Carnegie Mellon University

Database Systems

Query Planning & Optimization

15-445/645 FALL 2024 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #3 is due Sunday Nov 17th @ 11:59pm
→ **Recitation will be next week**

Homework #4 is due Sunday Nov 3rd @ 11:59pm

UPCOMING DATABASE TALKS

Exon (DB Seminar)
→ Monday Oct 28th @ 4:30pm
→ Zoom

Synnada (DB Seminar)
→ Monday Nov 4th @ 4:30pm
→ Zoom

InfluxDB (DB Seminar)
→ Monday Nov 11th @ 4:30pm
→ Zoom

LAST CLASS

We talked about how to design the DBMS's architecture to execute queries in parallel.

The query plan is comprised of physical operators that specify the algorithm to invoke at each step of the plan.

LAST CLASS

We talked about how to design the DBMS's architecture to execute queries in parallel.

The query plan is comprised of physical operators that specify the algorithm to invoke at each step of the plan.

But how do we go from SQL to a query plan?

MOTIVATION

```
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

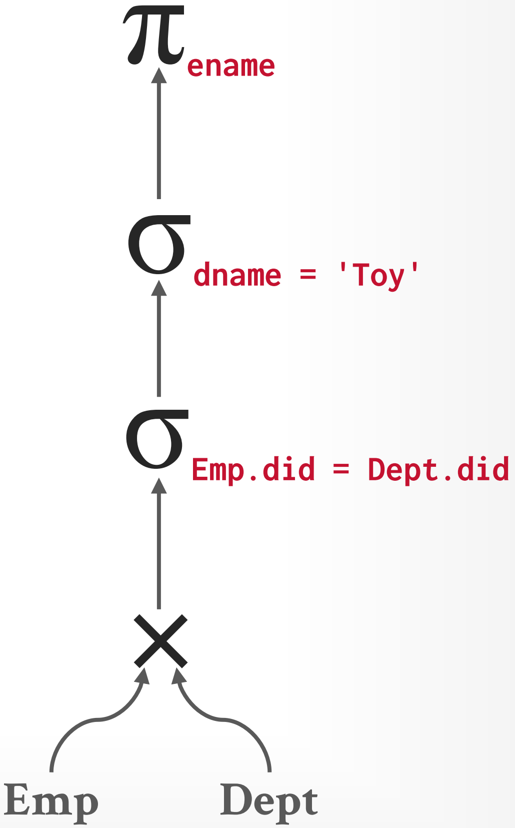
Emp(ssn,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(did,dname,floor,mgr)

500 records
50 pages



MOTIVATION

Total: 2M I/Os

```
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

Emp(ssn,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(did,dname,floor,mgr)

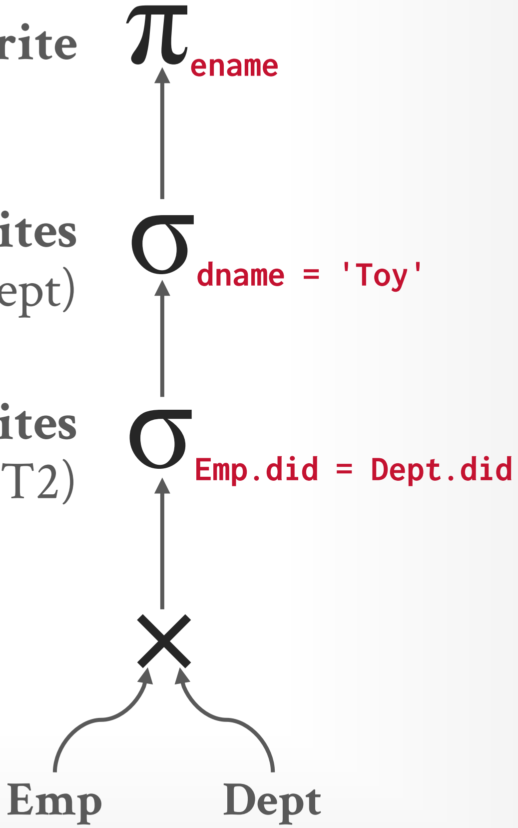
500 records
50 pages

4 reads + 1 write

2,000 reads + 4 writes
(10K/500 = 20 emps per dept)

1,000,000 reads + 2,000 writes
(FK join, 10k tuples in temp T2)

(50 + 50,000) reads
+ 1,000,000 writes
Write temp file T1
5 tuples per page in T1



MOTIVATION

```sql
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

Emp(ssn,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(did,dname,floor,mgr)

500 records
50 pages

Total: 54k I/Os

4 reads + 4 writes
Read temp T2

π ename

2,000 reads + 4 writes
Read temp T1, Write temp T2

σ dname = 'Toy'

(50 + 50,000) reads
+ 2,000 writes
Page Nested-Loop Join
Write Temp T1

$\bowtie$ Emp.did = Dept.did

Emp **Dept**

MOTIVATION

```sql
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

Emp(<u>ssn</u>,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(<u>did</u>,dname,floor,mgr)

500 records
50 pages

Total: 54k I/Os

4 reads + 4 writes
Read temp T2

π_{ename}

2,000 reads + 4 writes
Read temp T1, Write temp T2

$\sigma_{\text{dname = 'Toy'}}$

(50 + 50,000) reads
+ 2,000 writes
Page Nested-Loop Join
Write Temp T1

$\bowtie_{\text{Emp.did = Dept.did}}$

Emp **Dept**

MOTIVATION

```sql
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

Emp(ssn,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(did,dname,floor,mgr)

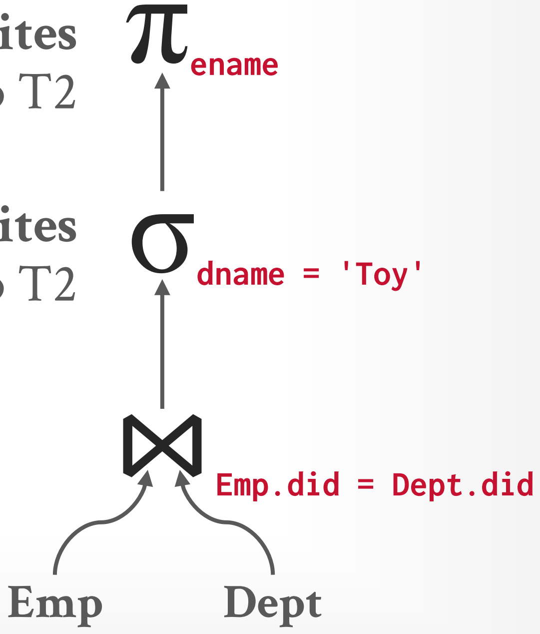
500 records
50 pages

Total: 7,159 I/Os

4 reads + 4 writes
Read temp T2

2,000 reads + 4 writes
Read temp T1, Write temp T2

3×(|Emp| + |Dept|) =
3,150 reads + 2,000 writes
Sort-Merge Join (50 Buffers)
Write Temp T1



MOTIVATION

```
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

No Pipelining!

↳ **Materialization Model** → **Total: 7,159 I/Os**

Catalog

clustered *unclustered* *unclustered*

Emp(ssn,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(did,dname,floor,mgr)

500 records
50 pages

4 reads + 4 writes
Read temp T2

π_{ename}

2,000 reads + 4 writes
Read temp T1, Write temp T2

$\sigma_{\text{dname = 'Toy'}}$

3×(|Emp| + |Dept|) =
3,150 reads + 2,000 writes
Sort-Merge Join (50 Buffers)
Write Temp T1

$\bowtie_{\text{Emp.did = Dept.did}}$

Emp **Dept**

MOTIVATION

```
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

Emp(ssn,ename,addr,sal,did)

10,000 records

1,000 pages

clustered *unclustered*

Dept(did,dname,floor,mgr)

500 records

50 pages

Vectorization Model → Total: 3,151 I/Os

No Pipelining!

Materialization Model → Total: 7,159 I/Os

~~4~~ reads + 4 writes
Read temp T2

π_{ename}

2,~~0~~00 reads + ~~4~~ writes
Read temp T1, Write temp T2

$\sigma_{dname = 'Toy'}$

3×(|Emp| + |Dept|) =
3,150 reads + 2,~~0~~00 writes
Sort-Merge Join (50 Buffers)
Write Temp T1

$\bowtie_{Emp.did = Dept.did}$

Emp **Dept**

MOTIVATION

```
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

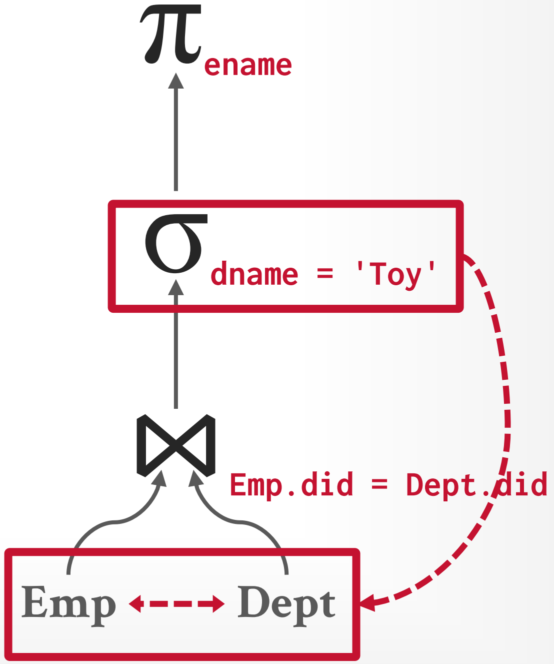
Emp(ssn,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(did,dname,floor,mgr)

500 records
50 pages



MOTIVATION

```
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

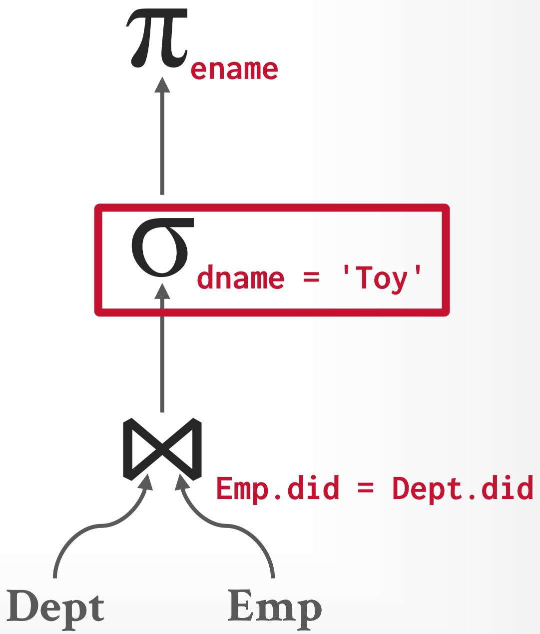
Emp(<u>ssn</u>,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(<u>did</u>,dname,floor,mgr)

500 records
50 pages



MOTIVATION

```
SELECT DISTINCT ename
  FROM Emp E JOIN Dept D
    ON E.did = D.did
 WHERE D.dname = 'Toy'
```

Catalog

clustered *unclustered* *unclustered*

Emp(ssn,ename,addr,sal,did)

10,000 records
1,000 pages

clustered *unclustered*

Dept(did,dname,floor,mgr)

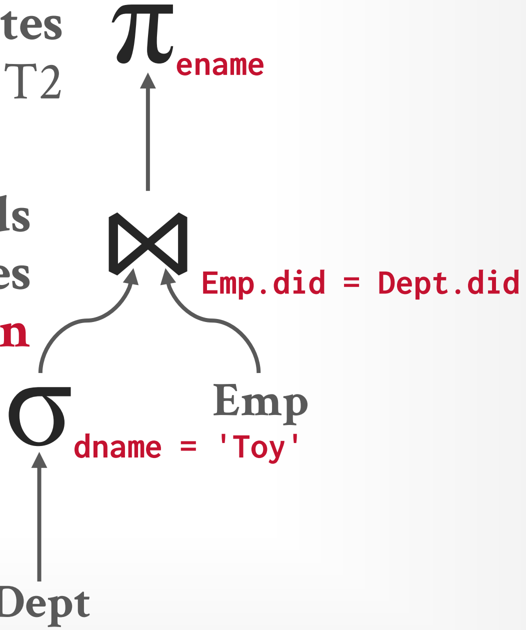
500 records
50 pages

Total: 37 I/Os

4 reads + 1 writes
Read temp T2

1 + 3 (idx) + 20 (ptr chase) reads
+ 4 writes
Index Nested-Loop Join

3 reads + 1 writes
Access: Index(dname)



TODAY'S AGENDA

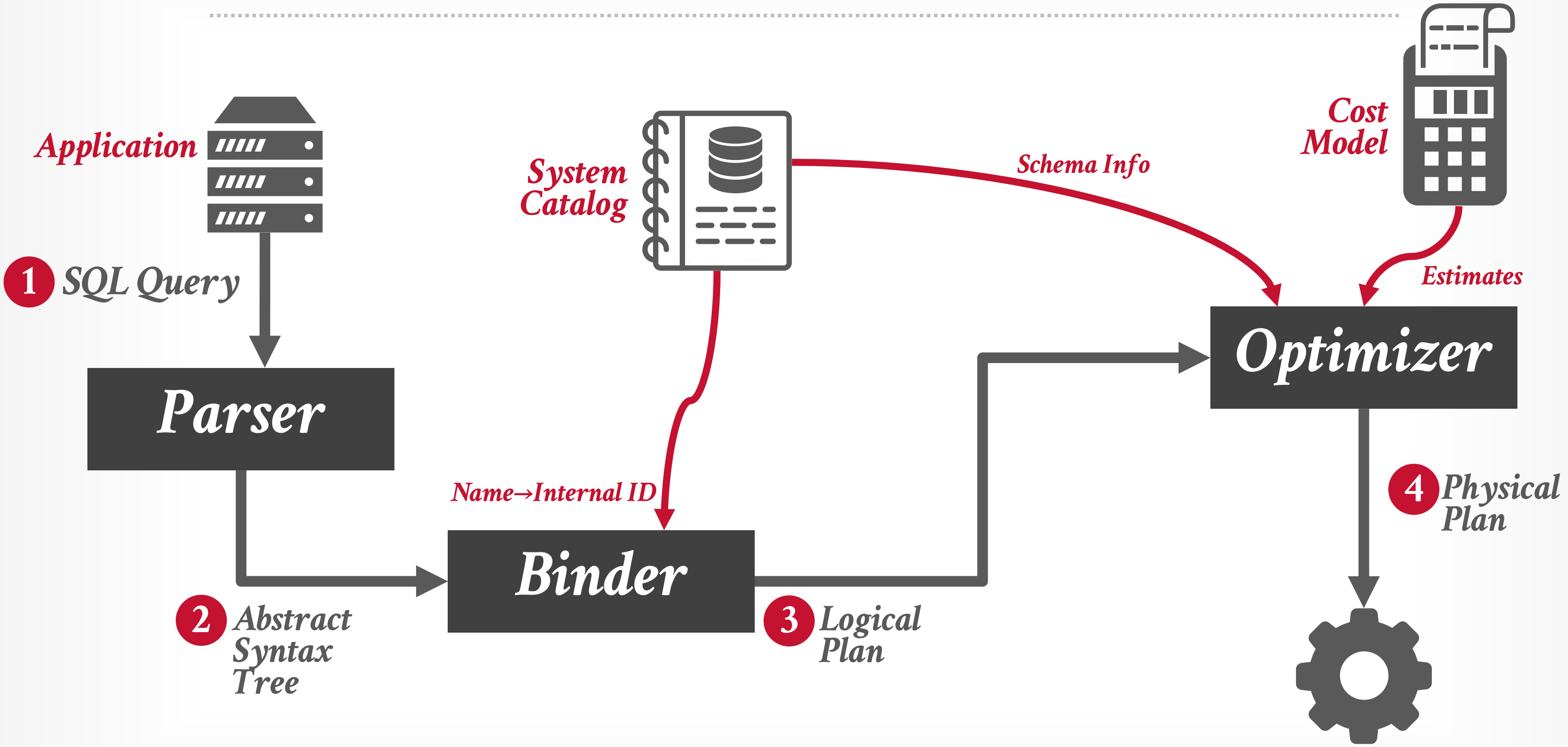
Background

Heuristic / Ruled-based Optimization

Cost-based Optimization

Cost Model Estimation

ARCHITECTURE OVERVIEW



LOGICAL VS. PHYSICAL PLANS

The optimizer generates a mapping of a **logical** algebra expression to the optimal equivalent physical algebra expression.

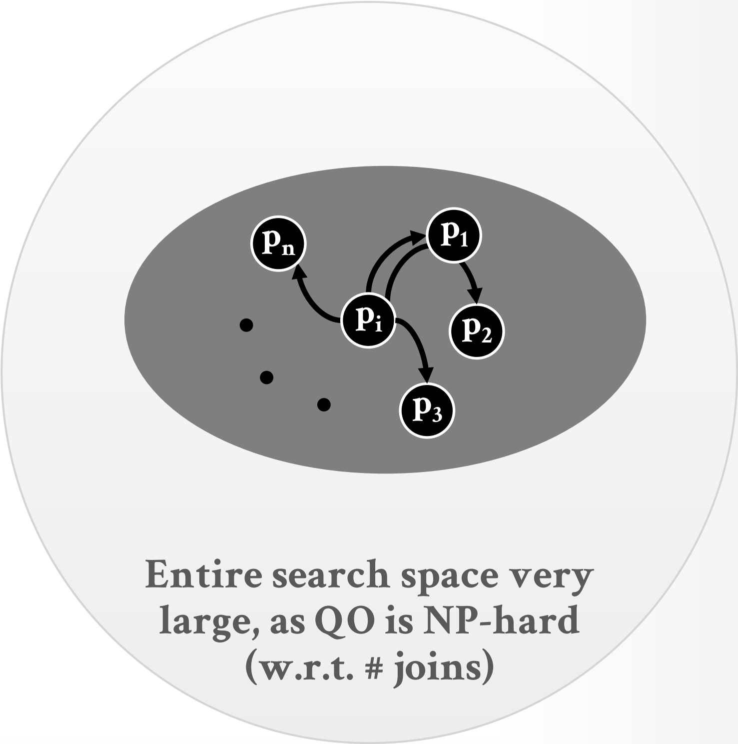
Physical operators define a specific execution strategy using an access path.

→ They can depend on the physical format of the data that they process (i.e., sorting, compression).
→ Not always a 1:1 mapping from logical to physical.

QUERY OPTIMIZATION (QO)

1. Identify candidate equivalent trees (logical). It is an NP-hard problem, so the space is large.
2. For each candidate, find the execution plan (physical). Estimate the cost of each plan.
3. Choose the best (physical) plan.

Practically: Choose from a subset of all possible plans.



QUERY OPTIMIZATION

Heuristics / Rules

→ Rewrite the query to remove (guessed) inefficiencies.
→ Examples: always do selections first or push down projections as early as possible.
→ These techniques may need to examine catalog, but they do not need to examine data.

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.

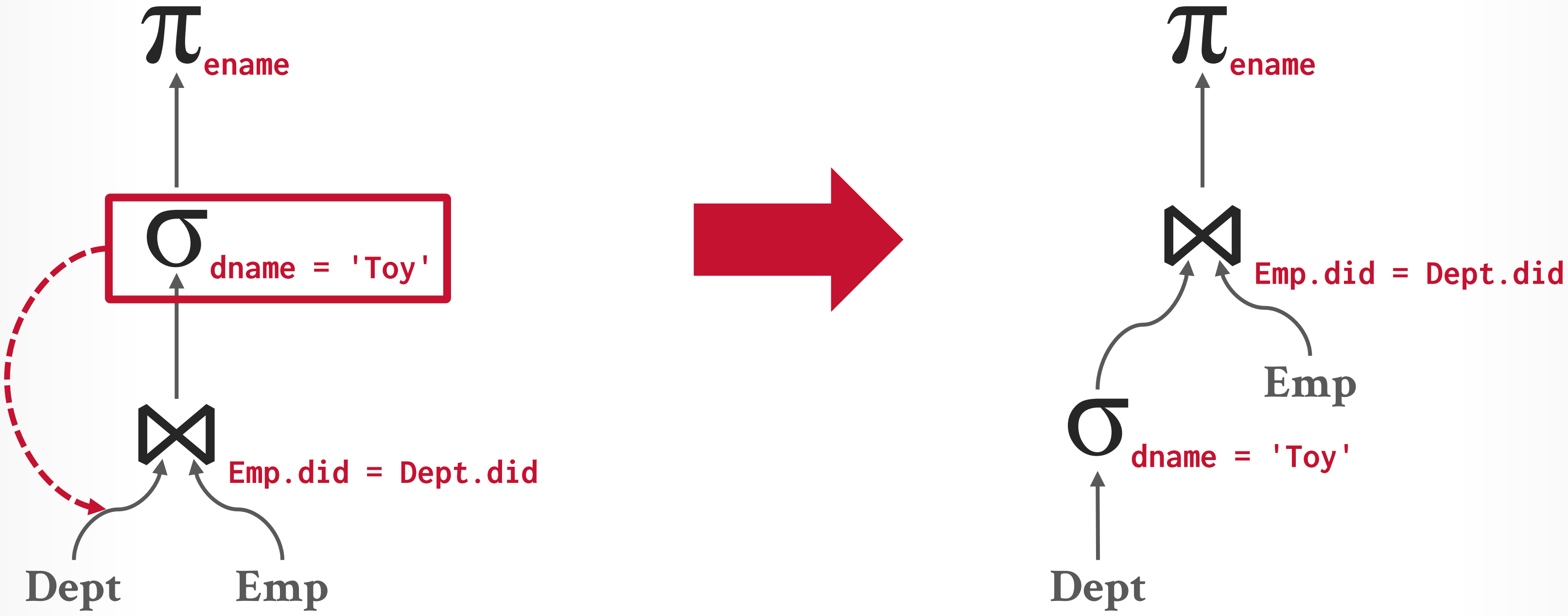
LOGICAL PLAN OPTIMIZATION

Transform a logical plan into an equivalent logical plan using pattern matching rules.

The goal is to increase the likelihood of enumerating the optimal plan in the search.
→ Many equivalence rules for relational algebra!

Cannot compare plans because there is no cost model but can "direct" a transformation to a preferred side.

PREDICATE PUSHDOWN

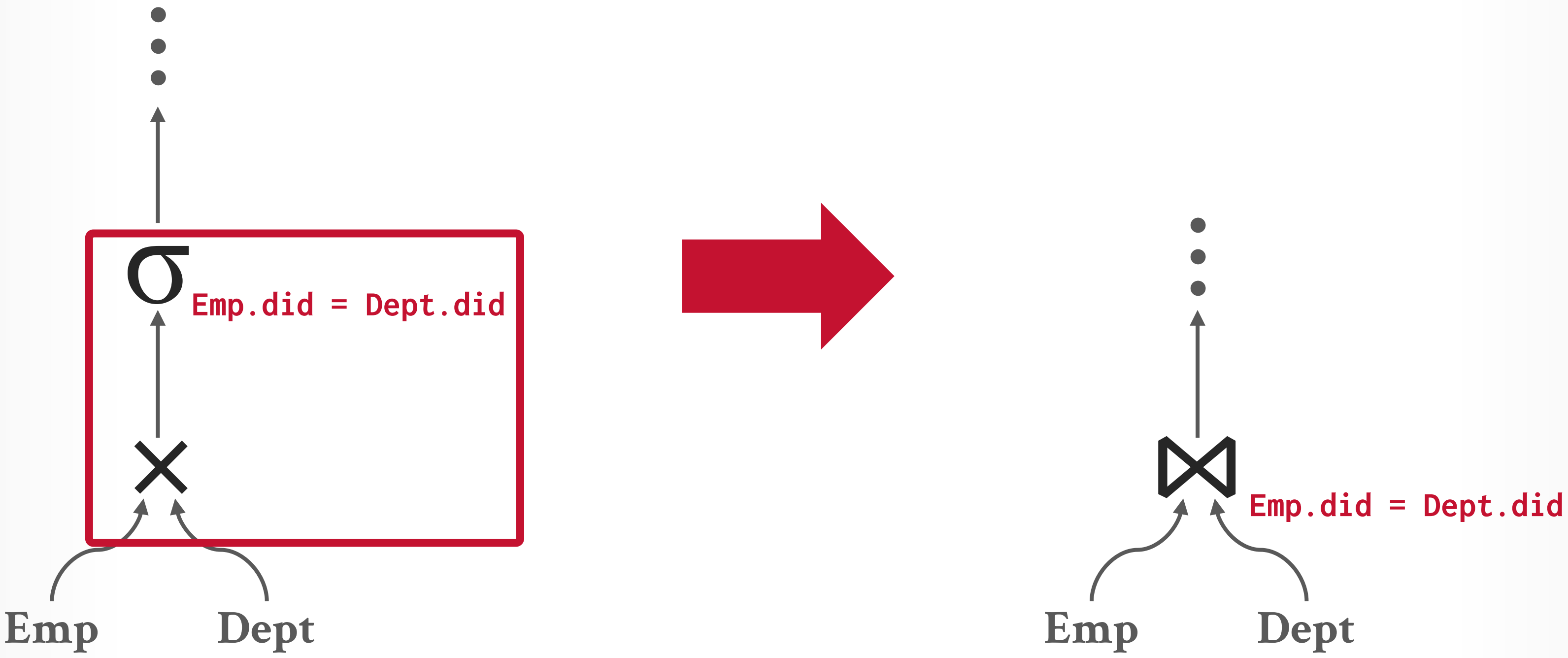


$\pi_{\text{ename}}(\sigma_{\text{dname = 'Toy'}}(\text{Dept} \bowtie \text{Emp}))$

Rewrite

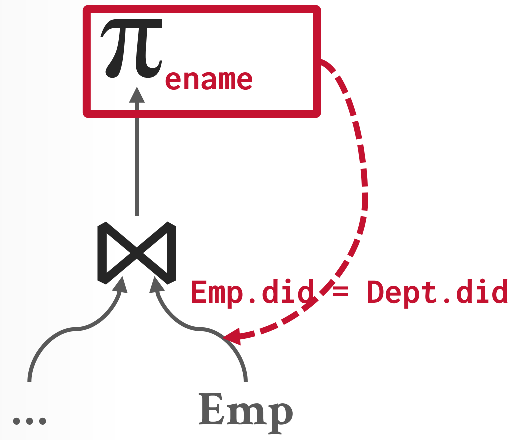
$\pi_{\text{ename}}(\text{Emp} \bowtie \sigma_{\text{dname = 'Toy'}}(\text{Dept}))$

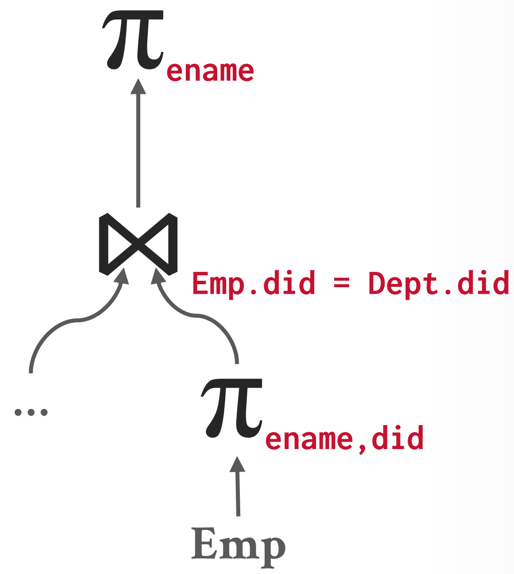
REPLACE CARTESIAN PRODUCT



$\ldots(\sigma_{\text{Dept.did = Emp.did}}(\text{Dept} \times \text{Emp}))$ **Rewrite** $\ldots(\text{Emp} \bowtie_{\text{Emp.did = Dept.did}} \text{Dept})$

PROJECTION PUSHDOWN





$\pi_{\text{Emp.ename}}(\ldots \bowtie_{\text{did}} \text{Emp})$

Rewrite

$\pi_{\text{Emp.ename}}(\ldots \bowtie_{\text{did}} (\pi_{\text{ename, did}} \text{Emp}))$

QUERY OPTIMIZATION

Heuristics / Rules

→ Rewrite the query to remove (guessed) inefficiencies.
→ Examples: always do selections first or push down projections as early as possible.
→ These techniques may need to examine catalog, but they do not need to examine data.

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

We will start with cost-based, bottom-up QO
→ Aka the "classic" IBM System R optimizer

Approach: Enumerate different plans for the query and estimate their costs.
→ Single relation.
→ Multiple relations.
→ Nested sub-queries.

It chooses the best plan it has seen for the query after exhausting all plans or some timeout.

SINGLE-RELATION QUERY PLANNING

Pick the best access method.
→ Sequential Scan
→ Binary Search (clustered indexes)
→ Index Scan

Predicate evaluation ordering.

Simple heuristics are often good enough for this.

MULTI-RELATION QUERY PLANNING

Approach #1: Generative / Bottom-Up

→ Start with nothing and then iteratively assemble and add building blocks to generate a query plan.
→ **Examples:** System R, Starburst

Approach #2: 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

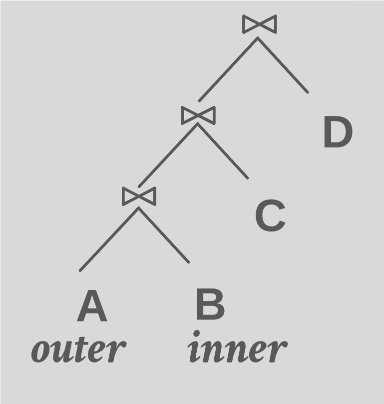
Break query into blocks and generate logical operators for each block.

For each logical operator, generate a set of physical operators that implement it.

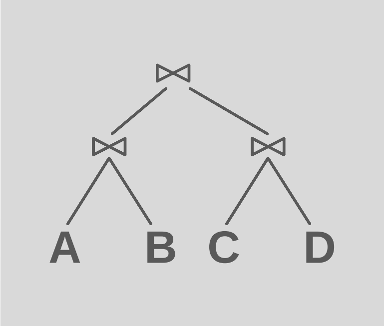
→ All combinations of join algorithms and access paths

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

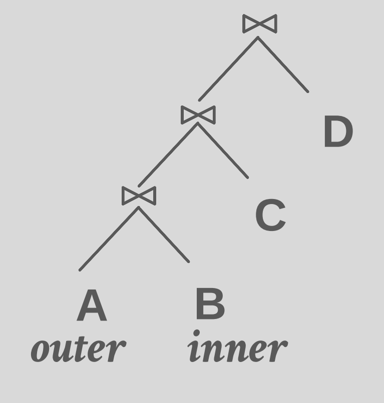
Break query into blocks and generate logical operators for each block.

For each logical operator, generate a set of physical operators that implement it.

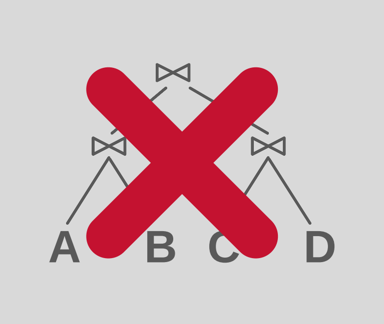
→ All combinations of join algorithms and access paths

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

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

ARTIST ⋈ APPEARS ⋈ ALBUM
APPEARS ⋈ ALBUM ⋈ ARTIST
ALBUM ⋈ APPEARS ⋈ ARTIST
APPEARS ⋈ ARTIST ⋈ ALBUM
ARTIST × ALBUM ⋈ APPEARS
ALBUM × ARTIST ⋈ APPEARS
⋮ ⋮ ⋮

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**

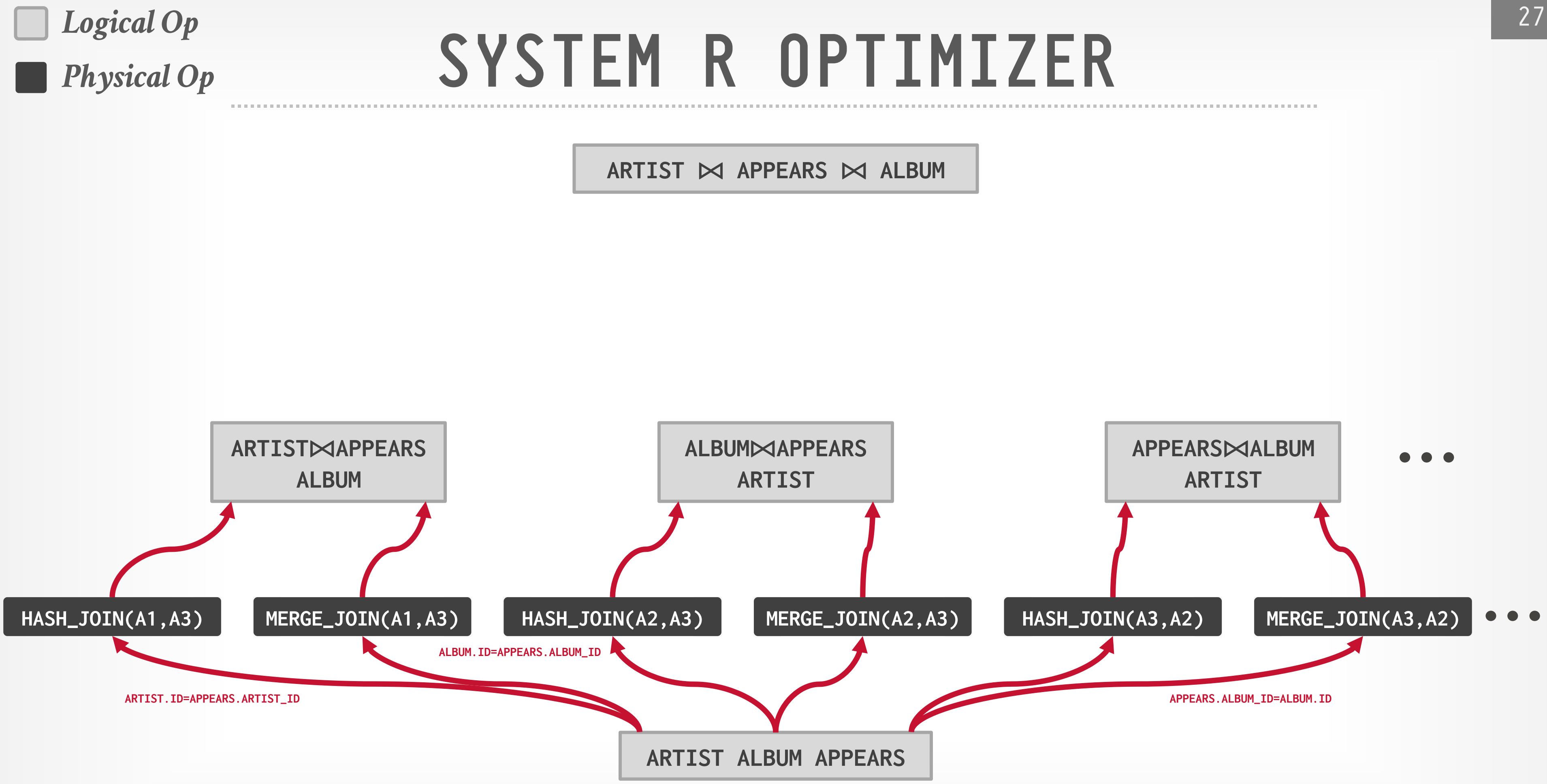
Step #1: Choose the best access paths to each table

Step #2: Enumerate all possible join orderings for tables

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

ARTIST ⋈ APPEARS ⋈ ALBUM
APPEARS ⋈ ALBUM ⋈ ARTIST
ALBUM ⋈ APPEARS ⋈ ARTIST
APPEARS ⋈ ARTIST ⋈ ALBUM
ARTIST × ALBUM ⋈ APPEARS
ALBUM × ARTIST ⋈ APPEARS
⋮ ⋮ ⋮

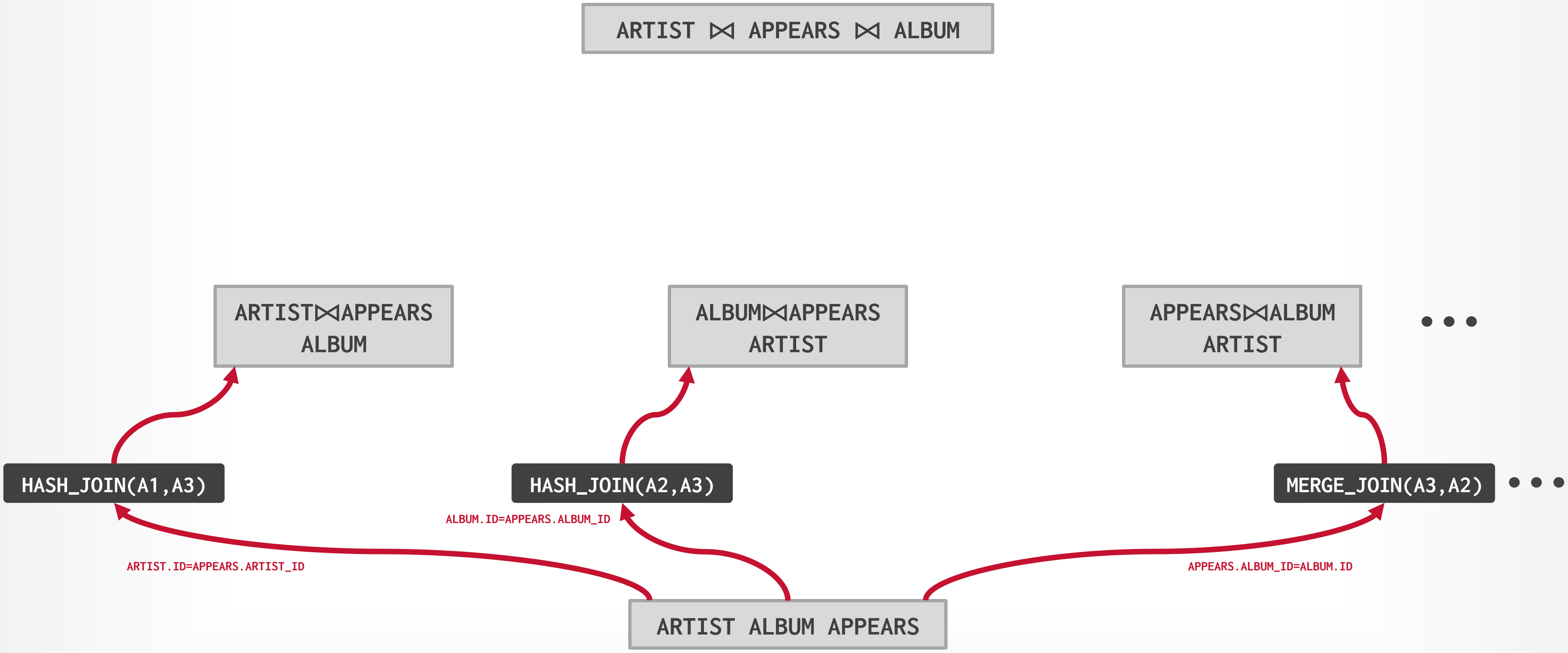
SYSTEM R OPTIMIZER



Logical Op

Physical Op

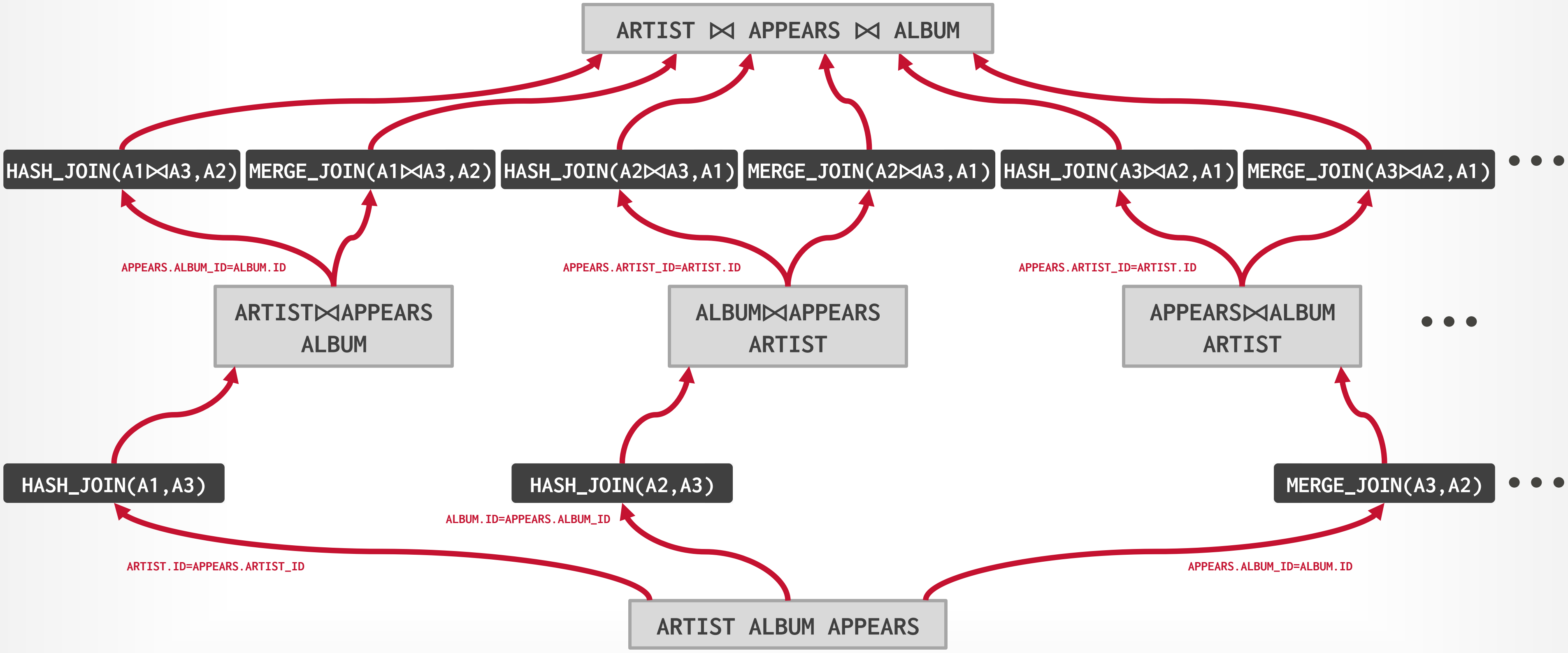
SYSTEM R OPTIMIZER



Logical Op

Physical Op

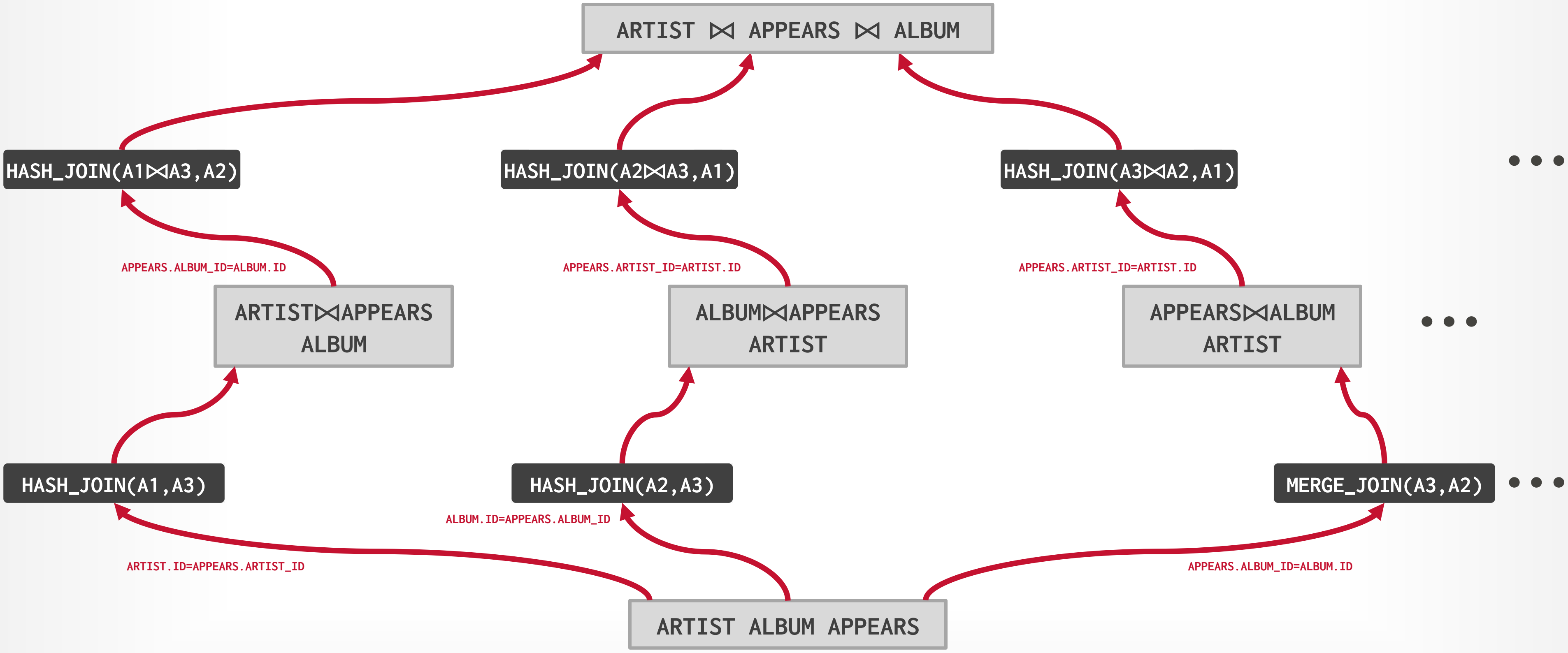
SYSTEM R OPTIMIZER



Logical Op

Physical Op

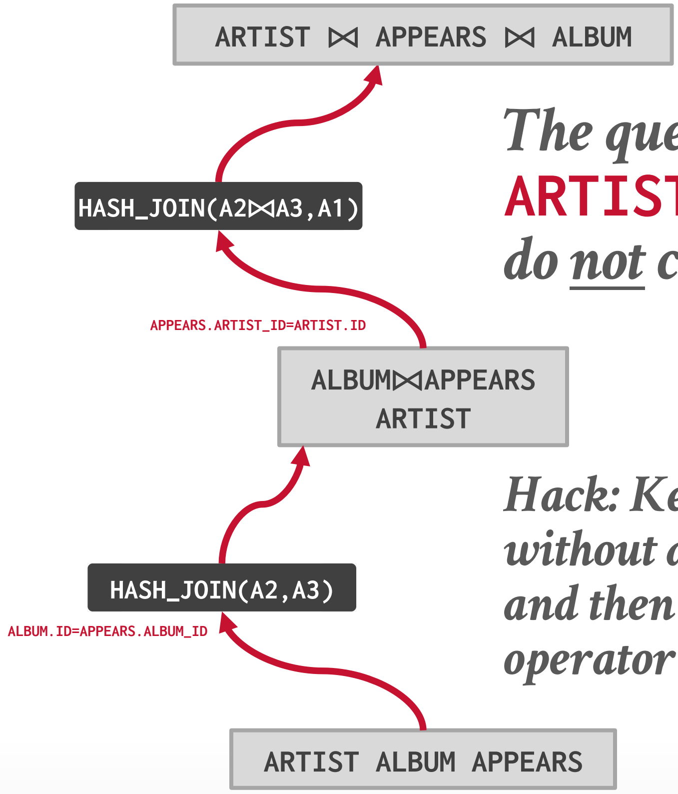
SYSTEM R OPTIMIZER



Logical Op

Physical Op

SYSTEM R OPTIMIZER



The query has **ORDER BY** *on* **ARTIST.ID** *but the logical plans do not contain sorting properties.*

Hack: Keep track of best plans with and without data in proper physical form, and then check whether tacking on a sort operator at the end is better.

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

Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

Start with a logical plan of what we want the query to be.

ARTIST ⋈ APPEARS ⋈ ALBUM
ORDER-BY(ARTIST.ID)

Logical Op

Physical Op

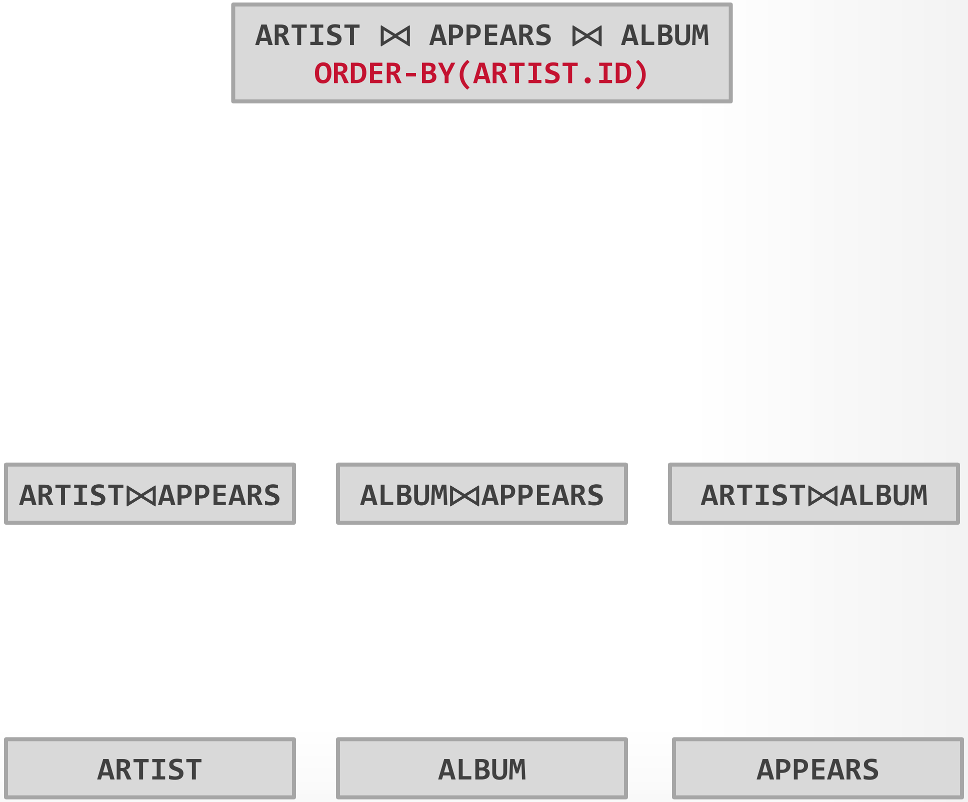
TOP-DOWN OPTIMIZATION

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)



Logical Op

Physical Op

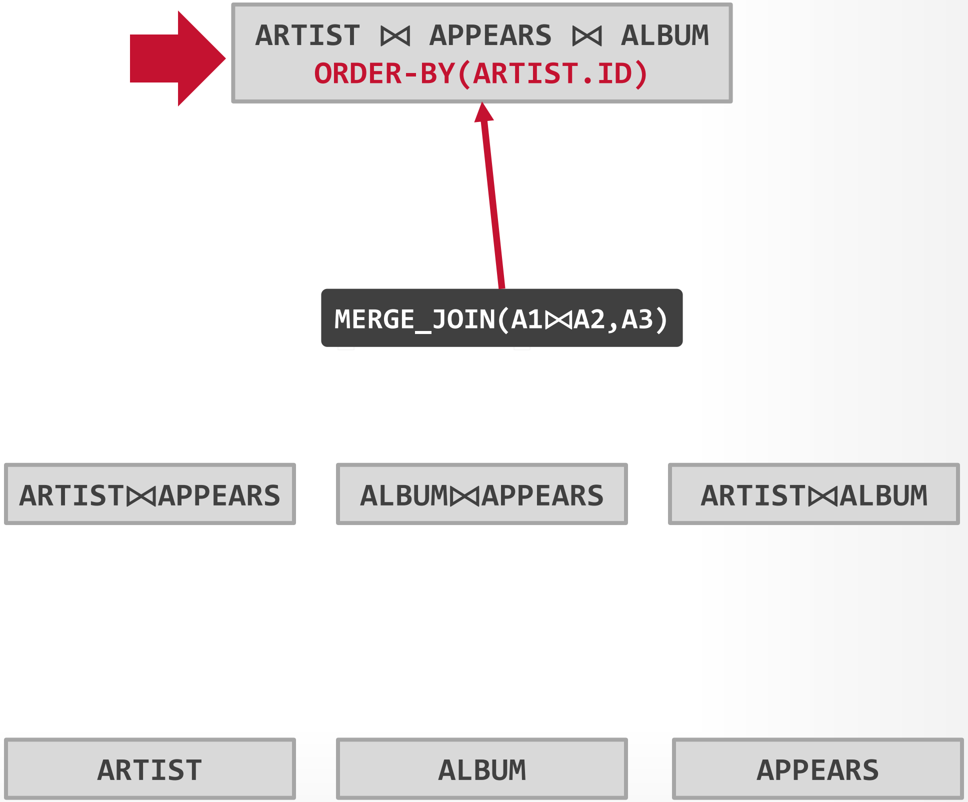
TOP-DOWN OPTIMIZATION

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)



Logical Op

Physical Op

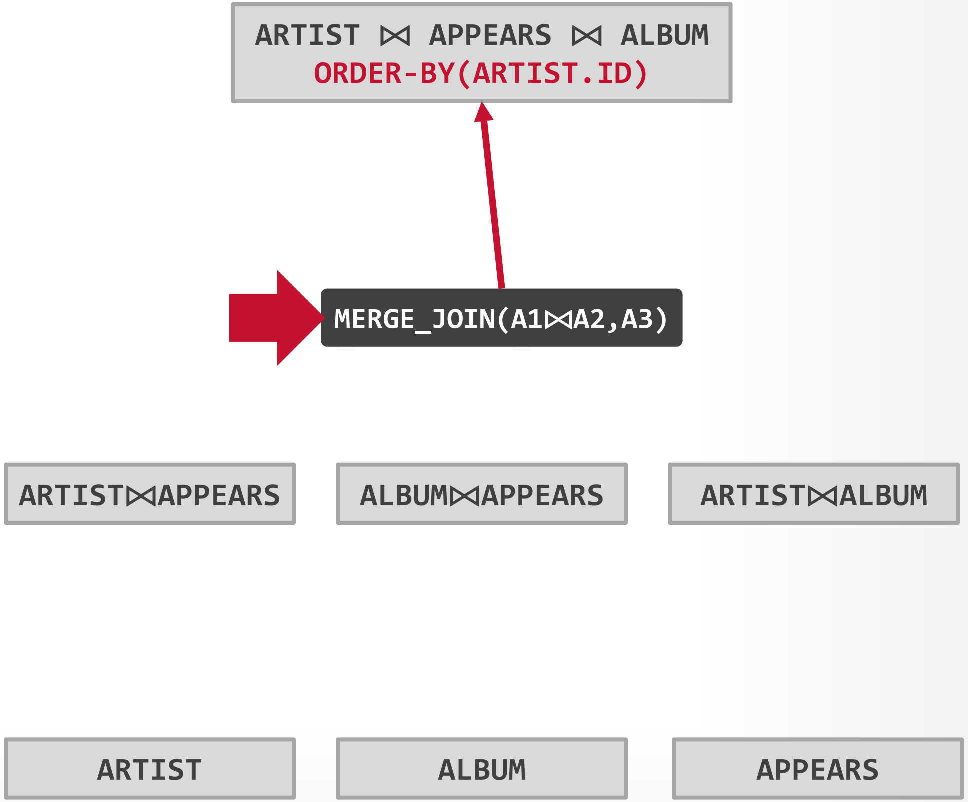
TOP-DOWN OPTIMIZATION

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)



Logical Op
Physical Op

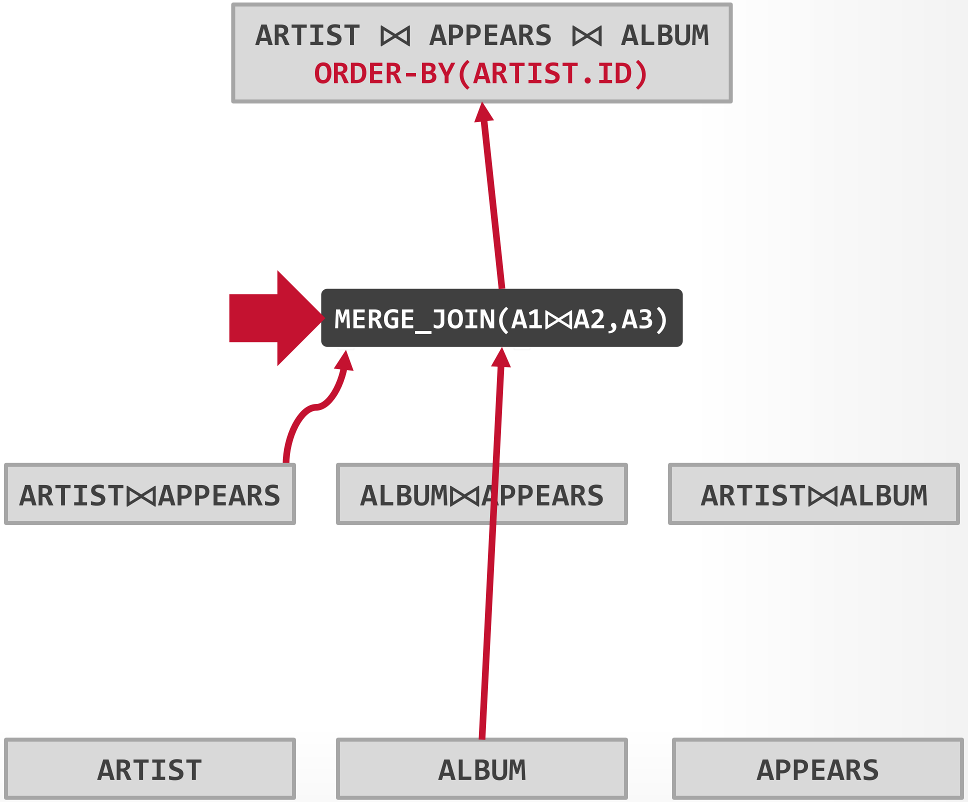
TOP-DOWN OPTIMIZATION

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)



Logical Op

Physical Op

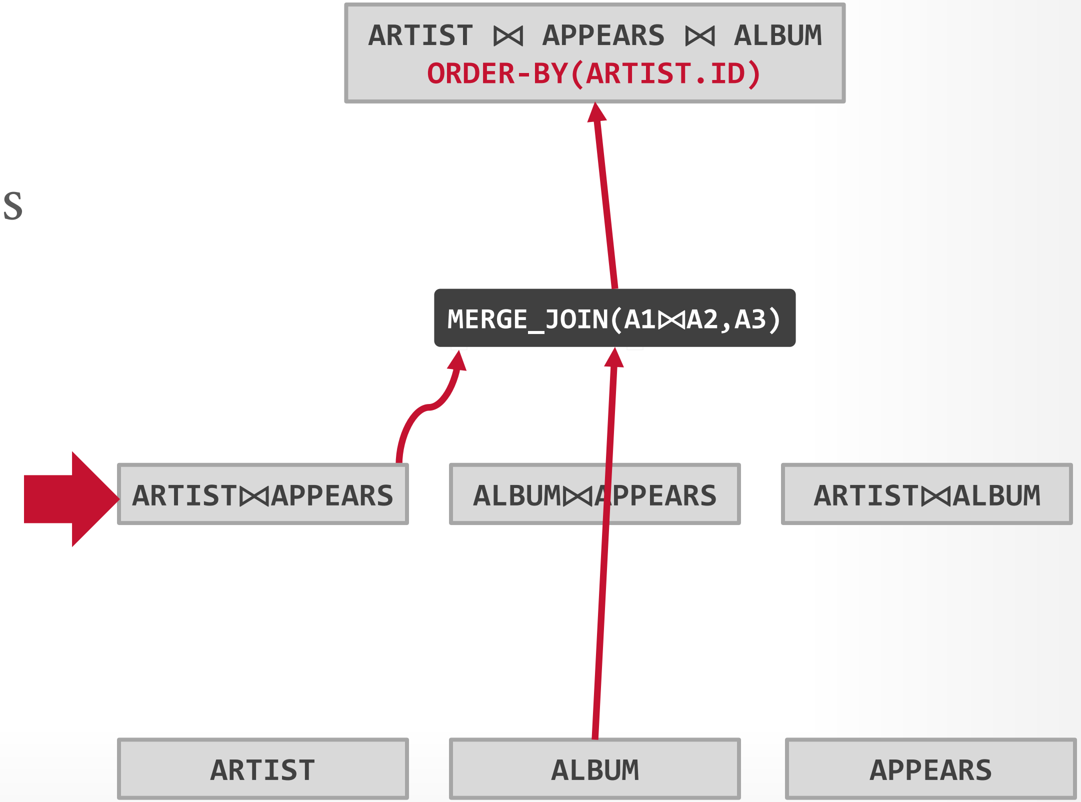
TOP-DOWN OPTIMIZATION

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)



Logical Op

Physical Op

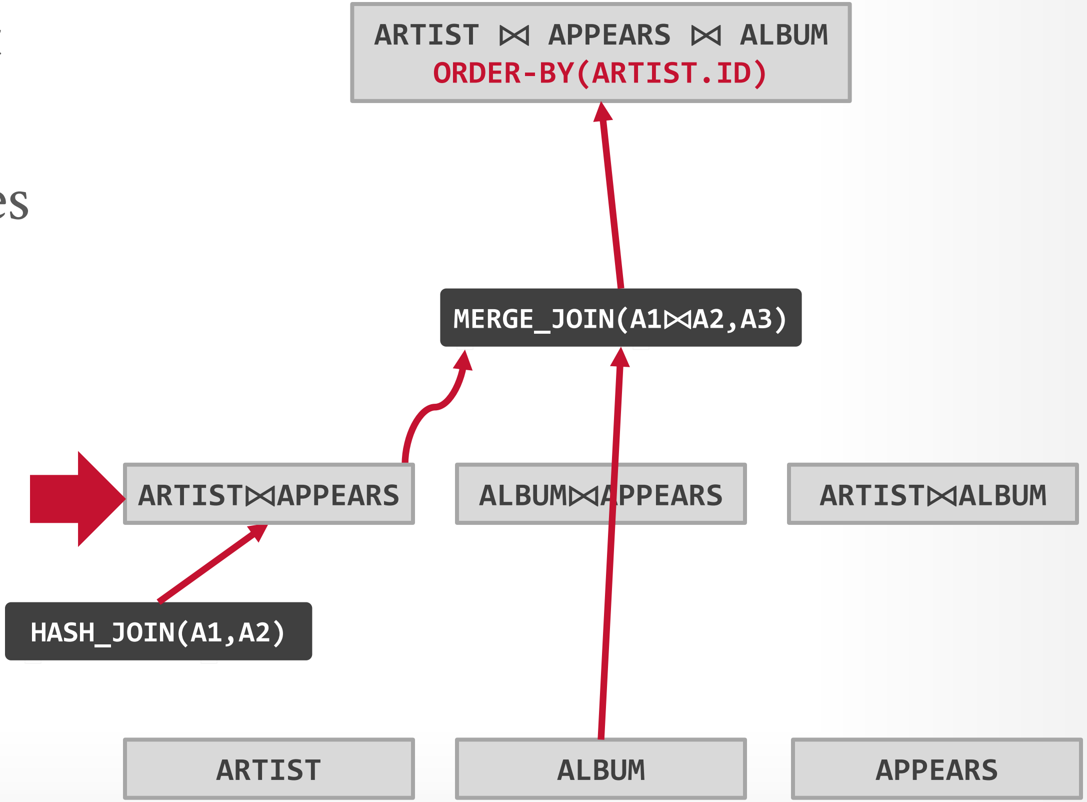
TOP-DOWN OPTIMIZATION

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)



Logical Op

Physical Op

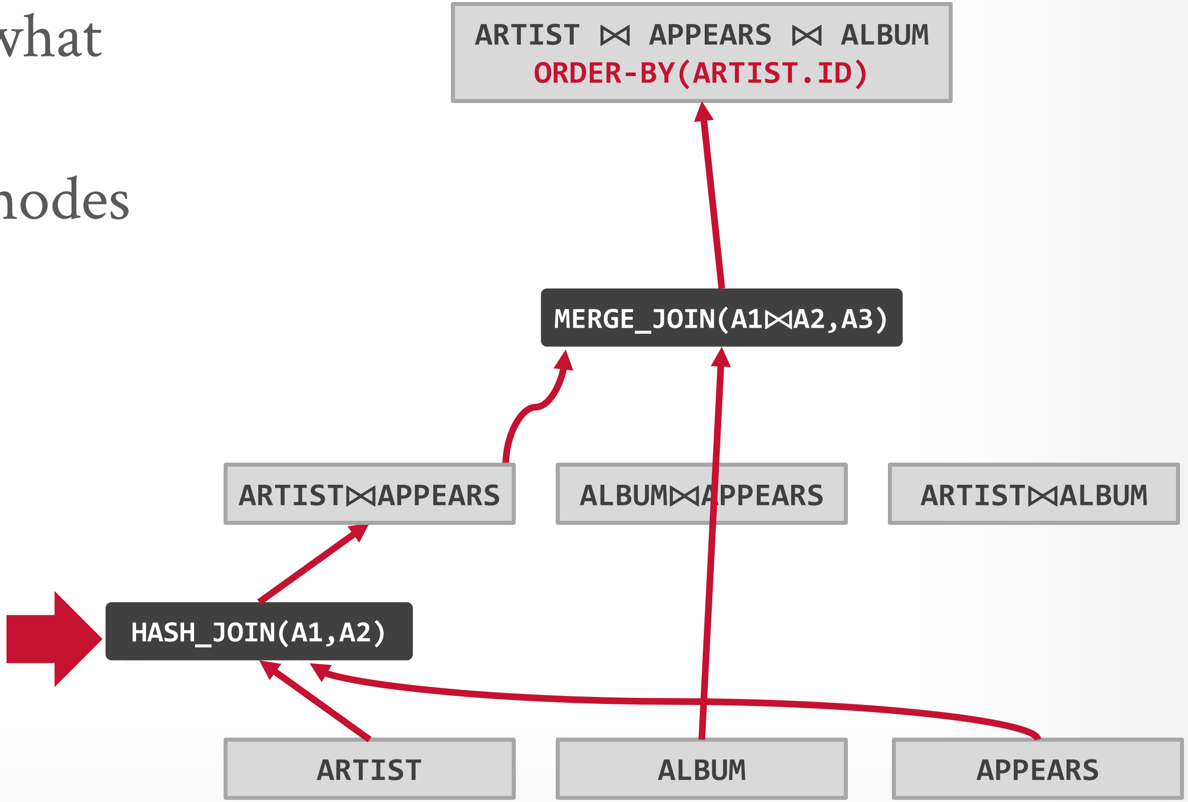
TOP-DOWN OPTIMIZATION

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)



Logical Op

Physical Op

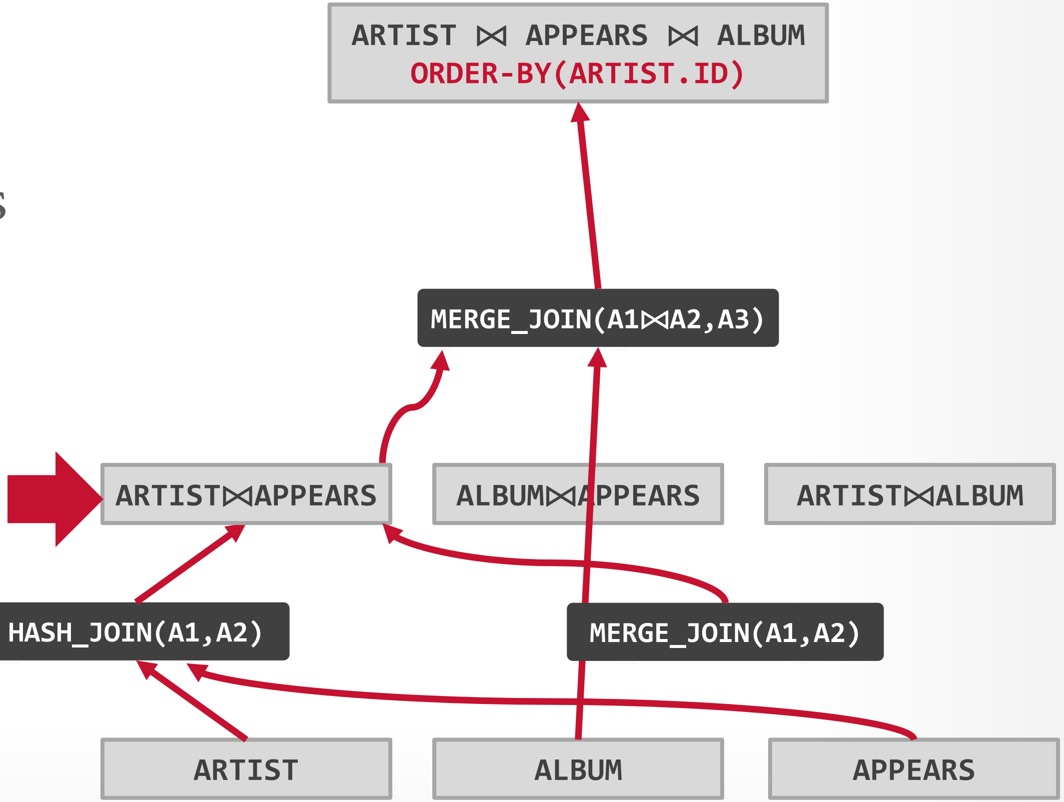
TOP-DOWN OPTIMIZATION

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)



Logical Op

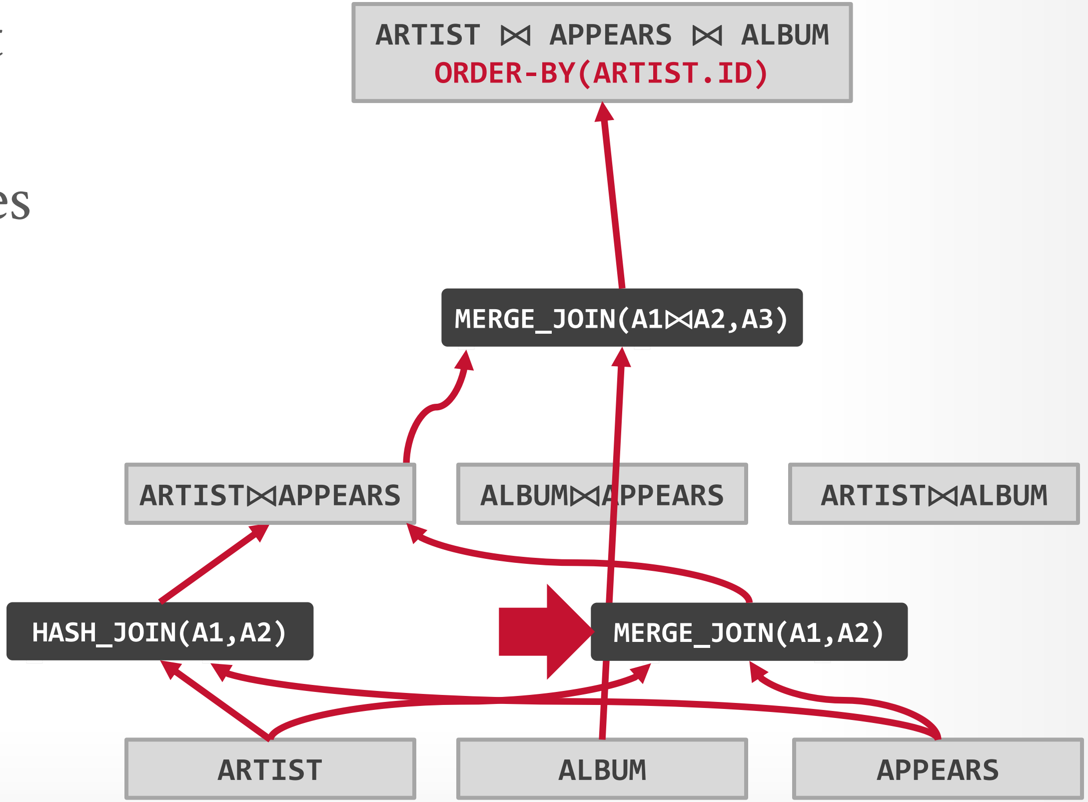
Physical Op

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)



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

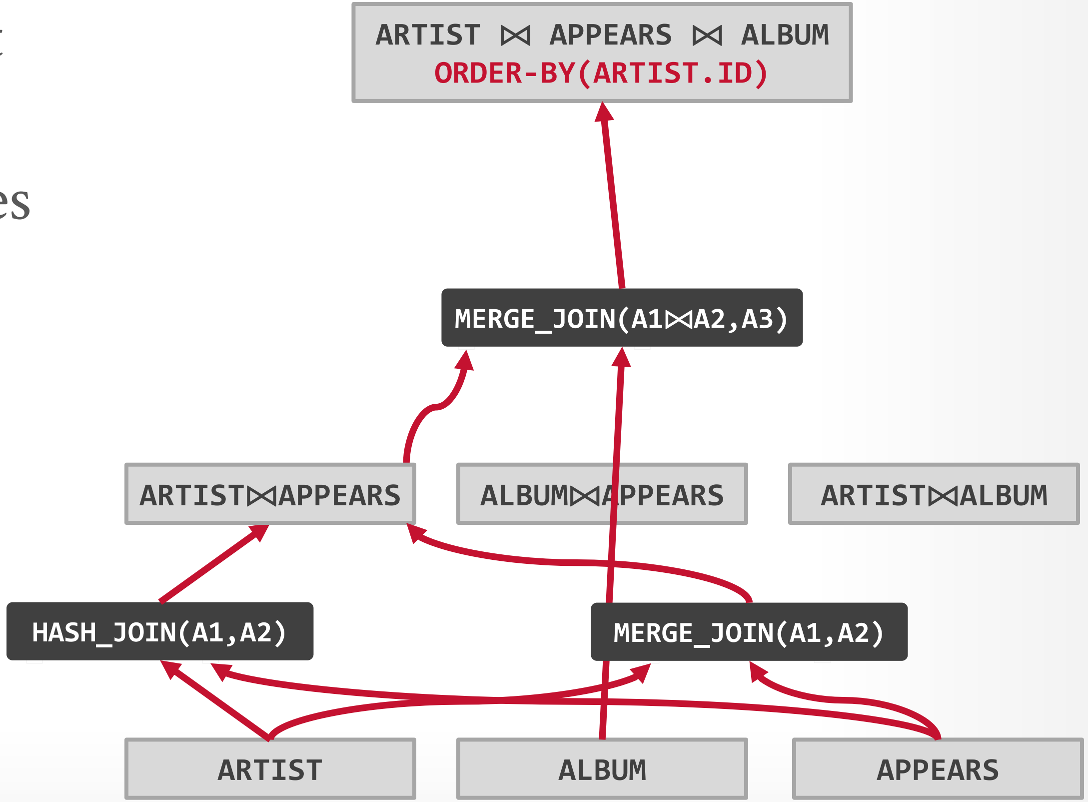
Start with a logical plan of what we want the query to be.

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.



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

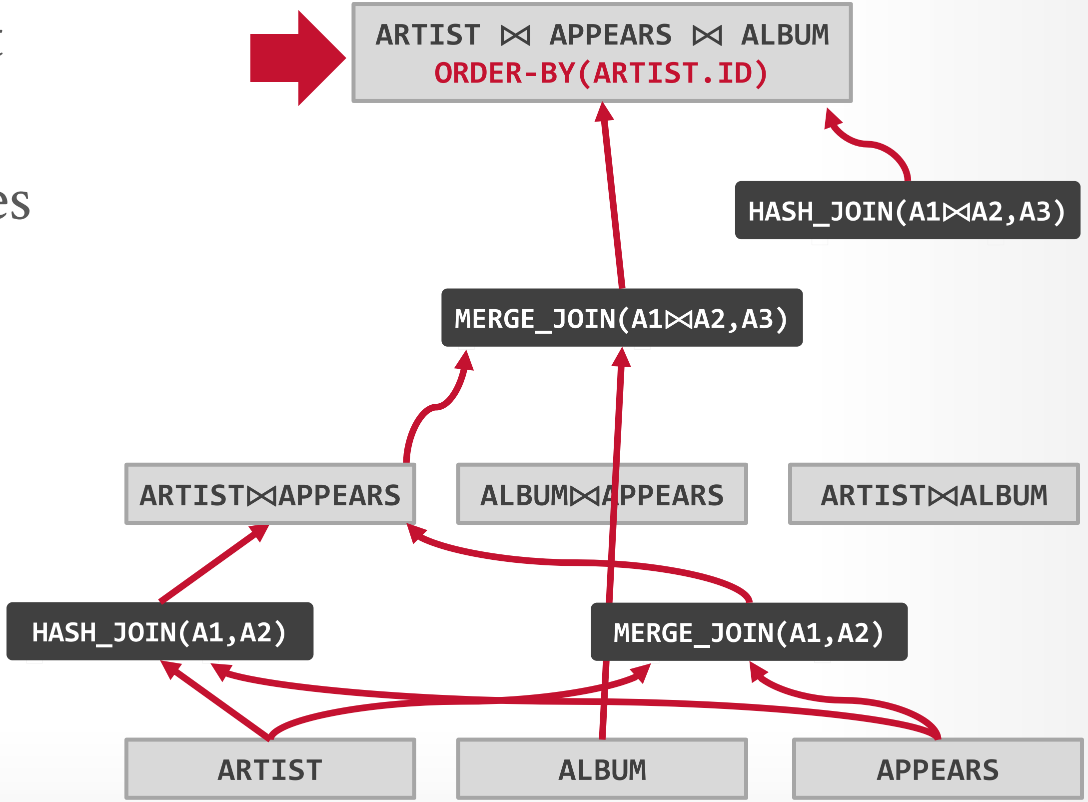
Start with a logical plan of what we want the query to be.

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.



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

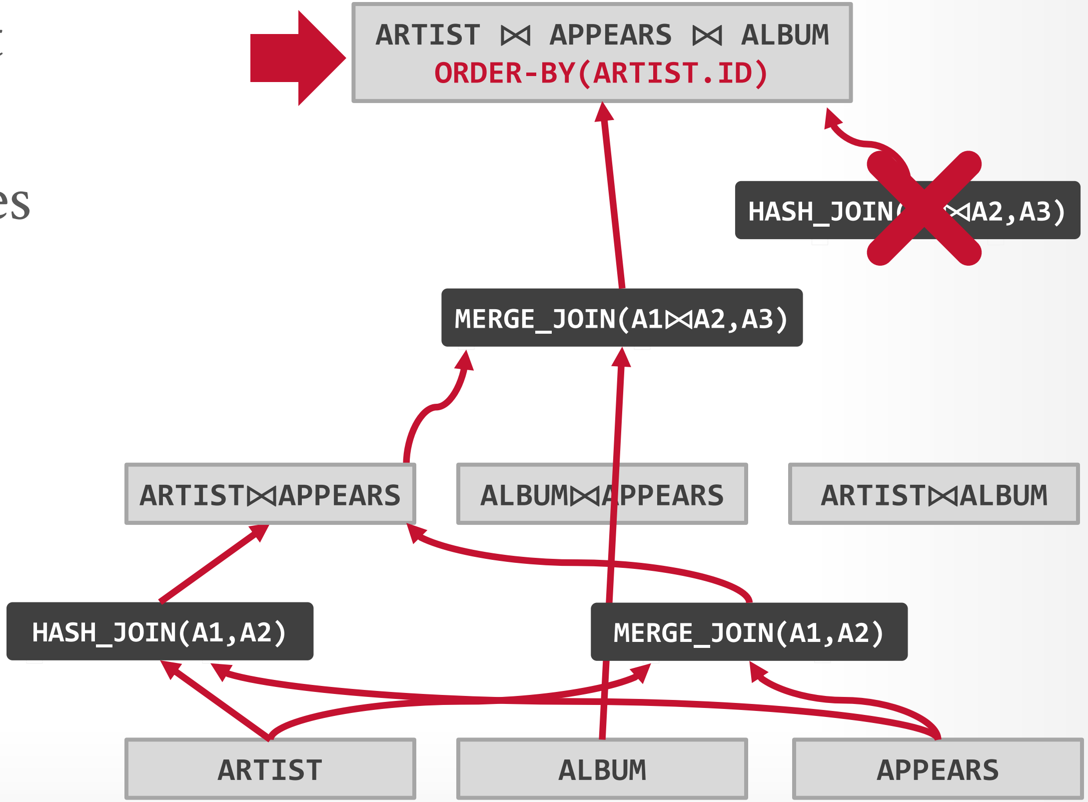
Start with a logical plan of what we want the query to be.

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.



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

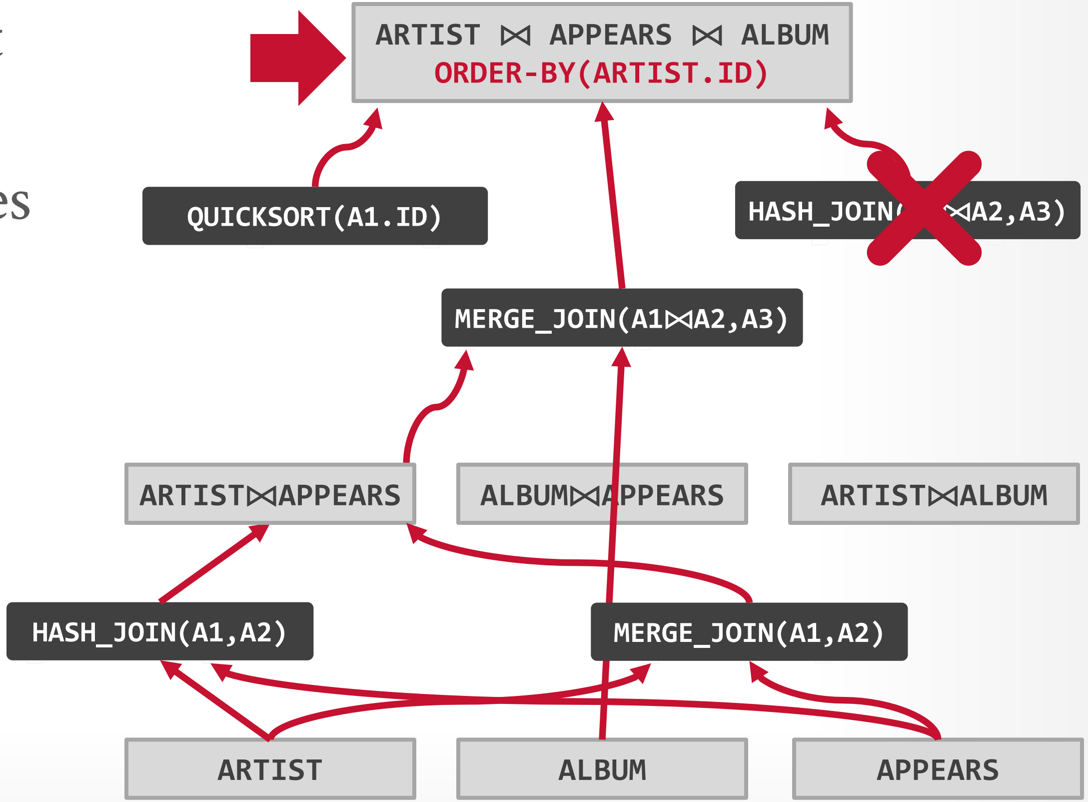
Start with a logical plan of what we want the query to be.

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.



Logical Op

Physical Op

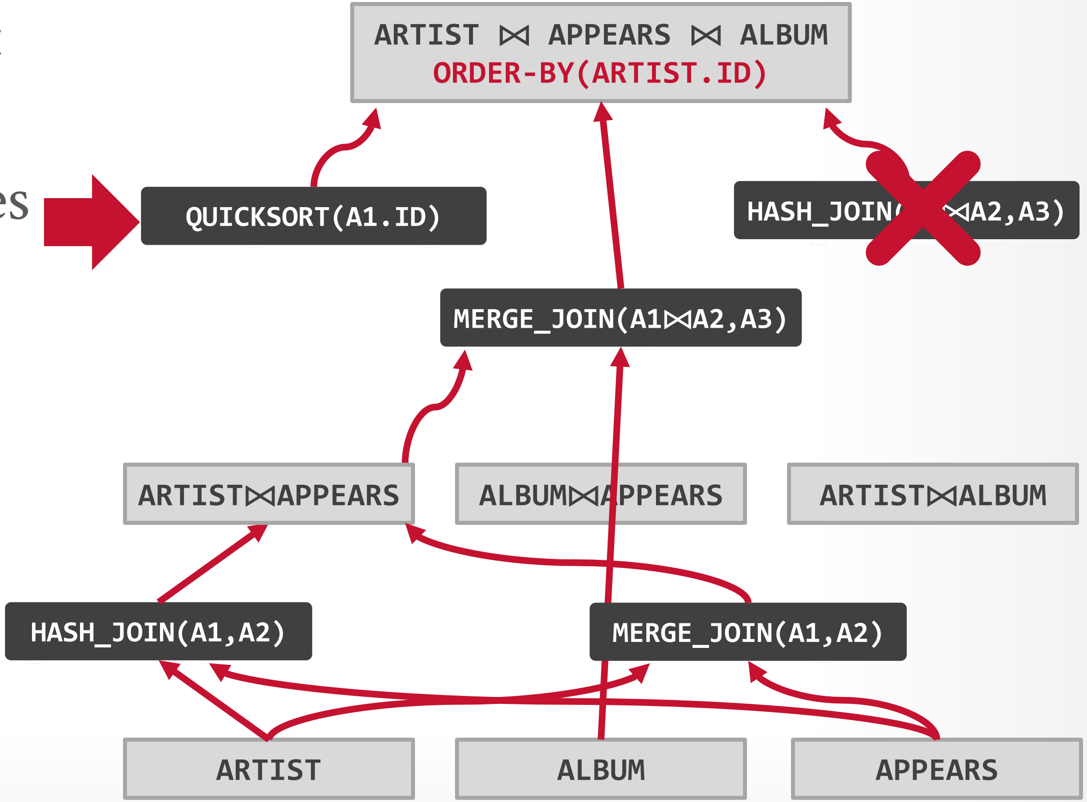
TOP-DOWN OPTIMIZATION

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.



Logical Op

Physical Op

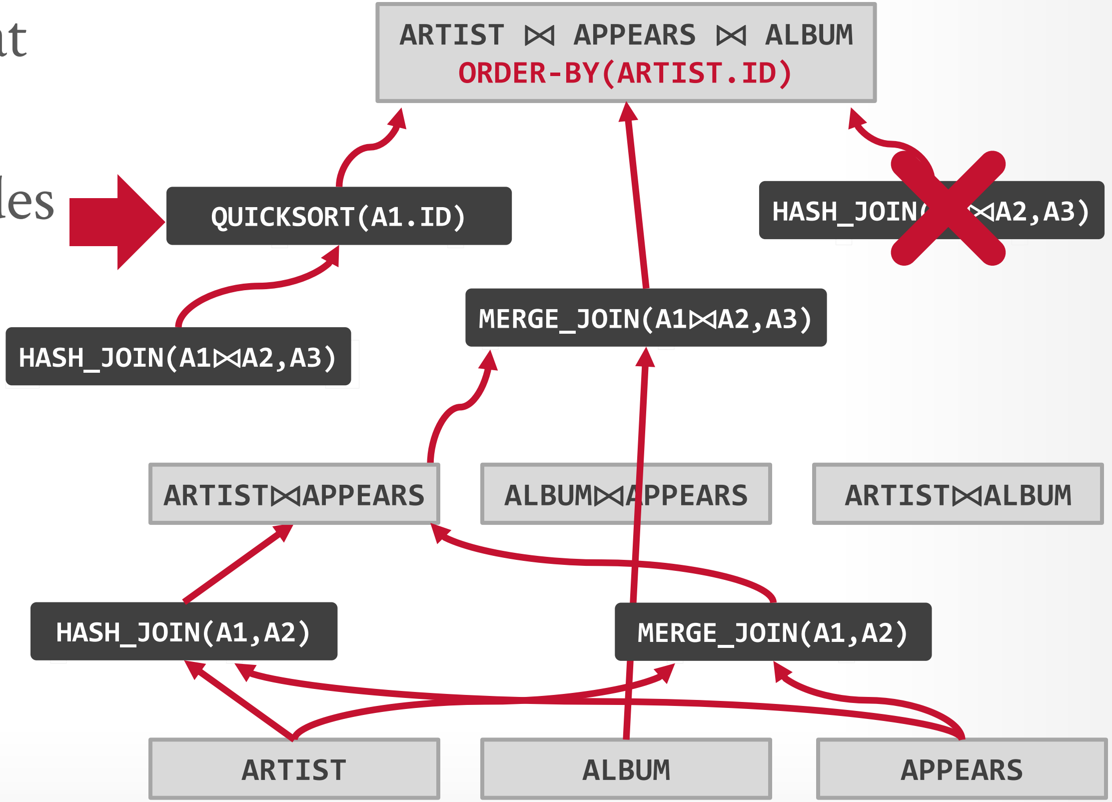
TOP-DOWN OPTIMIZATION

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.



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

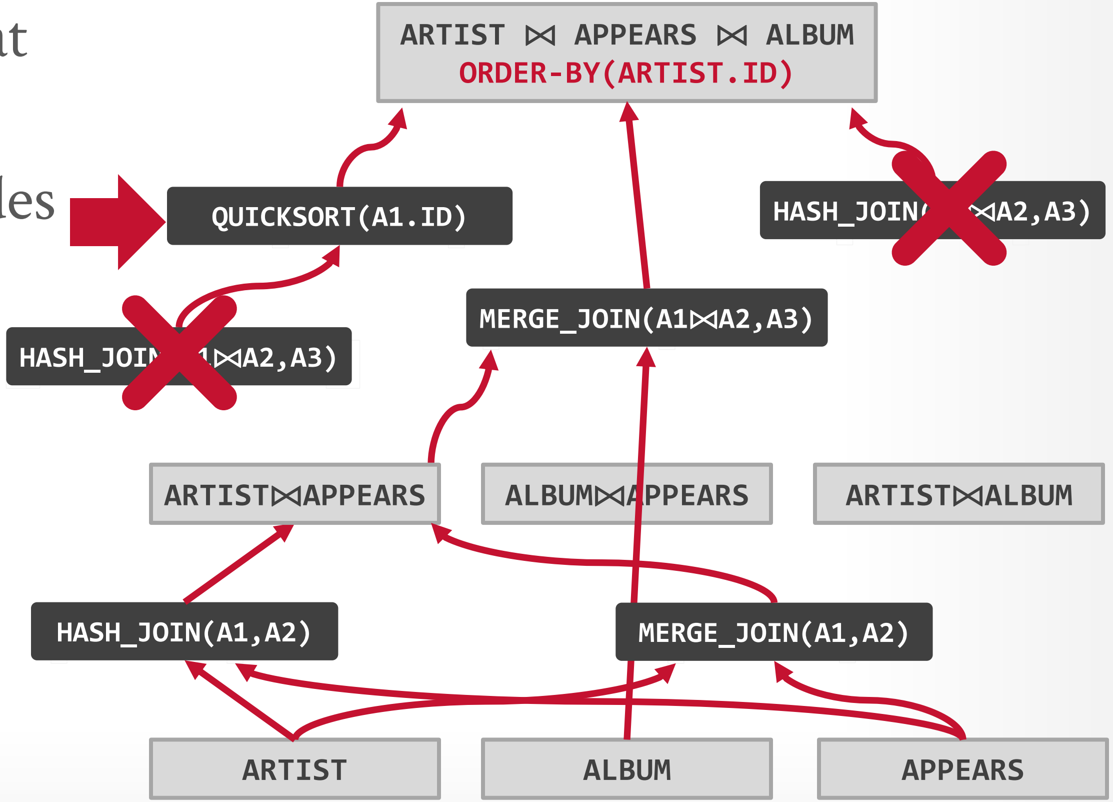
Start with a logical plan of what we want the query to be.

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.



OBSERVATION

Applications often execute nested queries.

→ We could optimize each block using the methods we have discussed.

→ However, this may be inefficient since we optimize each block separately without a global approach.

What if we could flatten a nested query into a single block and optimize it?

→ Then, apply single-block query optimization methods.

→ Even if one cannot flatten to a single block, flattening to fewer blocks is still beneficial.

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.

Approach #1: Rewrite to de-correlate and/or flatten them.

Approach #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 = '2022-10-25'
 )
```

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 = '2022-10-25'
 )
```

```
SELECT name
  FROM sailors AS S, reserves AS R
 WHERE S.sid = R.sid
   AND R.day = '2022-10-25'
```

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

```
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 = (SELECT MAX(S2.rating)
                     FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Nested Block

DECOMPOSING QUERIES

```
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
```

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;
```

EXPRESSION REWRITING

Impossible / Unnecessary Predicates

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

EXPRESSION REWRITING

Impossible / Unnecessary Predicates

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

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

EXPRESSION REWRITING

Impossible / Unnecessary Predicates

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



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

EXPRESSION REWRITING

Impossible / Unnecessary Predicates

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

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

Merging Predicates

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

EXPRESSION REWRITING

Impossible / Unnecessary Predicates

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



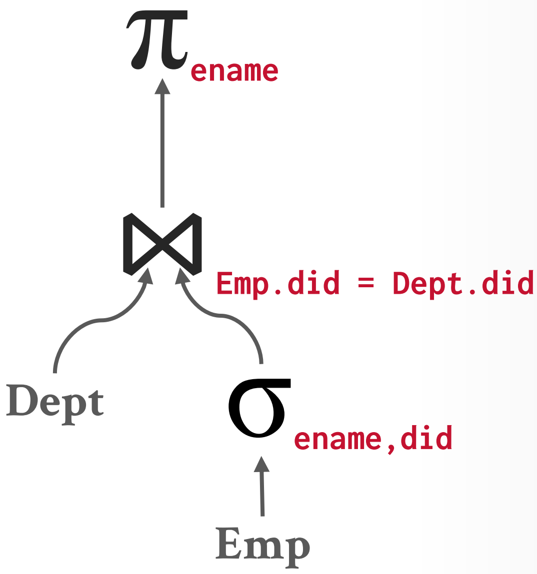
```
SELECT * FROM A WHERE false;
```


Merging Predicates

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

OBSERVATION

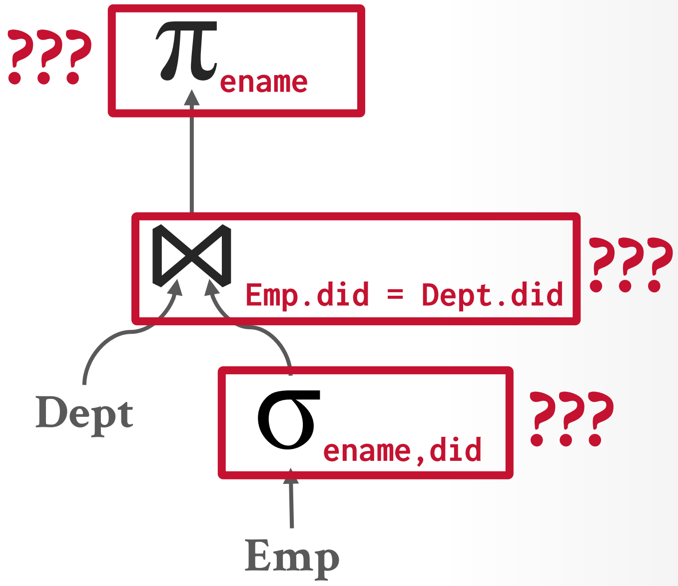
We have formulas for the operator algorithms (e.g. the cost formulas for hash join, sort merge join, ...), but we also need to estimate the size of the output that an operator produces.



OBSERVATION

We have formulas for the operator algorithms (e.g. the cost formulas for hash join, sort merge join, …), but we also need to estimate the size of the output that an operator produces.

This is hard because the output of each operators depends on its input.



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

Choice #1: Physical Costs

→ Predict CPU cycles, I/O, cache misses, RAM consumption, network messages…

→ Depends heavily on hardware.

Choice #2: Logical Costs

→ Estimate output size per operator.

→ Independent of the operator algorithm.

→ Need estimations for operator result sizes.

POSTGRES COST MODEL

Uses a combination of CPU and I/O costs that are weighted by “magic” constant factors.

Default settings are obviously for a disk-resident database without a lot of memory:

→ Processing a tuple in memory is **400x** faster than reading a tuple from disk.

→ Sequential I/O is **4x** faster than random I/O.

19.7.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, `seq_page_cost` is conventionally set to `1.0` and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

> **Note:** Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost (floating point)`

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`random_page_cost (floating point)`

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**

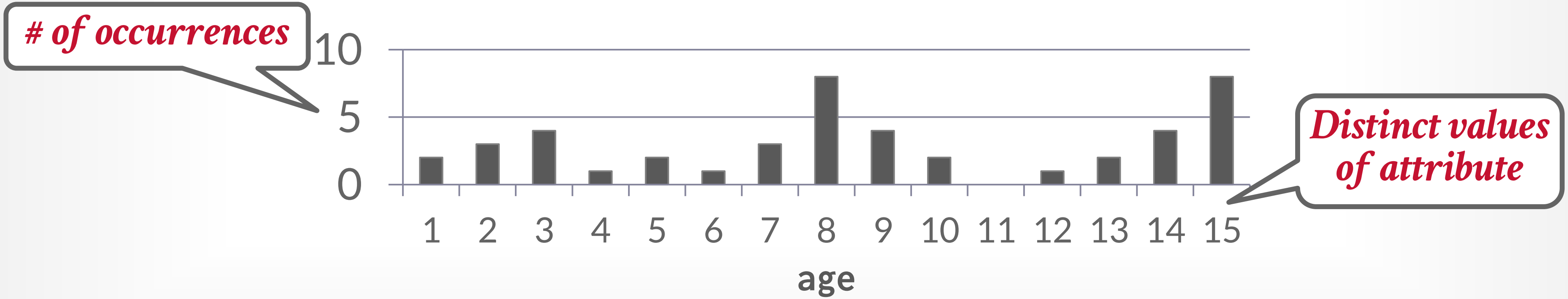
SELECTION CARDINALITY

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
```



SELECTION CARDINALITY

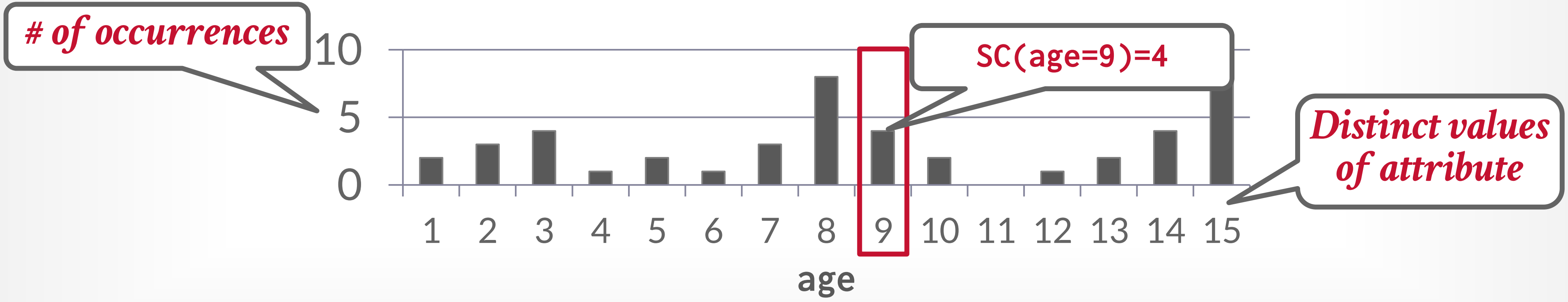
The **selectivity** (**sel**) of a predicate **P** is the fraction of tuples that qualify.

Equality Predicate: `A=constant`

→ `sel(A=constant) = #occurences/|R|`

→ Example: `sel(age=9) =`

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



SELECTION CARDINALITY

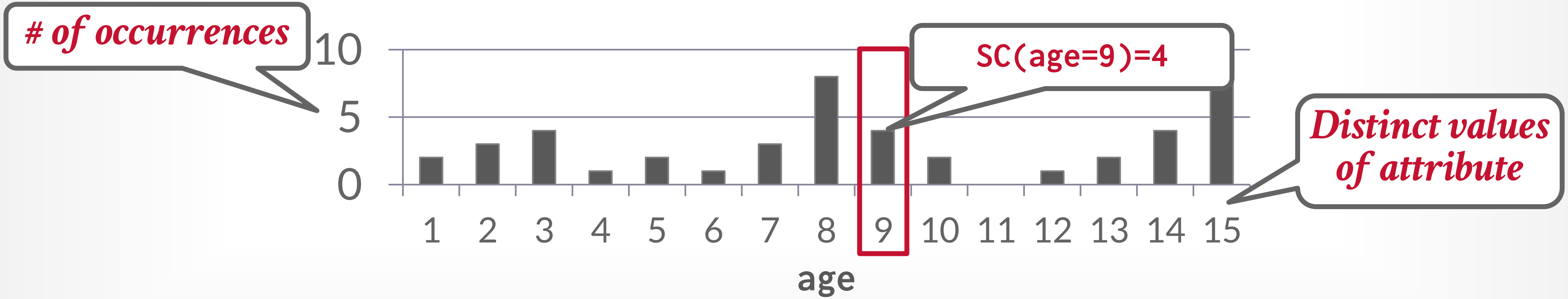
The **selectivity** (**sel**) of a predicate **P** is the fraction of tuples that qualify.

Equality Predicate: A=constant

→ sel(A=constant) = #occurences/|R|

→ Example: sel(age=9) = 4/45

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



SELECTION CARDINALITY

Assumption #1: Uniform Data

→ The distribution of values (except for the heavy hitters) is the same.

Assumption #2: Independent Predicates

→ The predicates on attributes are independent

Assumption #3: Inclusion Principle

→ The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.

CORRELATED ATTRIBUTES

Consider a database of automobiles:

→ # of Makes = 10, # of Models = 100

And the following query:

→ `(make="Honda" AND model="Accord")`

With the independence and uniformity assumptions, the selectivity is:

→ $1/10 \times 1/100 = 0.001$

But since only Honda makes Accords the real selectivity is $1/100 = 0.01$

Source: Guy Lohman

STATISTICS

Choice #1: Histograms

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

Choice #2: Sketches

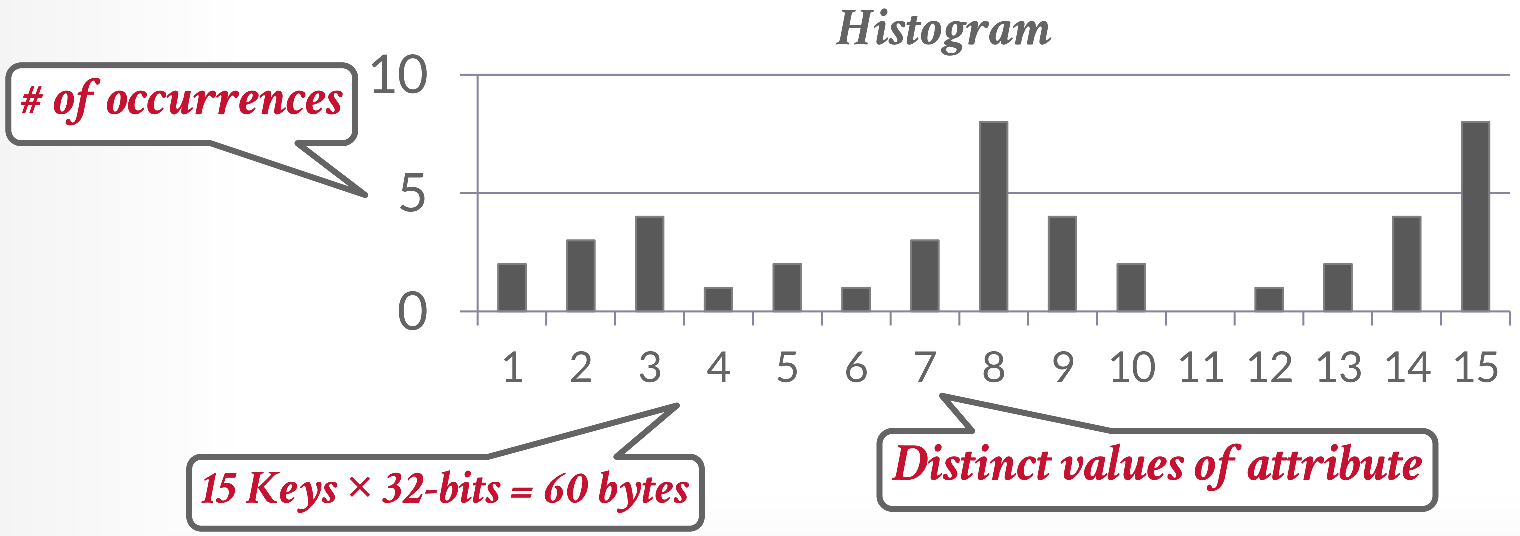
→ Probabilistic data structure that gives an approximate count for a given value.

Choice #3: Sampling

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

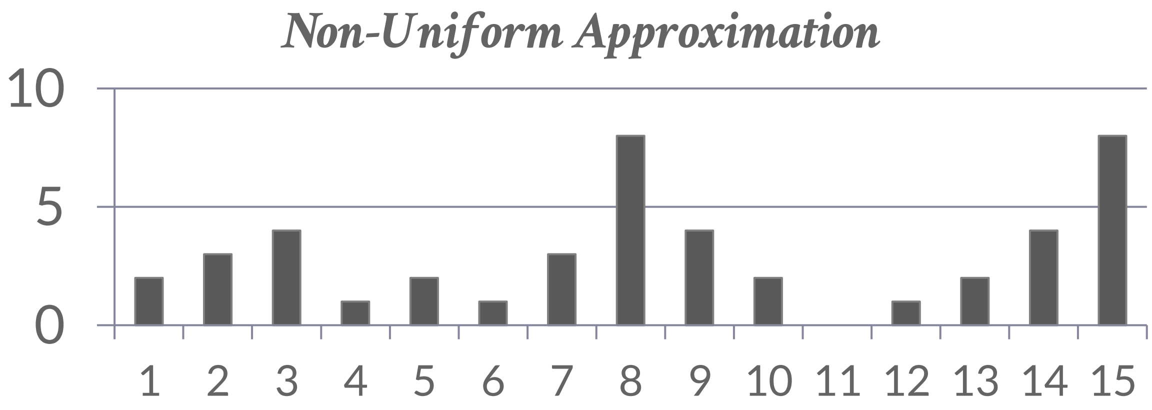
HISTOGRAMS

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



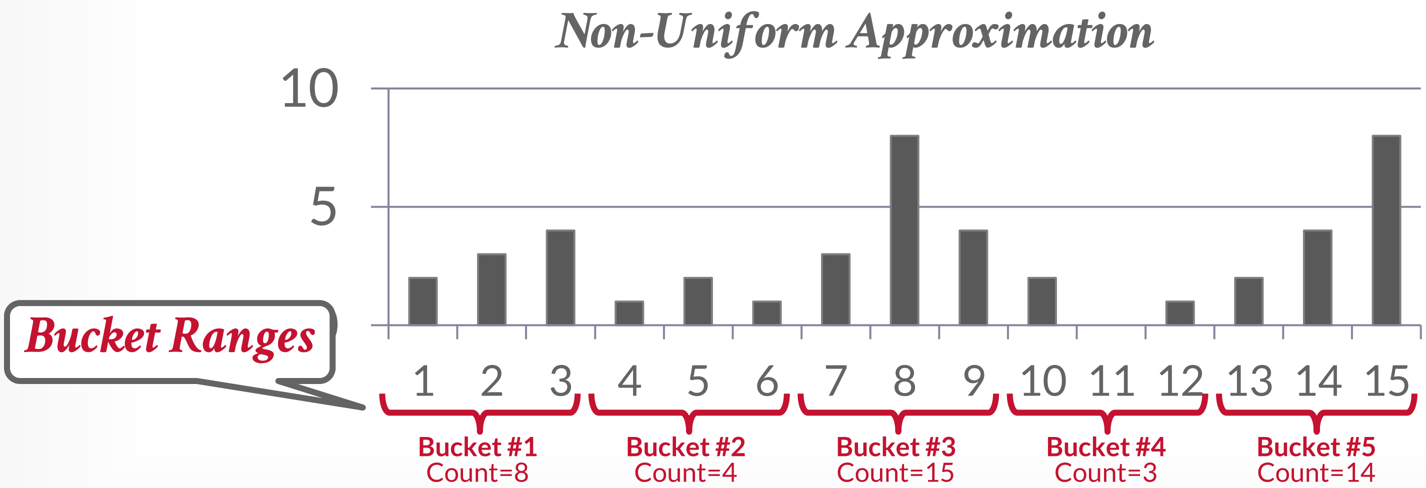
EQUI-WIDTH HISTOGRAM

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



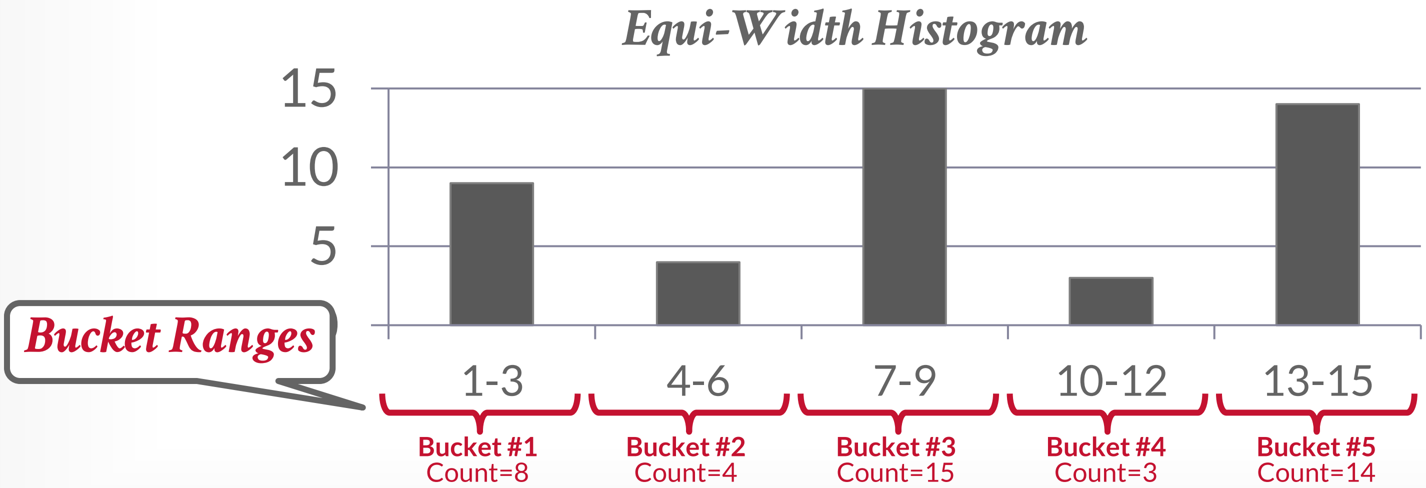
EQUI-WIDTH HISTOGRAM

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



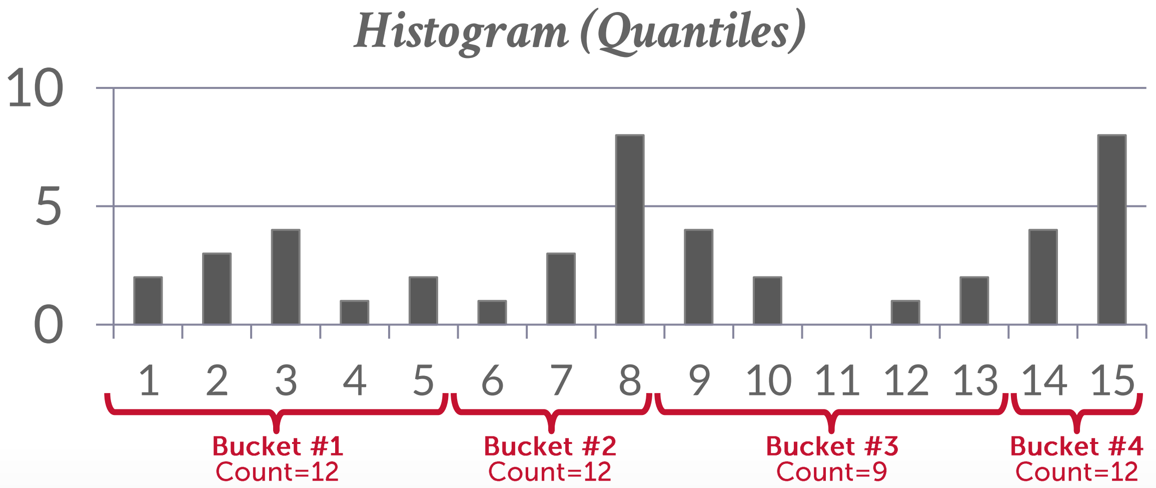
EQUI-WIDTH HISTOGRAM

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



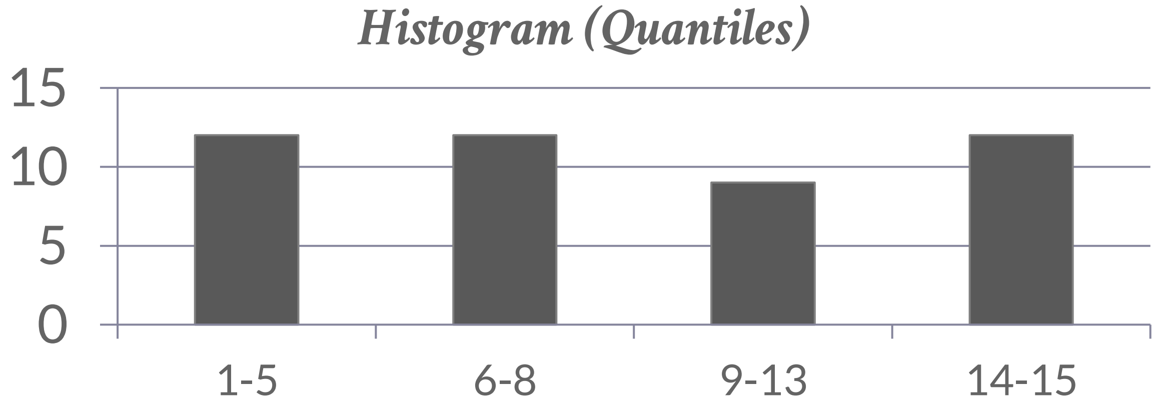
EQUI-DEPTH HISTOGRAMS

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.



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 (1988): Approximate frequency count of elements in a set.

→ HyperLogLog (2007): 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

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

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

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.

If you like this and want to make cash money after you leave CMU, take **15-799** in spring 2025.

NEXT CLASS

Transactions!
→ aka the second hardest part about database systems