cost, taking into account every sorting algorithm for each combination.

3.3 Cost Estimation

The optimizer needs to estimate the cost of each enumerated plan to determine the lowest cost plan. Each estimate needs to be:

- **Accurate**: Providing precise cost assessments.
- **Fast**: Ensuring quick computation.
- **Space-Efficient**: Minimizing the storage required for any summary structures, such as histograms.

The cost is calculated as: $Cost = W \times CPU\,Cost + I/O\,Cost$

where W is a magic constant that weights the CPU Cost. The CPU cost is determined by the number of tuples accessed in the storage layer, while the I/O cost is more complex and involves several factors. System R tracks various statistics for each table and index to compute these costs:

- **For Tables**: It records the number of tuples, pages, and non-empty pages in a segment.
- **For Indexes**: It tracks the number of distinct keys, the number of pages, and the range defined by the high-key and low-key.

To calculate the selectivity of simple predicates, we can proceed as follows:

- **colA = value:**
  1. If the index exists,
  $$\frac{1}{number\,of\,distinct\,keys\,in\,the\,index} \tag{1}$$
  2. Else
  $$\frac{1}{10} \tag{2}$$
  which is a magic number.
- **colA > value:**
  1. If the index exists,
  $$\frac{high\,key - value}{high\,key - low\,key} \tag{3}$$

To calculate the selectivity of complex predicates, we can proceed as follows:

- **Conjunctions: p1 and p2**:

$$Estimate(p1) \times Estimate(p2) \tag{4}$$

  - An example of a conjunction is the query:

$$salary > 100K \, and \, age < 30 \tag{5}$$

  - Equation 4 makes the assumption that p1 and p2 are independent.
- **Disjunct: p1 or p2**:

$$Estimate(p1) + Estimate(p2) - Estimate(p1) \times Estimate(p2) \tag{6}$$

- **Negation: not p1**:

$$1 - Estimate(p1) \tag{7}$$

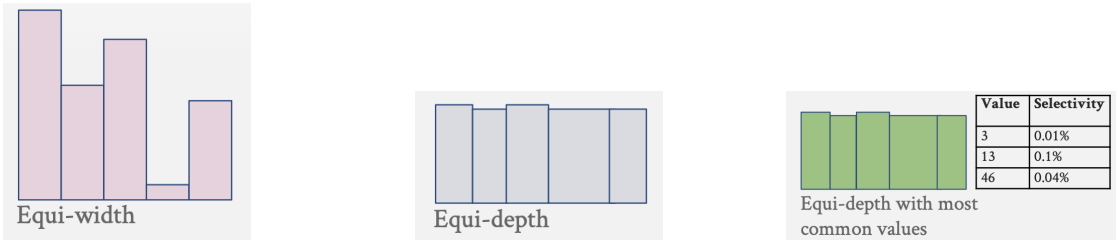
4 Modern Cardinality Estimation Methods

Scanning the entire dataset or table is resource-intensive, so instead, a sample of the data is collected, and histograms are constructed based on this sample.

4.1 Histograms

To estimate the distribution of values, we can construct **Histograms** which is a special type of column statistic that provides more detailed information about the data distribution in a table column. In a Histogram, we divide data into discrete "bins". For each bin, we store the boundaries and the count of values within those boundaries. This is used instead of relying on uniform estimation similar to Equation 3.

The different types of Histograms are:



(a) Equi-Width Histogram (b) Equi-Depth Histogram (c) Equi-Depth Histogram with MCV

Figure 2: Different kinds of Histograms.

- **Equi-Width**: Divide value space into bins of Equal Widths.
- **Equi-Depth**: Adjust the width of the bins to maintain the count of each bin to be approximately equal.

The Equi-Depth histogram, combined with a Most Common Value (MCV) adjustment, is typically the preferred method. This approach mitigates the influence of *heavy hitters*—values that disproportionately skew the data distribution. By storing these heavy hitters separately, the average estimates remain accurate. When a bin containing a heavy hitter is accessed, its value is incorporated into the estimate.

Given the large volume of data in modern DBMSs, generating histograms using all available tuples is impractical. Instead, a subset of tuples is sampled. A practical guideline is to sample 300 times the number of histogram bins. While increasing the sample size can lead to more refined results, such extensive sampling is usually unnecessary.

4.1.1 Background Statistic Recalculation

As the contents of a DBMS change over time, many systems have a background job that periodically recalculates statistics. This job runs at the lowest level of concurrency (level 0) with no locks, as it is read-only. The process is designed to be robust against occasional inaccuracies, as they tend to average out over time. A more cautious approach involves obtaining a read lock to ensure the page isn't modified during reading.

4.2 Sketches

A probabilistic data structure that estimates the cardinality of a given set, which is crucial in scenarios where searching through all tuples can be time-consuming, particularly with large datasets. These structures optimize search efficiency by minimizing the need for exhaustive lookups.

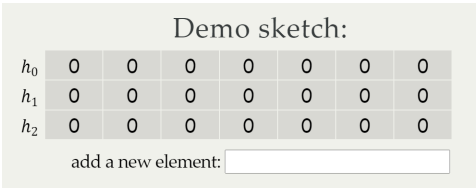
As a probabilistic data structure, it may not provide an exact count, but its estimates are accurate enough to be useful for query optimization statistics.

The structure consists of a 2D array (or table) of 'n' counters, where each row is associated with a different hash function.

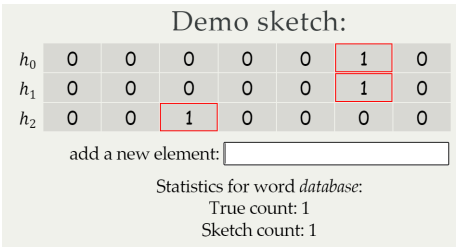
It works in following manner (assume n=3):

- When an element is added, it is hashed to three different values, and the corresponding indices in the data structure are each incremented by 1.
- Since each hash function generates a different hash value, it's likely that the same index won't be updated across all rows.
- To check the count of a given element, it is hashed, and the values at the corresponding indices are retrieved. The minimum of these values is estimated to be the frequency of that element in the table.

Here is a step wise example:



Step 1: Intialized with '0'.



Step 2: Hashing values after inserting the word "Database".

Note: Pessimistic Cardinality Estimation – a new class of estimation methods – guarantees no under-estimation in the predicted cardinalities; however, these methods are theoretical at the moment and not practical.

Demo sketch:

h_0	1	0	1	1	1	2	1
h_1	1	1	1	1	0	3	0
h_2	4	1	2	0	0	0	0

add a new element:

Statistics for word *database*:
True count: 2
Sketch count: 2

Step 3: Again inserting "Database" and obtaining count as "2" by min(2,3,2).