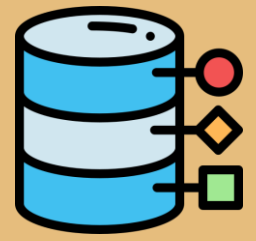


CSAI 302

Advanced Database Systems

Lec 06

Query Processing & Optimization [2]

Cost-based Search

- Use a model to estimate the cost of executing a plan.
- Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

Cost-based query optimization

- Start with cost-based, bottom-up QO
- Enumerate different plans for the query and estimate their costs.
- It chooses the best plan it has seen for the query after exhausting all plans or some timeout.
  - Single relation.
  - Multiple relations.
  - Nested sub-queries.

Single-relation query planning

- Pick the best access method.
  - Sequential Scan
  - Binary Search
  - Index Scan
- Predicate evaluation ordering.
- Simple heuristics are often good enough for this.

Multi-relation query planning

- Generative / Bottom-Up
  - Start with nothing and then iteratively assemble and add building blocks to generate a query plan.
  - Examples: System R, Starburst
- Transformation / Top-Down
  - Start with the outcome that the query wants, and then transform it to equivalent alternative sub-plans to find the optimal plan that gets to that goal.
  - Examples: Volcano, Cascades

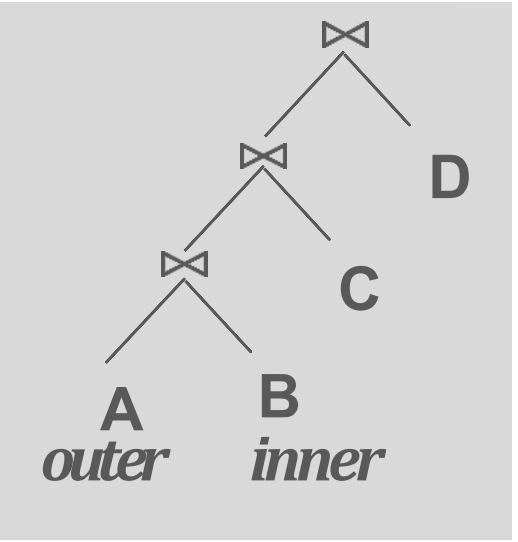
Bottom-up optimization

- 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
- Examples
  - IBM System R, DB2, MySQL, Postgres, most open-source DBMSs.

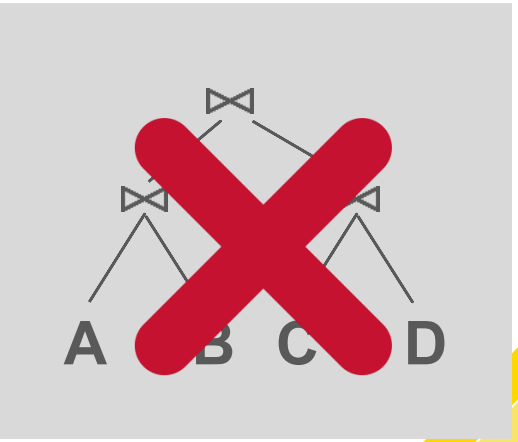
System R optimizer

- Break query into blocks and generate logical operators for each block.
- For each logical operator, generate a set of physical operators that implement it.
  - ★ All combinations of join algorithms and access paths
- Then, iteratively construct a "left-deep" join tree that minimizes the estimated amount of work to execute the plan.

Left-Deep Tree



Bushy Tree



System R optimizer

```
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
```

ARTIST: Sequential Scan

APPEARS: Sequential Scan

ALBUM: Index Look-up on **NAME**

Step #1: Choose the best access paths to each table

Step #2: Enumerate all possible join orderings for tables

Step #3: Determine the join ordering with the lowest cost

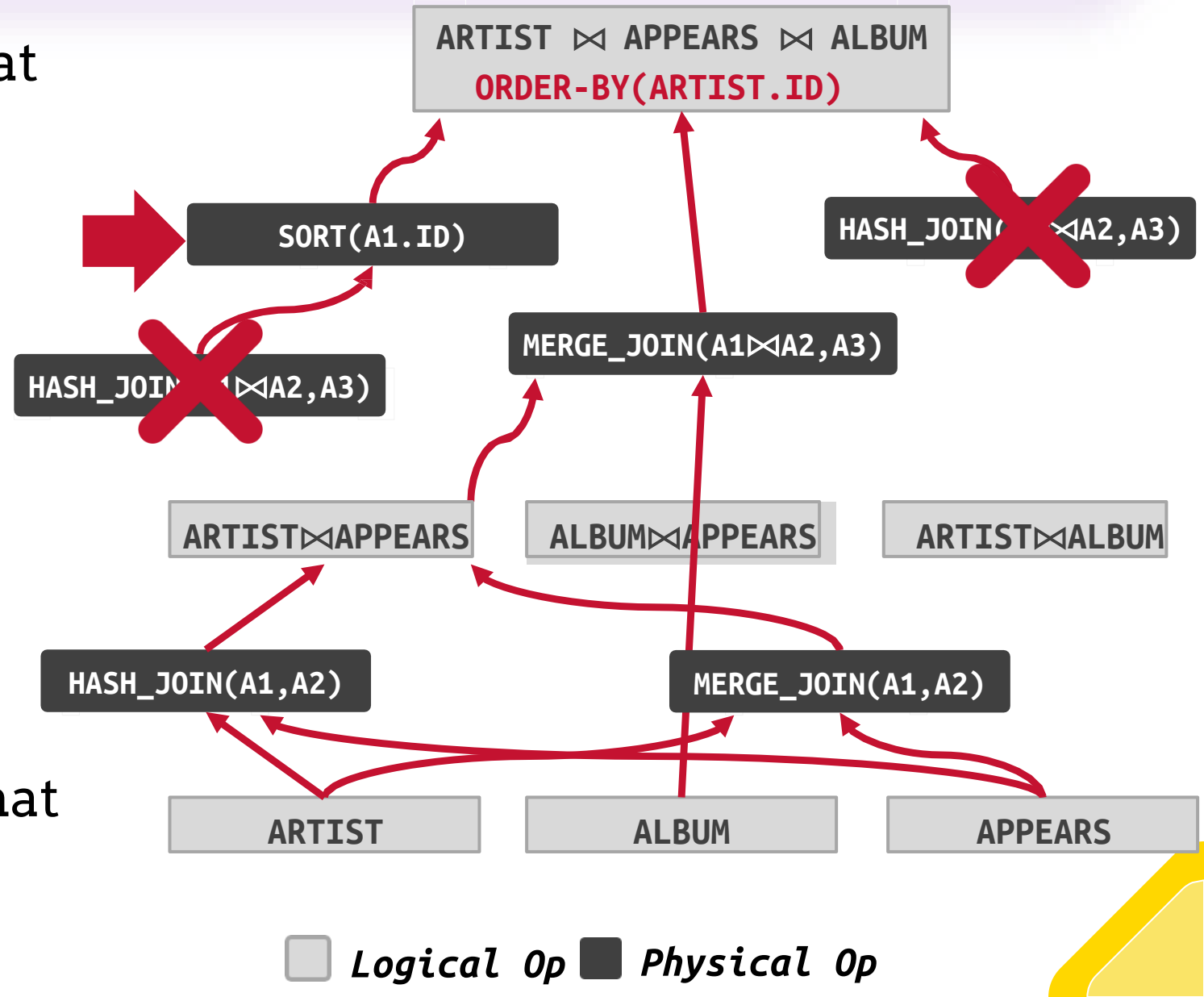
$$
\begin{array}{lll}
\text{ARTIST} & \bowtie & \text{APPEARS} \bowtie \text{ALBUM} \\
\text{APPEARS} & \bowtie & \text{ALBUM} \bowtie \text{ARTIST} \\
\text{ALBUM} & \bowtie & \text{APPEARS} \bowtie \text{ARTIST} \\
\text{APPEARS} & \bowtie & \text{ARTIST} \bowtie \text{ALBUM} \\
\text{ARTIST} & \times & \text{ALBUM} \bowtie \text{APPEARS} \\
\text{ALBUM} & \times & \text{ARTIST} \bowtie \text{APPEARS} \\
\vdots & & \vdots \qquad\qquad \vdots
\end{array}
$$

Top-down optimization

- Start with a logical plan of what we want the query to be.
- Perform a branch-and-bound search to traverse the plan tree by converting logical operators into physical operators.
  - Keep track of global best plan during search.
  - Treat physical properties of data as first-class entities during planning.
- Examples: MSSQL, Greenplum, CockroachDB

Top-down optimization

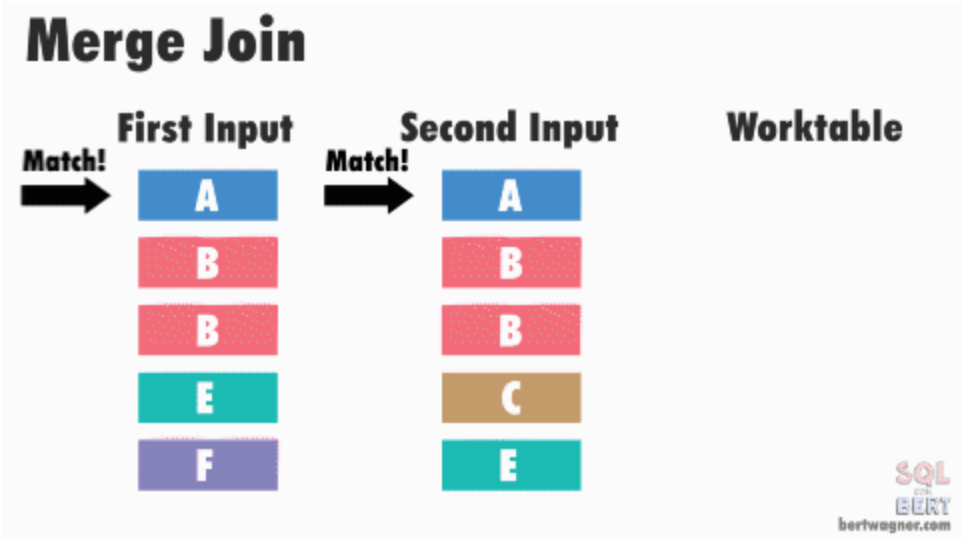
- Start with a logical plan of what we want the query to be.
- Invoke rules to create new nodes and traverse tree.
  - Logical→Logical:
  - JOIN(A,B) to JOIN(B,A)
  - Logical→Physical:
  - JOIN(A,B) to HASH_JOIN(A,B)
- Can create "enforcer" rules that require input to have certain properties.



Hash join vs. Merge join

Feature	Hash Join	Merge Join
How it works	Builds a hash table from one table's join column and then probes the hash table with rows from the other table.	Sorts both tables on the join key and then merges them together by comparing rows in a single pass.
When it's best	Large, unsorted tables, especially when one fits into memory.	Tables that are already sorted on the join key, or when the join is a range join.
Memory usage	Can be high, especially if the hash table doesn't fit in memory and spills to disk.	Lower, as it primarily requires memory for sorting (if needed) and the merge process itself.
Performance	Fast for large datasets when the hash table fits in memory; performance degrades if it has to use disk.	Slower if tables need to be sorted first; very fast and efficient if data is already sorted.
Join types	Primarily used for equi-joins (=).	Can handle both equi-joins and non-equi-joins.

Merge Join

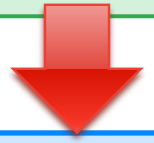

Merge Join
First Input
Second Input
Worktable
Match!
Match!
A
B
B
E
F
A
B
B
C
E
SQL
BERT
bertwagner.com

Nested sub-queries

- The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.
  1. Rewrite to de-correlate and/or flatten them.
  2. Decompose nested query and store results in a temporary table.

Nested sub-queries: Rewrite

```
  SELECT name FROM sailors AS S
WHERE EXISTS (
    SELECT * FROM reserves AS R
        WHERE S.sid = R.sid
       AND R.day = '2024-10-25'
)
```



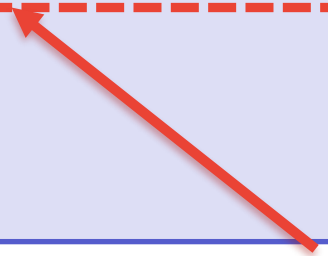
```
SELECT name
  FROM sailors AS S, reserves AS R
 WHERE S.sid = R.sid
   AND R.day = '2024-10-25'
```

Decomposing queries

- For harder queries, the optimizer breaks up queries into blocks and then concentrates on one block at a time.
- Sub-queries are written to temporary tables that are discarded after the query finishes.

Decomposing queries

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid  AND R.bid
   = B.bid  AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                     FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```



Nested Block

Decomposing queries

Inner Block

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid  AND R.bid
   = B.bid  AND B.color = 'red'
   AND S.rating =

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Outer Block

Expression rewriting

- An optimizer transforms a query's expressions (e.g., **WHERE/ON** clause predicates) into the minimal set of expressions.
- Implemented using if/then/else clauses or a pattern-matching rule engine.
  - Search for expressions that match a pattern.
  - When a match is found, rewrite the expression.
  - Halt if there are no more rules that match.

Expression rewriting

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0;
```

→

```
SELECT * FROM A WHERE false;
```

```
SELECT * FROM A WHERE NOW() IS NULL;
```

```
SELECT * FROM A WHERE false;
```

Merging Predicates

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 100
    OR val BETWEEN 50 AND 150;
```

→

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 150;
```

Cost estimation

Cost estimation

- The DBMS uses a cost model to predict the behavior of a query plan given a database state.
  - ★ This is an **internal** cost that allows the DBMS to compare one plan with another.
- It is too expensive to run every possible plan to determine this information, so the DBMS need a way to derive this information.

Cost model components

- Physical Costs
  - Predict CPU cycles, I/O, cache misses, RAM consumption, network messages…
  - Depends heavily on **hardware**.
- Logical Costs
  - Estimate output size per operator.
  - Independent of the operator algorithm.
  - Need estimations for operator result sizes.

Statistics

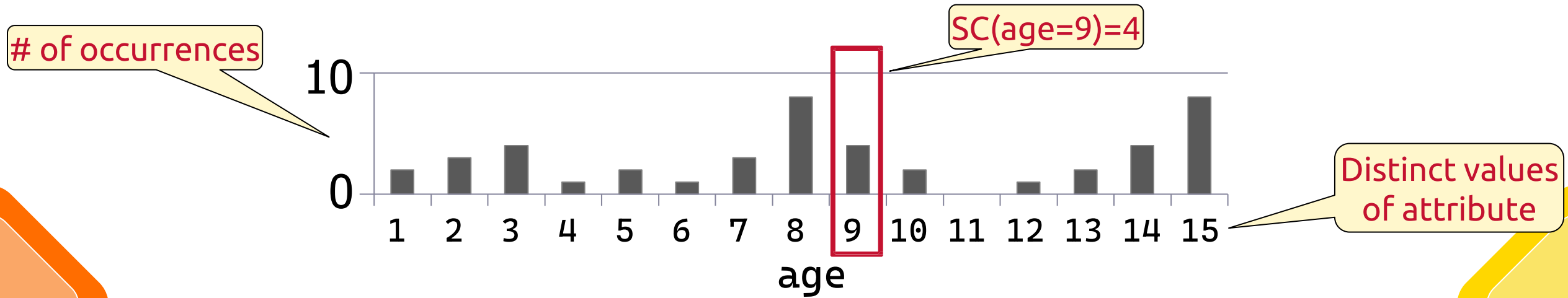
- The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.
- Different systems update them at different times.
- Manual invocations:
  - Postgres/SQLite: ANALYZE
  - Oracle/MySQL: ANALYZE TABLE
  - SQL Server: UPDATE STATISTICS
  - DB2: RUNSTATS

Selection cardinality

- The selectivity (**sel**) of a predicate **P** is the fraction of tuples that qualify:
  - Equality Predicate: **A=constant**
  - **sel(A=constant) = #occurences / |R|**

```
SELECT * FROM people
 WHERE age = 9
```

Example: **sel(age = 9) = 4/45**



Selection cardinality

- Uniform Data
  - The distribution of values 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.

Statistics

Histograms

- Maintain an occurrence count per value (or range of values) in a column.

Sketches

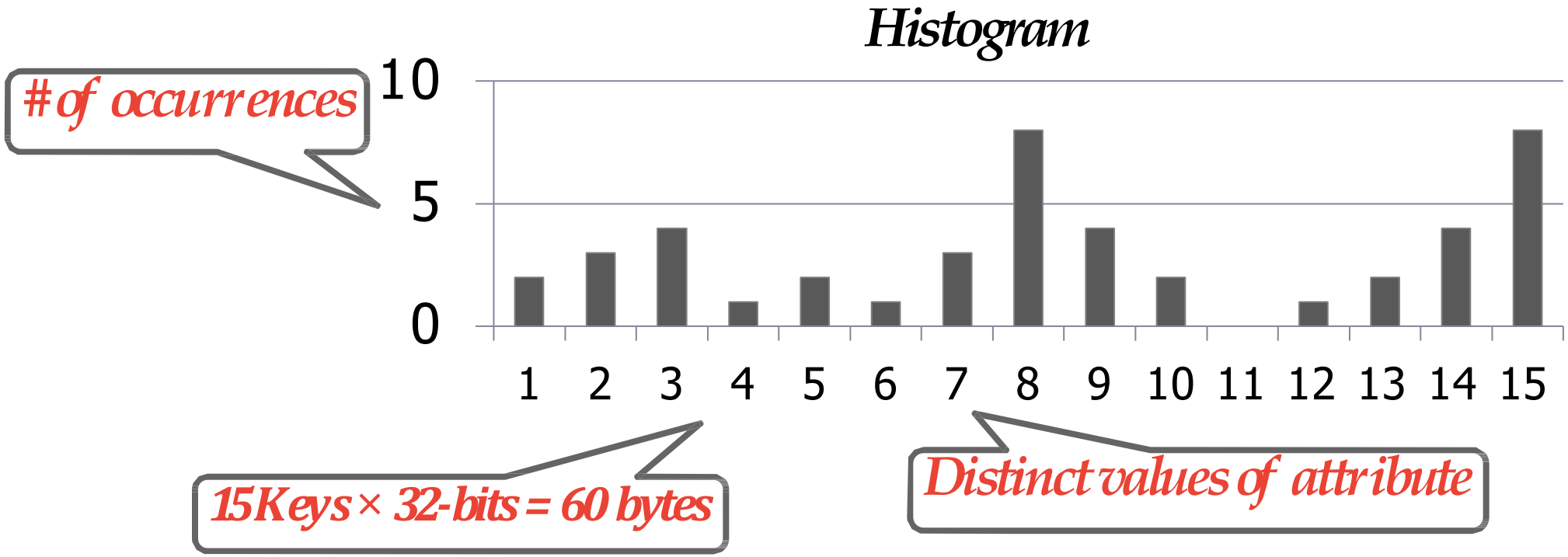
- Probabilistic data structure that gives an approximate count for a given value.

Sampling

- DBMS maintains a small subset of each table that it then uses to evaluate expressions to compute selectivity.

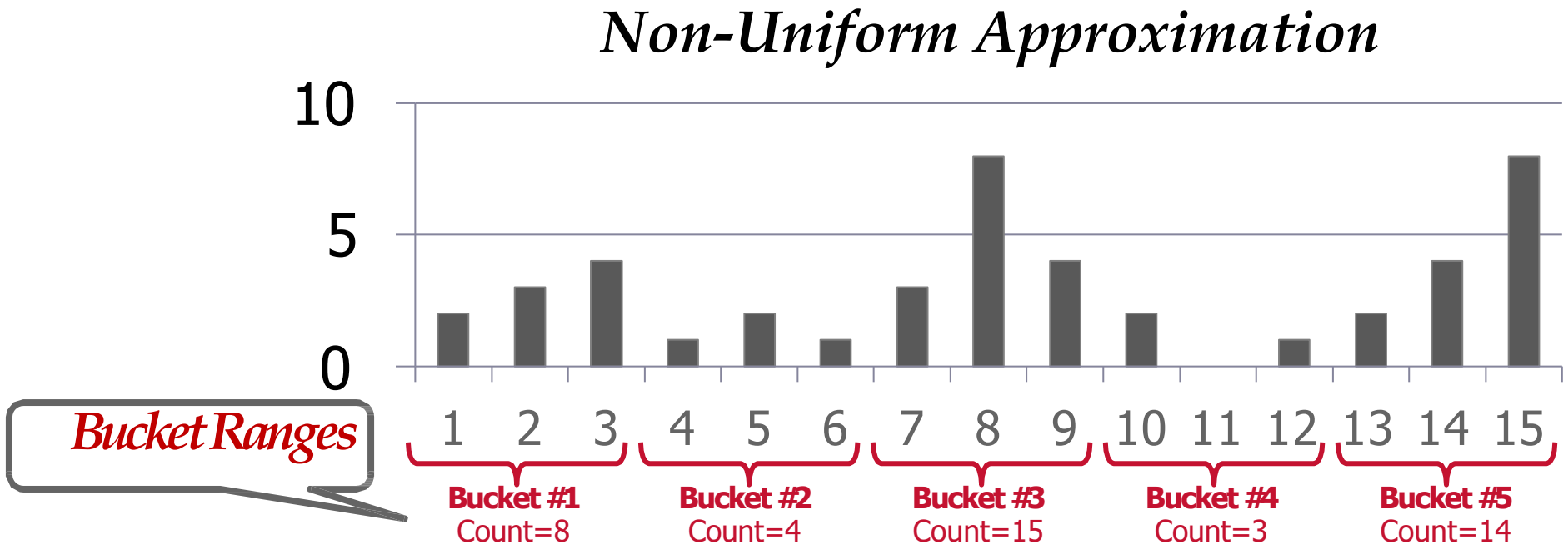
Histograms

Our formulas are nice, but we assume that data values are uniformly distributed.



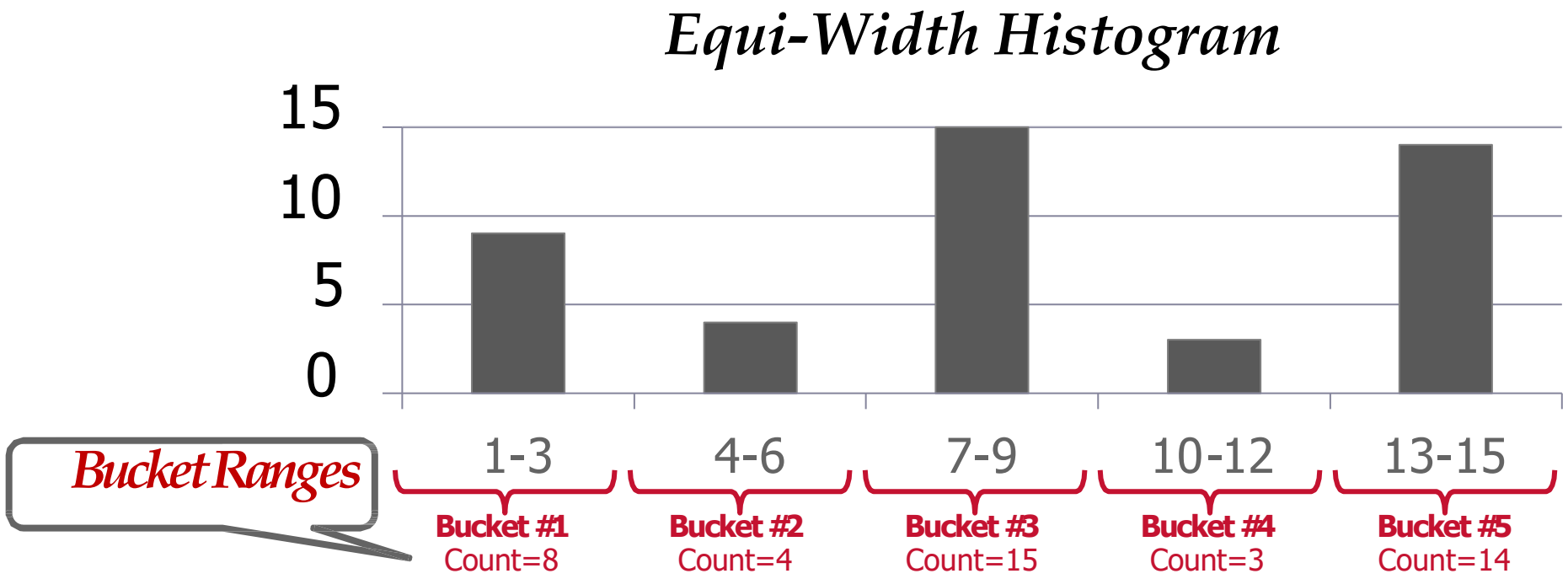
Equi-width histogram

Maintain counts for a group of values instead of each unique key. All buckets have the same width (i.e., same # of value).



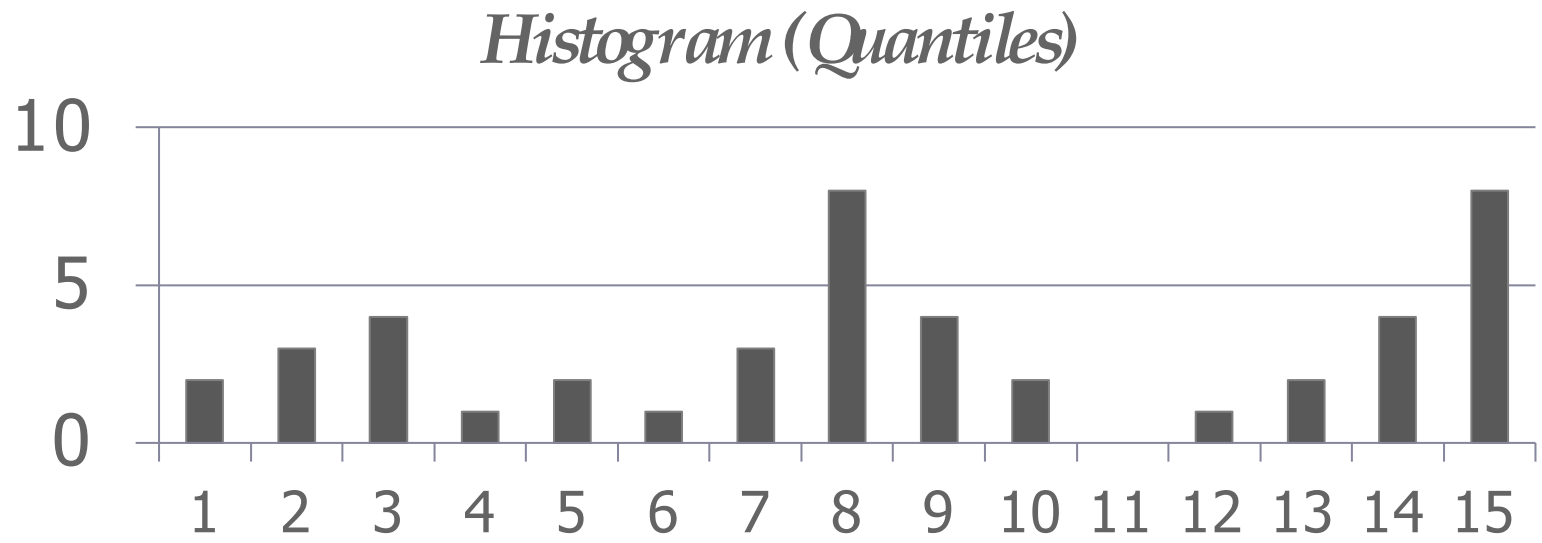
Equi-width histogram

Maintain counts for a group of values instead of each unique key. All buckets have the same width (i.e., same # of value).



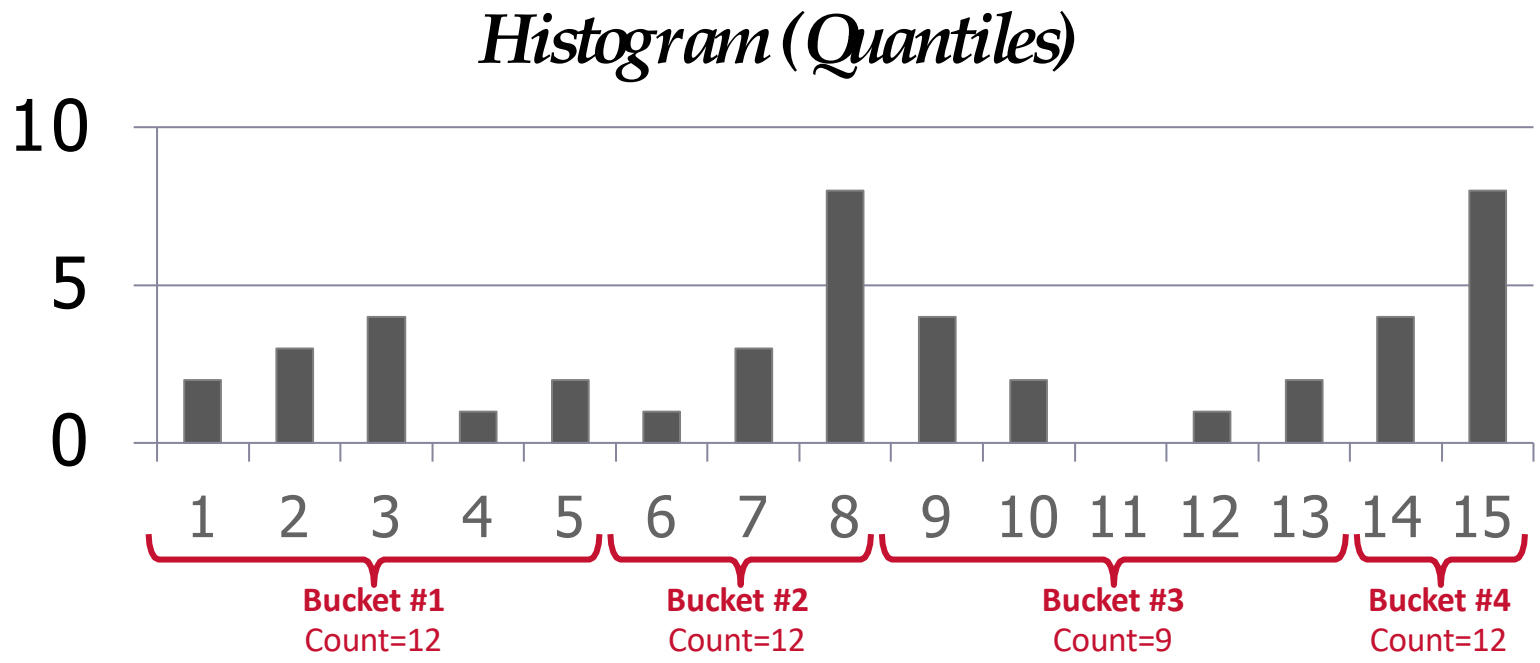
EQUI-DEPTH HISTOGRAMS

Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.



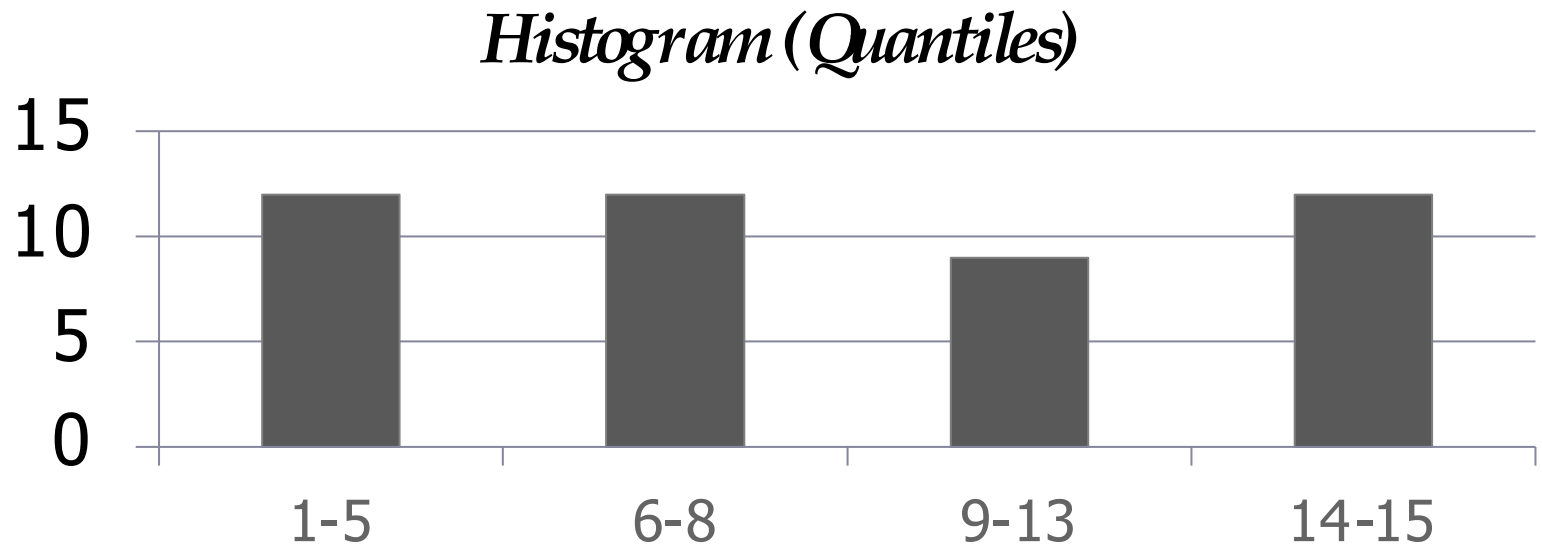
Equi-depth histograms

Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.



Equi-depth histograms

Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.



Sketches

- Probabilistic data structures that generate approximate statistics about a data set.
- Cost-model can replace histograms with sketches to improve its selectivity estimate accuracy.
- Most common examples:
  - Count-Min Sketch
    - Approximate frequency count of elements in a set.
  - HyperLogLog
    - Approximate the number of distinct elements in a set.

Sampling

- Modern DBMSs also collect samples from tables to estimate selectivities.
- Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	63	Rested
1002	Swift	34	Paid
1003	Tupac	25	Dead
1004	Bieber	30	Crunk
1005	Andy	43	Illin
1006	TigerKing	61	Jailed

⋮

1 billion tuples

Table Sample

1001	Obama	63	Rested
1003	Tupac	25	Dead
1005	Andy	43	Illin

sel(age>50) = 1/3

Adaptive Query Optimization

Adaptive Query Optimization

- The best plan for a query can change as ***the database evolves over time***.
  - Physical design changes.
  - Data modifications.
  - Prepared statement parameters.
  - Statistics updates.
- The query optimizers that we have talked about so far all generate a plan for a query **before** the DBMS executes a query.

Bad query plans

- The most common problem in a query plan is incorrect join orderings.
  - ★ This occurs because of inaccurate cardinality estimations that propagate up the plan.
- If the DBMS can detect how bad a query plan is, then it can decide to **adapt** the plan rather than continuing with the current sub-optimal plan.

Why good plans go bad?

- Estimating the execution behavior of a plan to determine its quality relative to other plans.
- These estimations are based on a **static** summarizations of the contents of the database and its operating environment:
  - Statistical Models / Histograms / Sampling
  - Hardware Performance
  - Concurrent Operations

Optimization timing

- Static Optimization
  - Select the best plan prior to execution.
  - Plan quality is dependent on cost model accuracy.
  - Can amortize over executions with prepared statements.
- Dynamic Optimization
  - Select operator plans on-the-fly as queries execute.
  - Will have re-optimize for multiple executions.
  - Difficult to implement/debug (non-deterministic)
- Adaptive Optimization
  - Compile using a static algorithm.
  - If the estimate errors > threshold, change or re-optimize.

Adaptive query optimization

- Modify the execution behavior of a query by generating multiple plans for it:
  - Individual complete plans.
  - Embed multiple sub-plans at materialization points.
- Use information collected during query execution to improve the quality of these plans.
  - Can use this data for planning one query or merge the it back into the DBMS's statistics catalog.

Adaptive query optimization

- Modify Future Invocations
- Replan Current Invocation
- Plan Pivot Points

Modify future invocations

- The DBMS monitors the behavior of a query during execution and uses this information to improve subsequent invocations.
  - **Plan Correction**
  - **Feedback Loop**

Reversion-based plan correction

- The DBMS tracks the history of query invocations:
  - Cost Estimations
  - Query Plan
  - Runtime Metrics
- If the DBMS generates a new plan for a query, then check whether that plan performs worse than the previous plan.
  - If it regresses, then switch back to the cheaper plans.

Replan current invocation

- If the DBMS determines that the observed execution behavior of a plan is far from its estimated behavior, them it can halt execution and generate a new plan for the query.
  - **Start-Over from Scratch**
  - **Keep Intermediate Results**

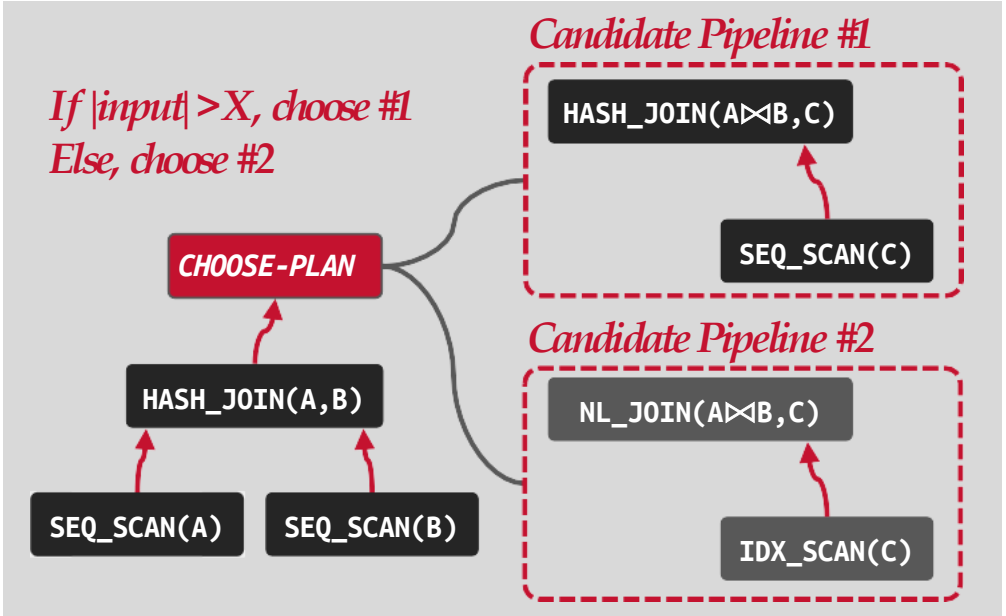
Plan pivot points

- The optimizer embeds alternative sub-plans at materialization points in the query plan.
- The plan includes "pivot" points that guides the DBMS towards a path in the plan based on the observed statistics.
  - **Parametric Optimization**
  - **Proactive Reoptimization**

Parametric optimization

- Generate multiple sub-plans per pipeline in the query.
- Add a choose-plan operator that allows the DBMS to select which plan to execute at runtime.

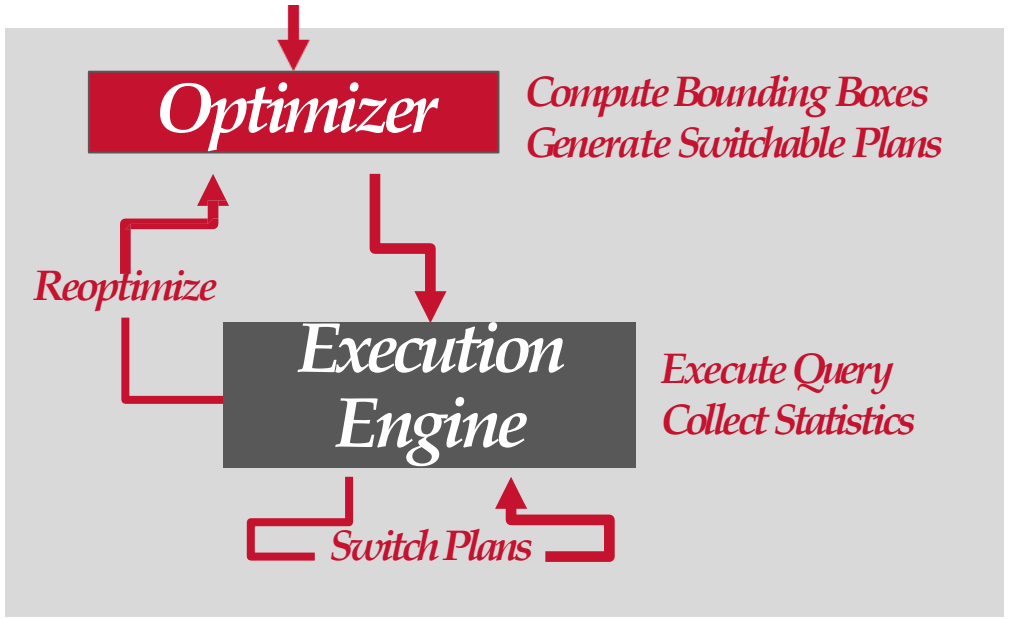
```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id;
```



Proactive reoptimization

- Generate multiple sub-plans within a single pipeline.
- Use a **switch** operator to choose between different sub-plans during execution in the pipeline.
- Computes bounding boxes to indicate the uncertainty of estimates used in plan.

```
SELECT * FROM A
  JOIN B ON A.id = B.id
  JOIN C ON A.id = C.id;
```



Plan stability

- Hints
  - ★Allow the DBA to provide hints to the optimizer.
- Fixed Optimizer Versions
  - ★Set the optimizer version number and migrate queries one-by-one to the new optimizer.
- Backwards-Compatible Plans
  - ★Save query plan from old version and provide it to the new DBMS.

Conclusion

- Query optimization is critical for a database system.
  - SQL → Logical Plan → Physical Plan
  - Flatten queries before going to the optimization part.
- Expression handling is also important.
  - Estimate costs using models based on summarizations.
- QO enumeration can be bottom-up or top-down.

THANK YOU