Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #15

Query Planning & Optimization

SPRING 2024 ❯❯ Prof. Jignesh Patel

ADMINISTRIVIA

Project #3 is due Sun April 7, 2024 @ 11:59pm

Mid-Term

→ See me during OH for exam viewing

Final Exam

→ Thu May 2, 2024, @ 05:30pm-08:30pm

Query

SELECT distinct ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'

Catalog

clustered | nonclustered | nonclustered

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

10,000 records
1,000 pages

clustered | nonclustered

DEPT (did, dname, floor, mgr)

500 records
50 pages

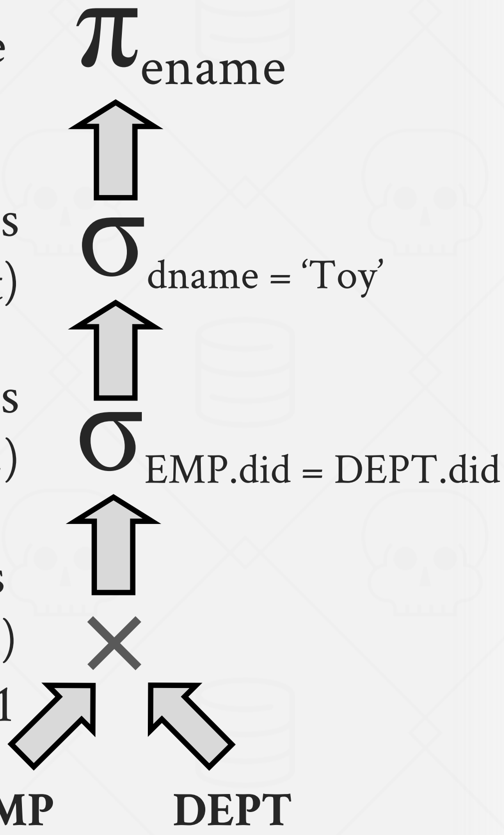
Total: 2M I/Os

4 reads, 1 write

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

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

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



Query

SELECT distinct ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'

Total: 54K I/Os

Catalog

clustered nonclustered nonclustered

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

10,000 records
1,000 pages

clustered nonclustered

DEPT (did, dname, floor, mgr)

500 records
50 pages

Read temp T2
4 reads + 1 writes

π_{ename}

Read temp T1
2,000 reads +4 writes

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

Page NL, write to temp T1
50 + 50,000 + 2000 writes

$\bowtie_{EMP.did = DEPT.did}$

EMP **DEPT**

Query

SELECT distinct ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'

w/ Materialization **Total: 7,159 I/Os**

w/ Pipelining **Total: 3,151 I/Os**

Catalog

clustered nonclustered nonclustered

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

10,000 records
1,000 pages

clustered nonclustered

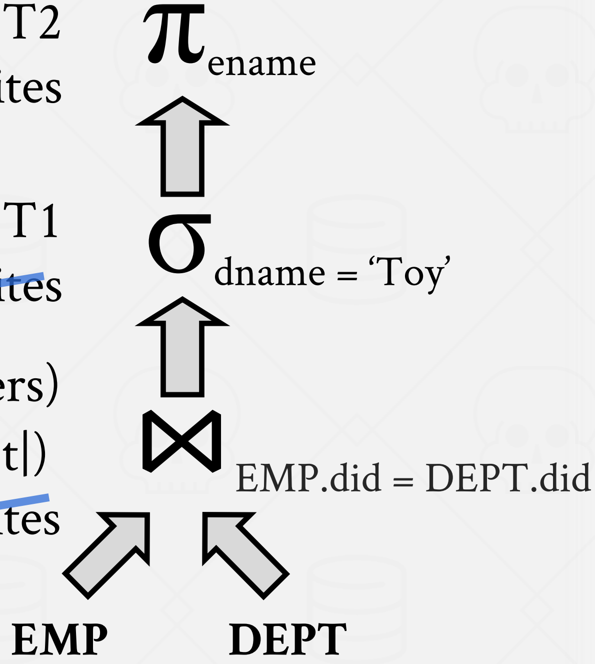
DEPT (did, dname, floor, mgr)

500 records
50 pages

Read temp T2
~~4 reads~~ + 1 writes

Read temp T1
~~2,000 reads~~ +~~4 writes~~

Sort-merge join (50 buffers)
3*(|Emp|+|Dept|)
= 3150 + ~~2000 writes~~



Query

SELECT distinct ename
FROM Emp E, Dept D
WHERE E.did = D.did AND D.dname = 'Toy'

Total: 37 I/Os

Catalog

clustered nonclustered nonclustered

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

10,000 records
1,000 pages

clustered nonclustered

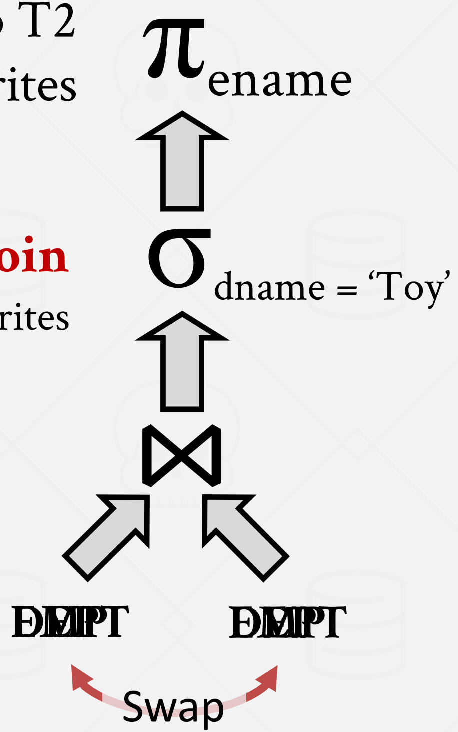
DEPT (did, dname, floor, mgr)

500 records
50 pages

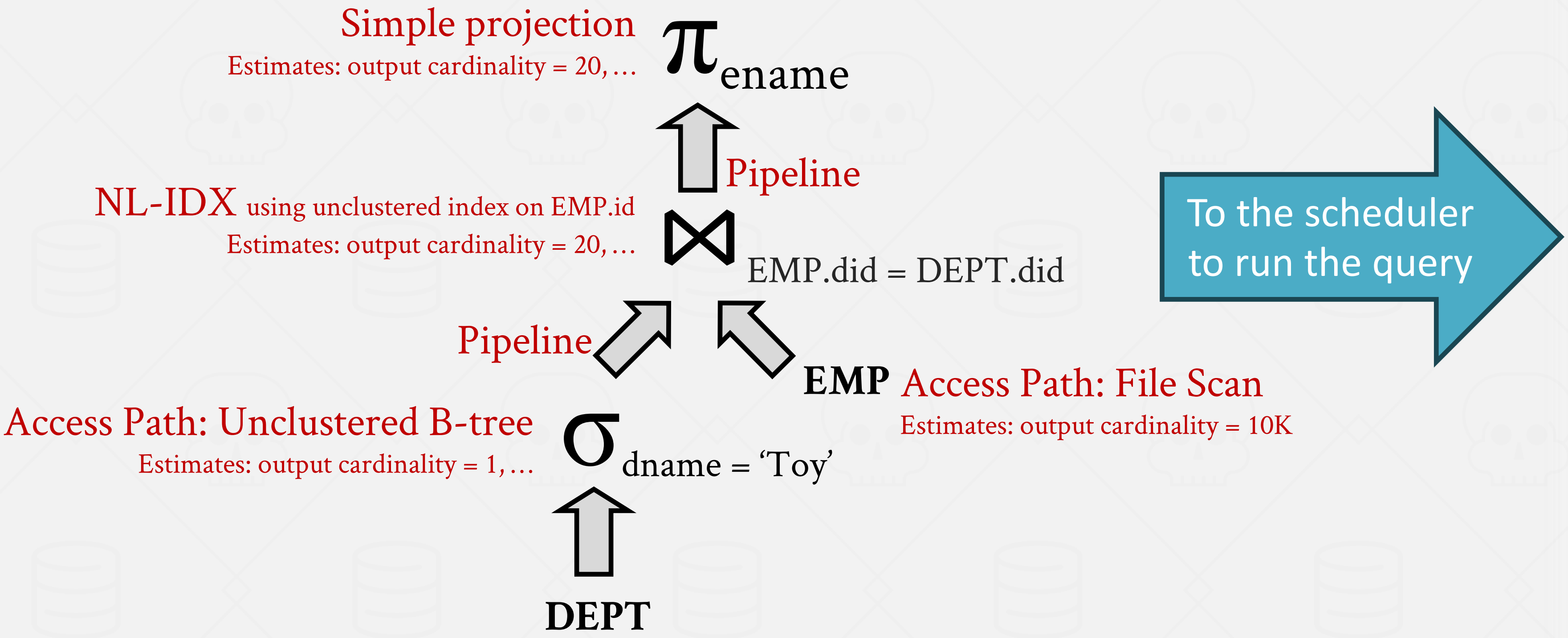
Read temp T2
4 reads + 1 writes

Read temp T1, **NL-IDX Join**
1 + 3 (idx) + 20 (ptr chase) + 4 writes

Access: **Index (name)**
3 reads + 1 write



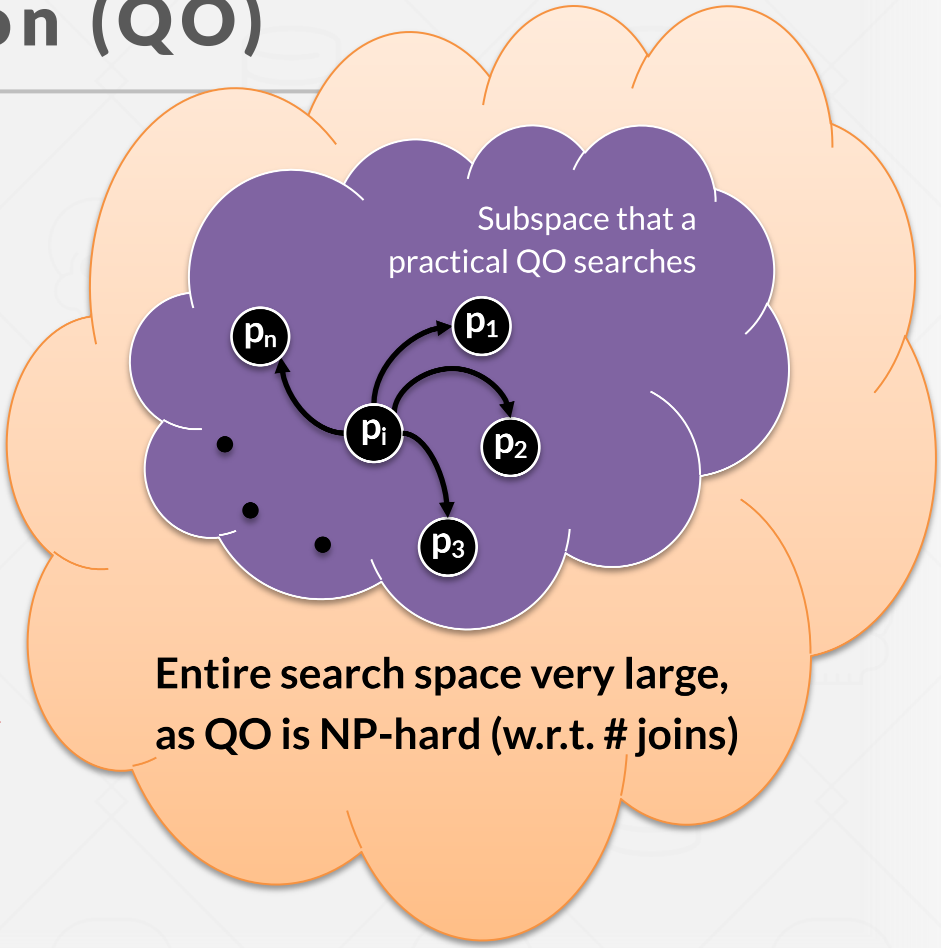
Annotated RA Tree a.k.a. The Physical Plan



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 tree (physical). We need to **estimate** the cost of each plan.
3. Choose the best overall (physical) plan.

Practically: Choose from a subset of all possible plans.



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

Heuristics / Rules

→ Rewrite the query to remove (guessed) inefficiencies; e.g, always do selections first, or push down projections as early as possible.

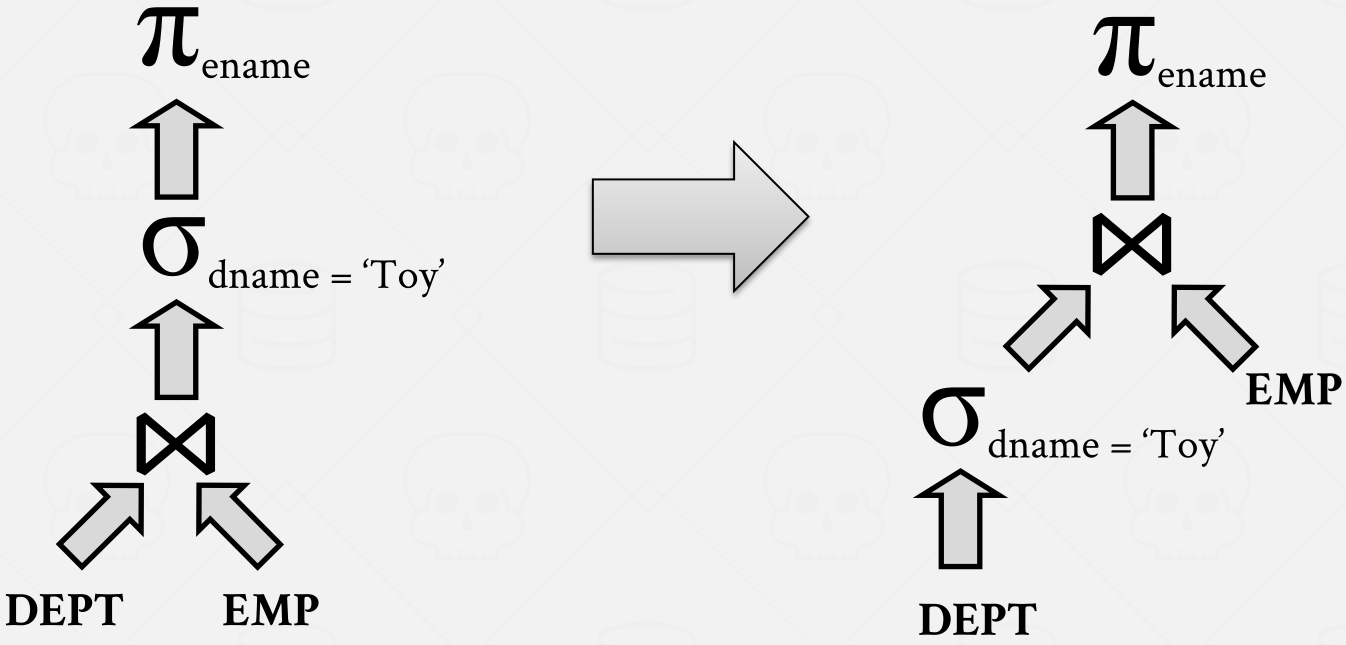
→ These techniques may need to examine catalog, but they do <u>not</u> need to examine data.

Cost-based Search

→ Use a model to estimate the cost of executing a plan.

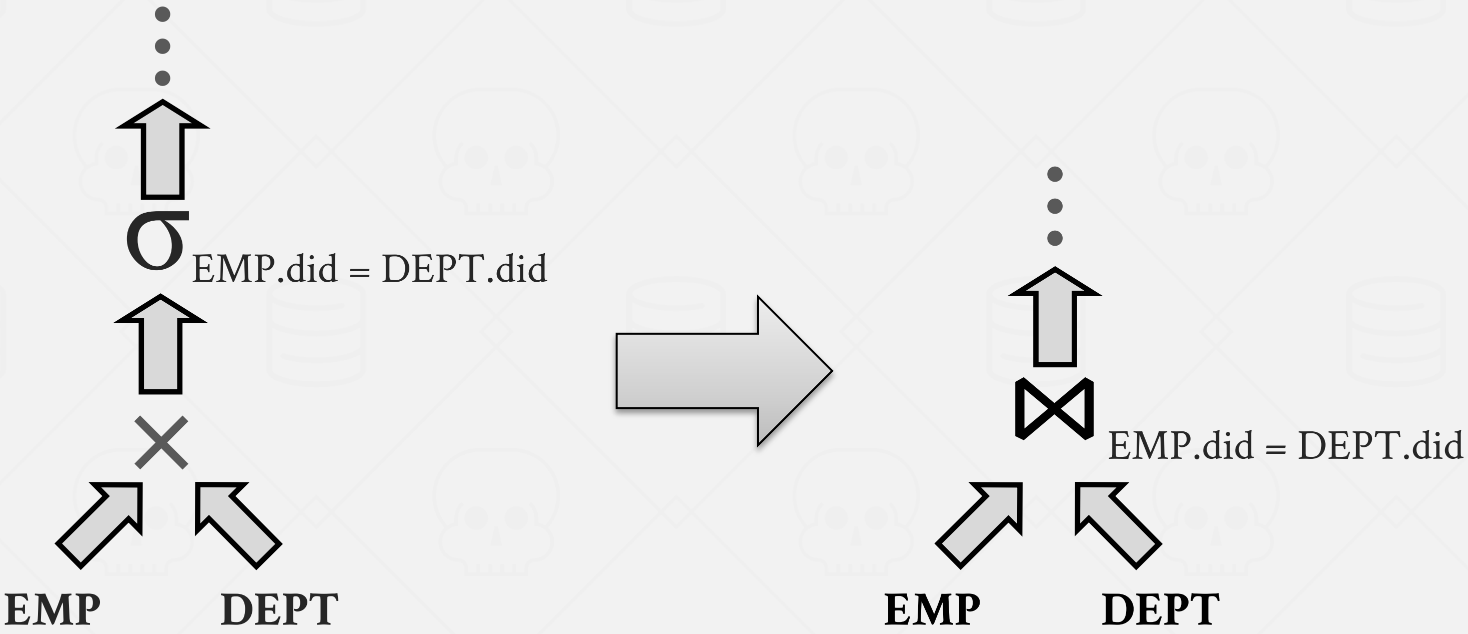
→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

Predicate Pushdown



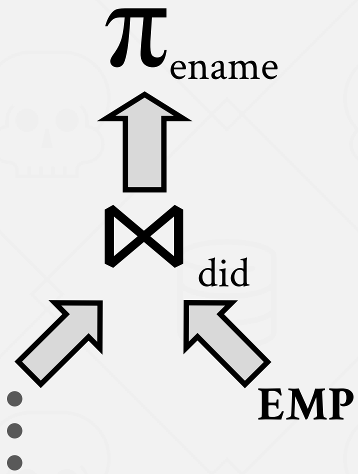
$\pi_{ename}\big(\sigma_{dname = 'Toy'}(DEPT \bowtie EMP)\big)$ $\xrightarrow{\text{rewrite}}$ $\pi_{ename}\big(EMP \bowtie \sigma_{dname = 'Toy'}(DEPT)\big)$

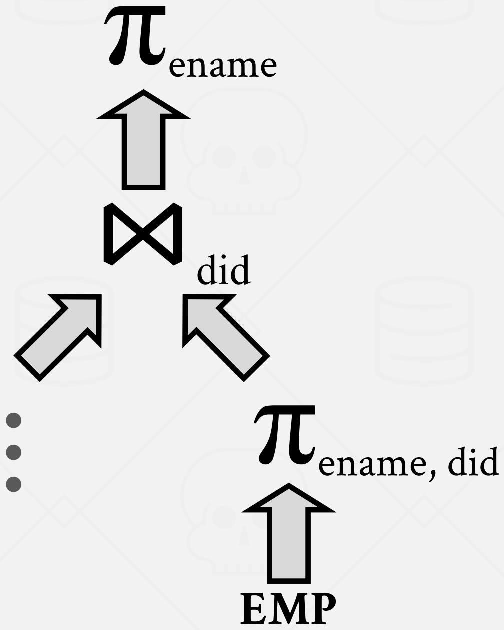
Replace Cartesian Product



$\ldots(\sigma_{DEPT.did = EMP.did}(DEPT \times EMP))$ rewrite $\ldots(EMP \bowtie_{did} DEPT)$

Projection Pushdown





$$\pi_{\text{EMP.ename}}\left(\ldots \bowtie_{\text{did}} \text{EMP}\right) \xrightarrow{\text{rewrite}} \pi_{\text{EMP.ename}}\left(\ldots \bowtie_{\text{did}} (\pi_{\text{ename, did}}\, \text{EMP})\right)$$

Equivalence

$\sigma_{P1}\ (\sigma_{P2}(R)) \equiv \sigma_{P2}\ (\sigma_{P1}(R))$ (σ commutativity)

$\sigma_{P1 \wedge P2\ \ldots\ \wedge Pn}\ (R) \equiv \sigma_{P1}(\sigma_{P2}(\ \ldots\ \sigma_{Pn}(R)))$ (cascading σ)

$\prod_{a1}(R) \equiv \prod_{a1}(\prod_{a2}(\ldots\prod_{ak}\ (R)\ldots))$, $a_i \subseteq a_{i+1}$ (cascading ∏)

$R \bowtie S \equiv S \bowtie R$ (join commutativity)

$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (join associativity)

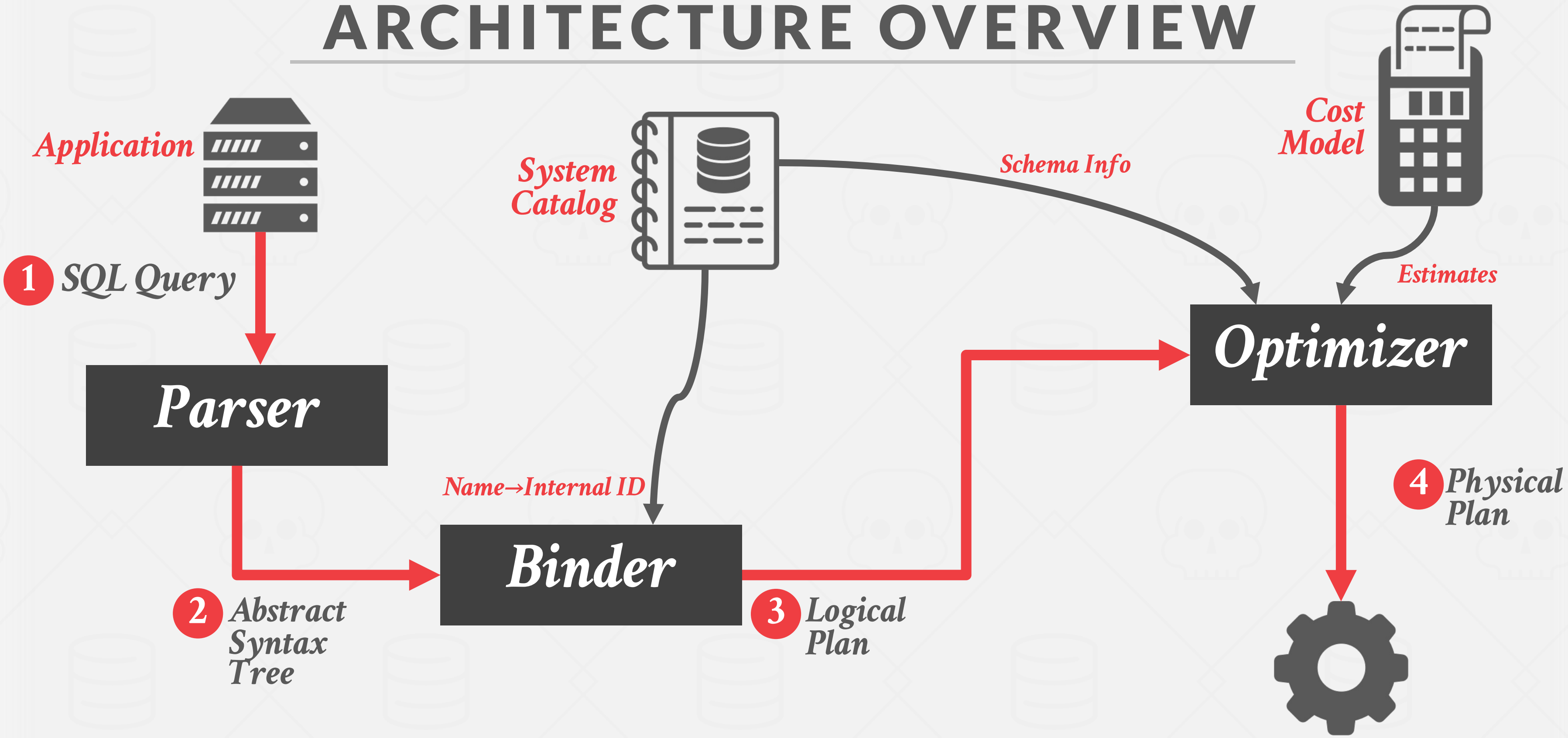
$\sigma_P\ (R \times S) \equiv (R \bowtie_P S)$, if P is a join predicate

$\sigma_P\ (R \times S) \equiv \sigma_{P1}\ (\sigma_{P2}(R) \bowtie_{P4}\ \sigma_{P3}(S))$, where $P = p1 \wedge p2 \wedge p3 \wedge p4$

$\prod_{A1,A2,\ldots An}(\sigma_P\ (R)) \equiv \prod_{A1,A2,\ldots An}(\sigma_P\ (\prod_{A1,\ldots An,\ B1,\ldots\ BM} R))$, where B1 … BM are columns in P

…

ARCHITECTURE OVERVIEW



QUERY OPTIMIZATION

Examples: predicate pushdown, replace cartesian product, projection pushdown …

Heuristics / Rules

→ Rewrite the query to remove inefficient patterns.

→ 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

Let's start with a certain style of QO: cost-based, bottom-up QO
(the classic System-R optimizer approach)

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.

SYSTEM R OPTIMIZER

Selinger

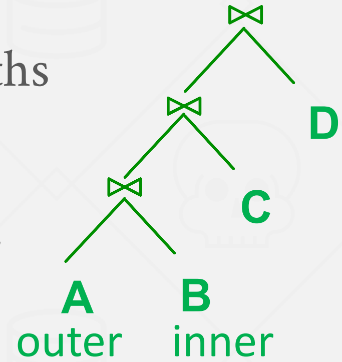
Break the query into blocks and generate the logical operators for each block.

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

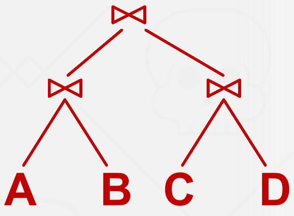
→ All combinations of join algorithms and access paths

Then, iteratively construct a "left-deep" join tree that minimizes the estimated amount of work to execute the plan.

A left-deep tree



A bushy tree



System-R optimizer does NOT consider this "shape"

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

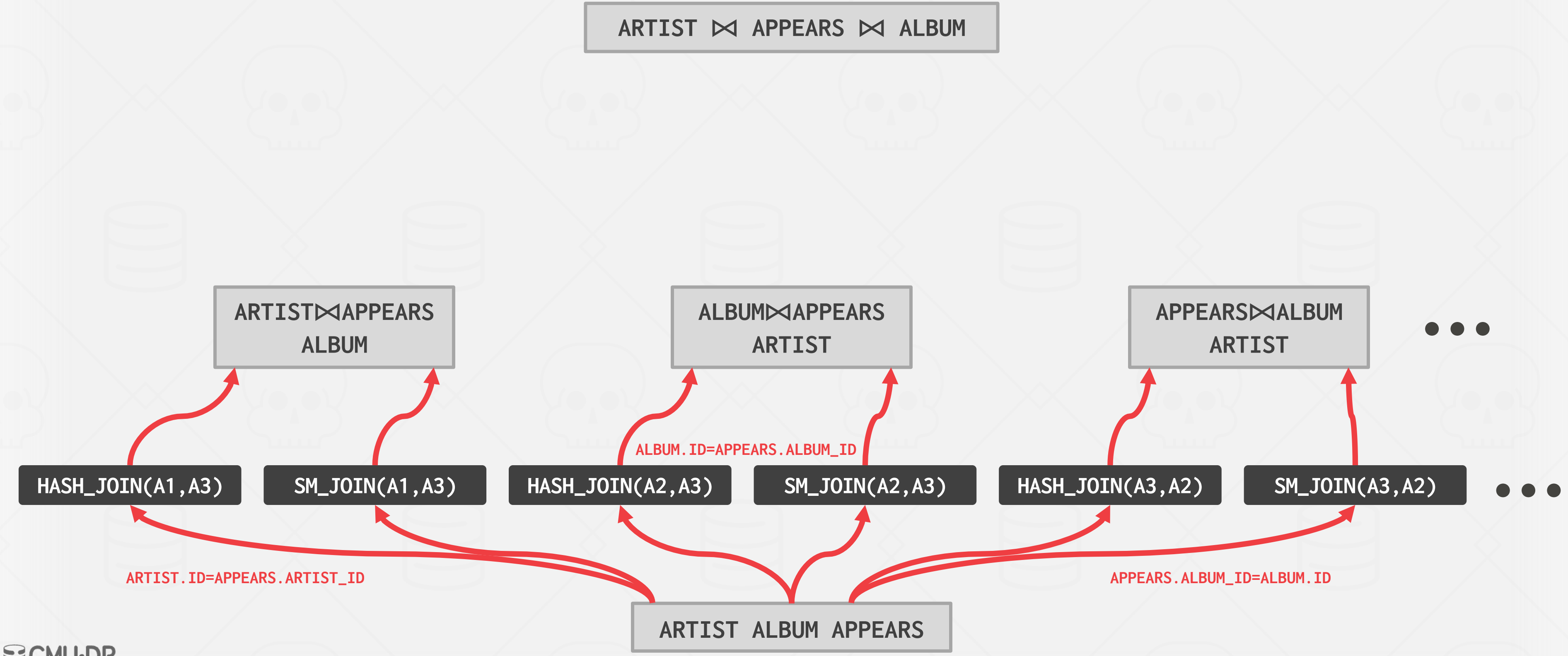
ARTIST ⋈ APPEARS ⋈ ALBUM
APPEARS ⋈ ALBUM ⋈ ARTIST
ALBUM ⋈ APPEARS ⋈ ARTIST
APPEARS ⋈ ARTIST ⋈ ALBUM
ARTIST × ALBUM ⋈ APPEARS
ALBUM × ARTIST ⋈ APPEARS
⋮ ⋮ ⋮

SYSTEM R OPTIMIZER

ARTIST ⋈ APPEARS ⋈ ALBUM

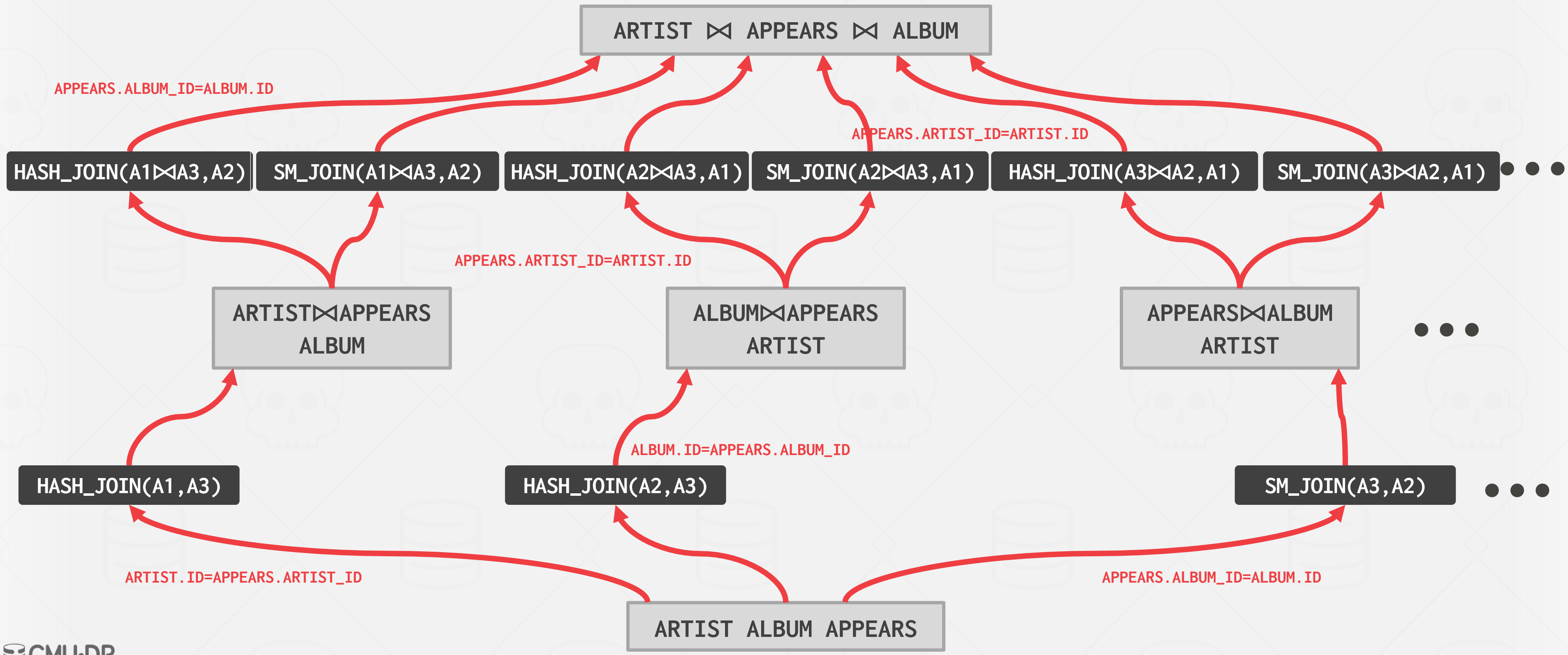
ARTIST ALBUM APPEARS

SYSTEM R OPTIMIZER

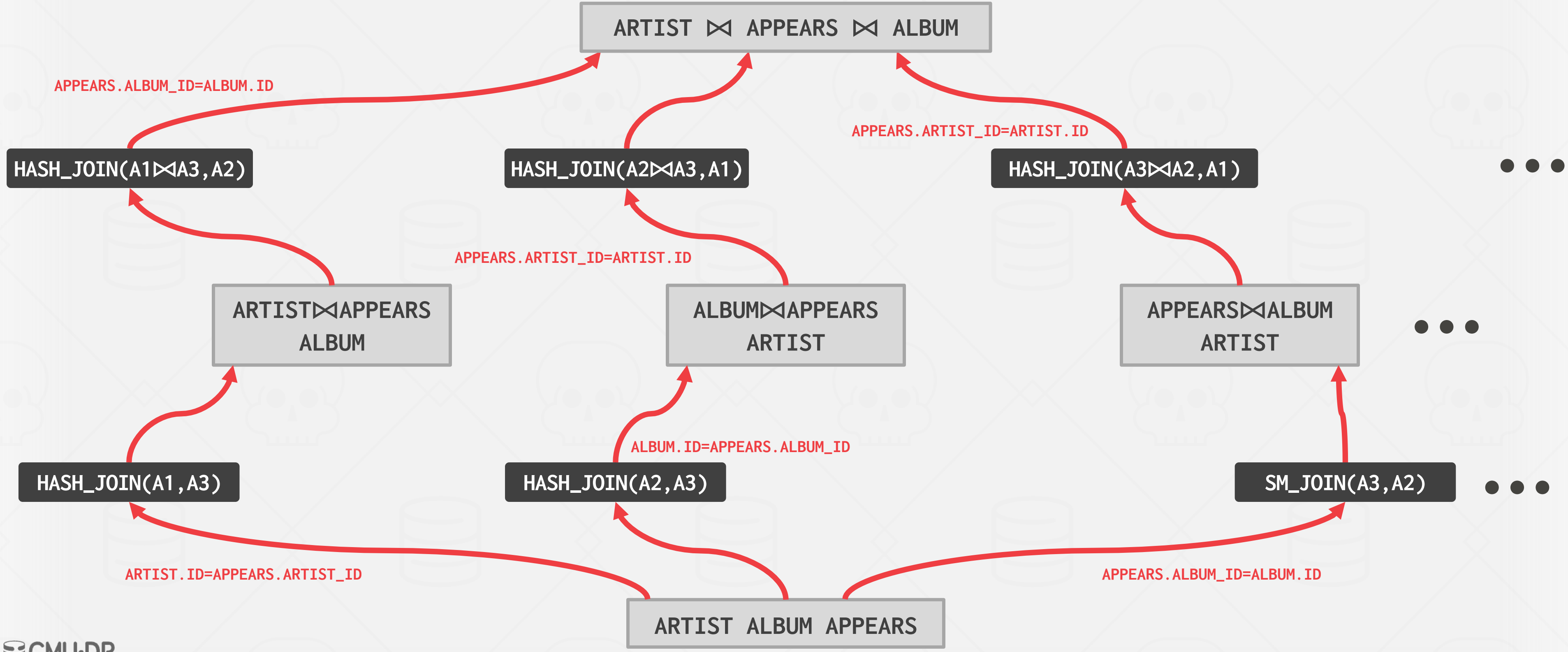


SYSTEM R OPTIMIZER

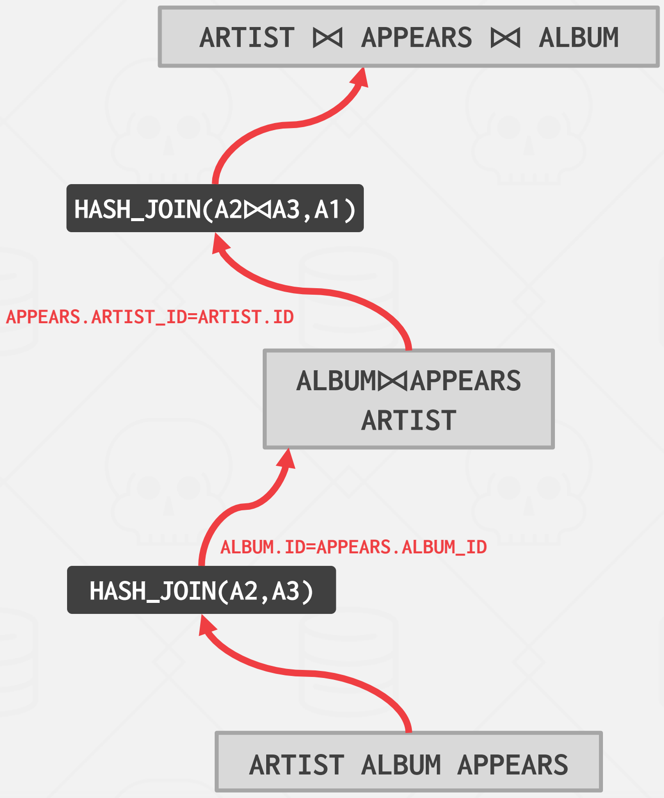
SYSTEM R OPTIMIZER



SYSTEM R OPTIMIZER

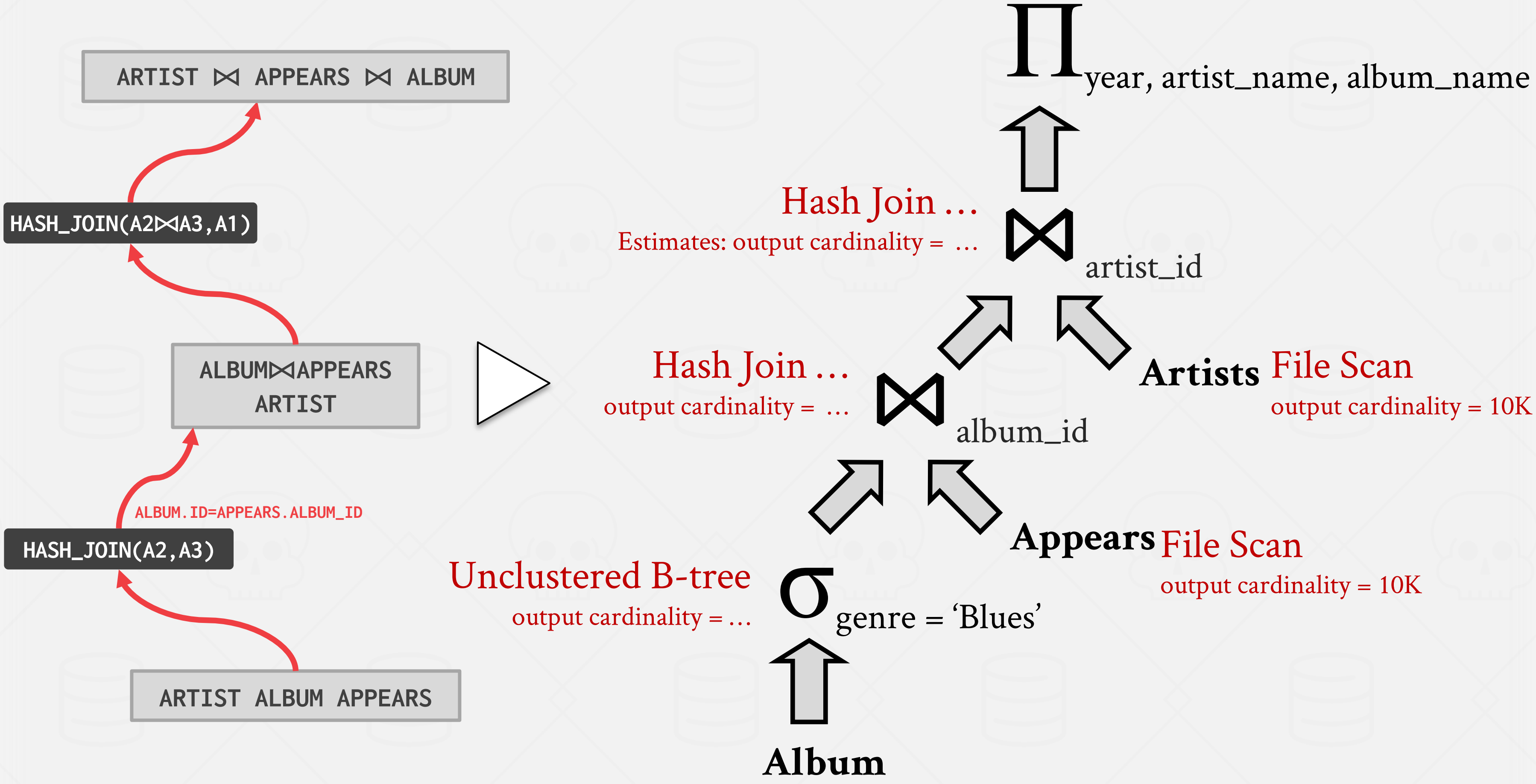


SYSTEM R OPTIMIZER



SYSTEM R OPTIMIZER

The query has **ORDER BY** on **ARTIST.ID** but the plans do not carry an explicit notion of the sorting properties.



ARTIST ⋈ APPEARS ⋈ ALBUM
HASH_JOIN(A2⋈A3,A1)
ALBUM⋈APPEARS
ARTIST
ALBUM.ID=APPEARS.ALBUM_ID
HASH_JOIN(A2,A3)
ARTIST ALBUM APPEARS
Π year, artist_name, album_name
Hash Join …
Estimates: output cardinality = …
⋈ artist_id
Hash Join …
output cardinality = …
⋈ album_id
Artists
File Scan
output cardinality = 10K
Appears
File Scan
output cardinality = 10K
Unclustered B-tree
output cardinality = …
σ genre = ‘Blues’
Album

MULTI-RELATION QUERY PLANNING

We just saw an example of this, the System R approach

Choice #1: Bottom-up Optimization

→ Start with nothing and then build up the plan to get to the outcome that you want.

Choice #2: Top-down Optimization

→ Start with the outcome that you want, and then work down the tree to find the optimal plan that gets you to that goal.

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.

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.

Graefe

Example: MSSQL, Greenplum, CockroachDB

TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

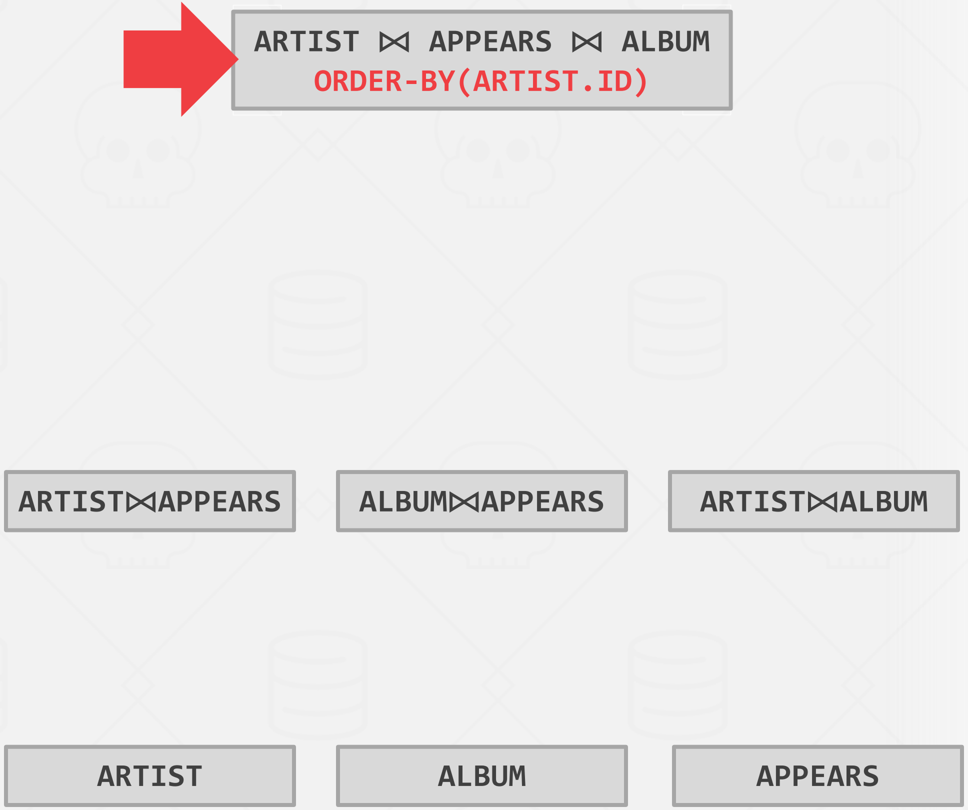
- → **Logical→Logical:**
  JOIN(A,B) to JOIN(B,A)
- → **Logical→Physical:**
  JOIN(A,B) to HASH_JOIN(A,B)

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

TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

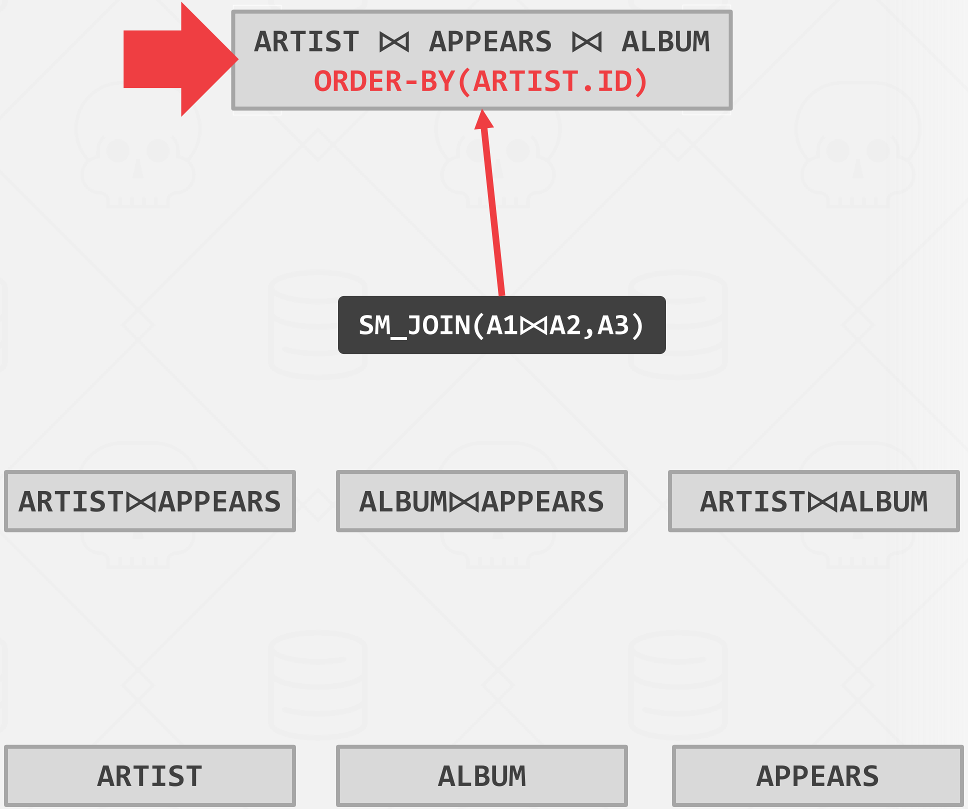
→ **Logical→Logical:**
  JOIN(A,B) to JOIN(B,A)
→ **Logical→Physical:**
  JOIN(A,B) to HASH_JOIN(A,B)



TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

- → **Logical→Logical:**
  JOIN(A,B) to JOIN(B,A)
- → **Logical→Physical:**
  JOIN(A,B) to HASH_JOIN(A,B)

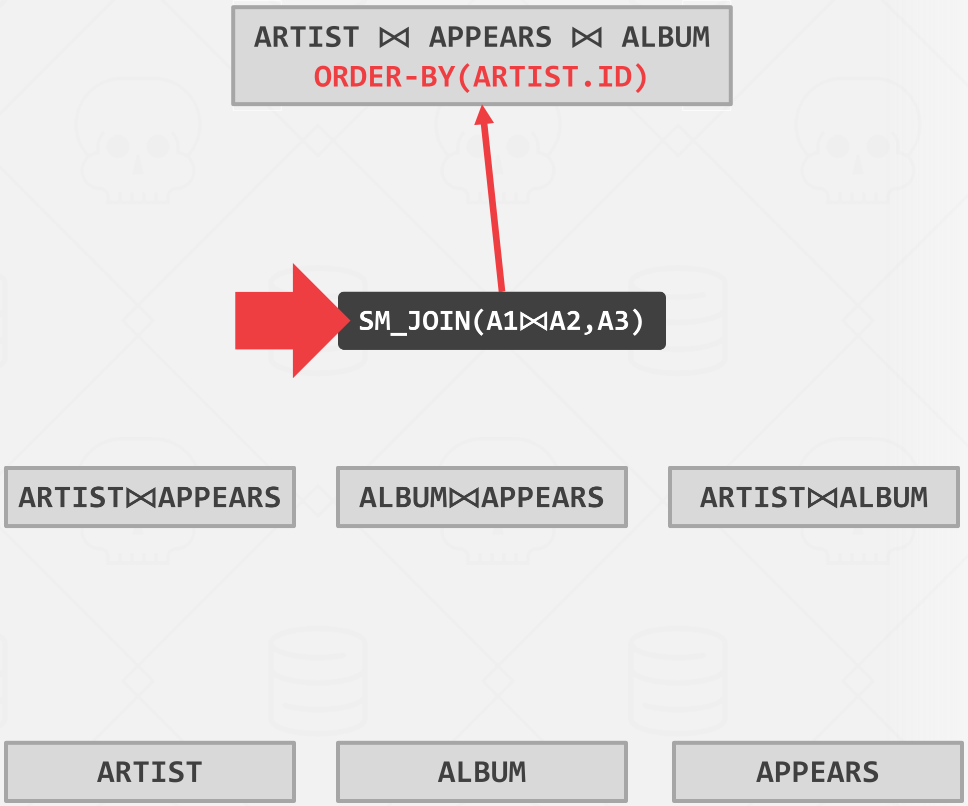


TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ **Logical→Logical:**
  JOIN(A,B) to JOIN(B,A)

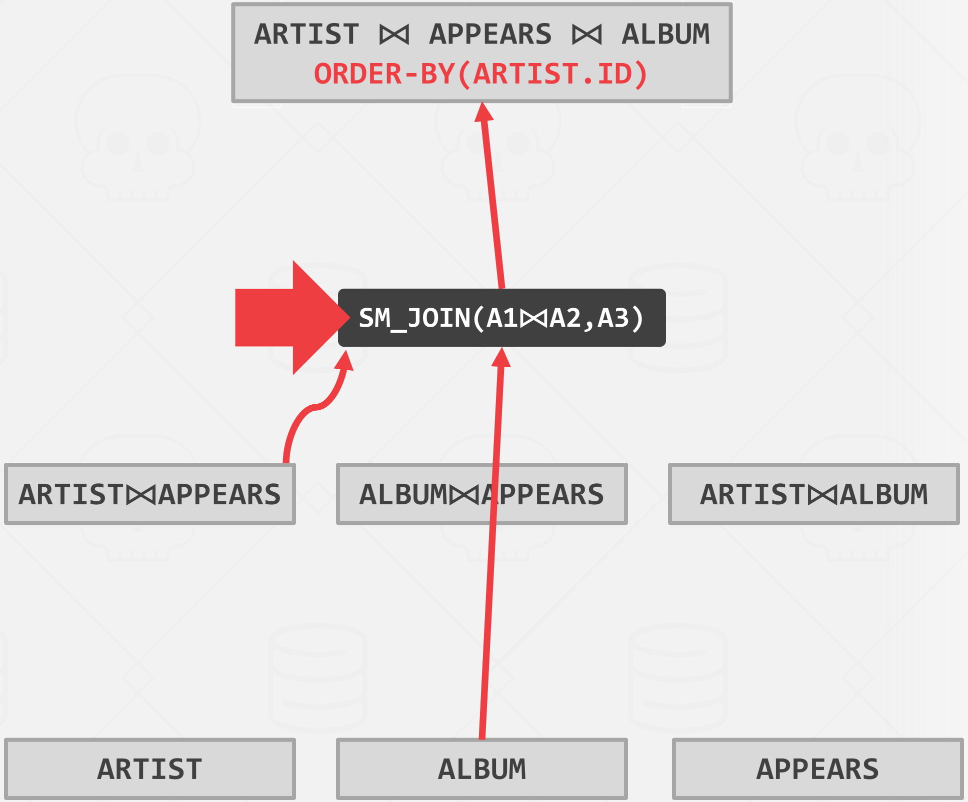
→ **Logical→Physical:**
  JOIN(A,B) to HASH_JOIN(A,B)



TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

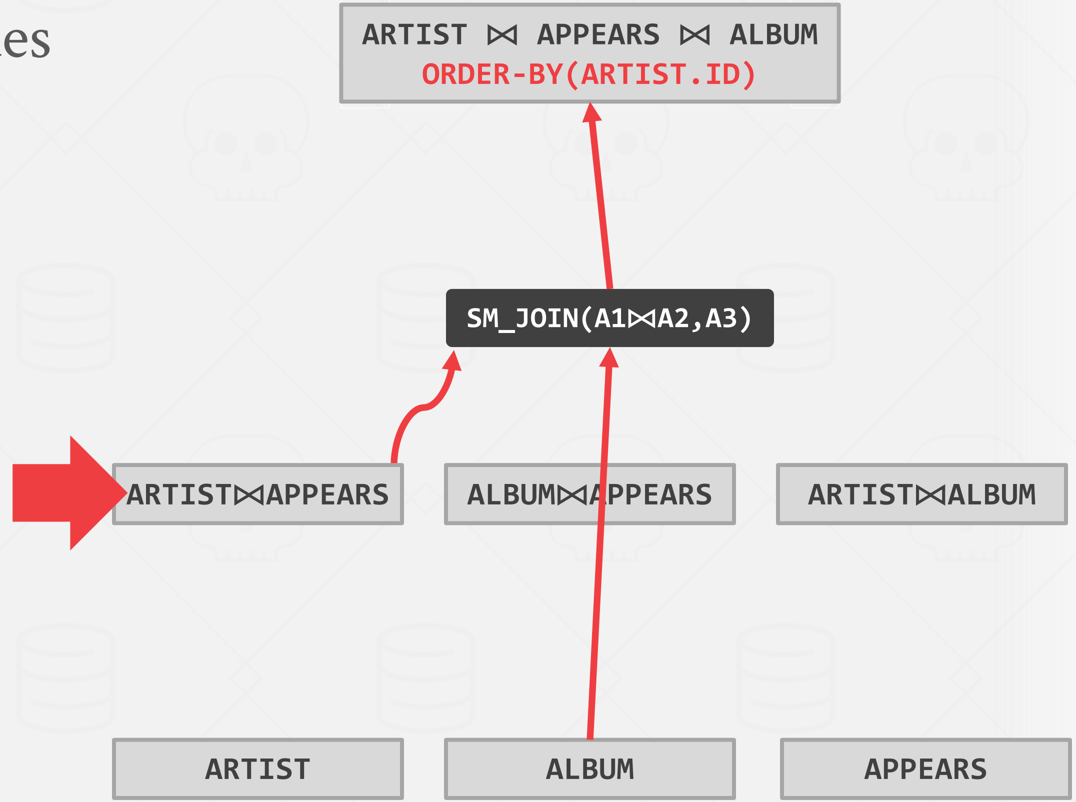
- → **Logical→Logical:**
  JOIN(A,B) to JOIN(B,A)
- → **Logical→Physical:**
  JOIN(A,B) to HASH_JOIN(A,B)



TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

- → **Logical→Logical:**
  JOIN(A,B) to JOIN(B,A)
- → **Logical→Physical:**
  JOIN(A,B) to HASH_JOIN(A,B)

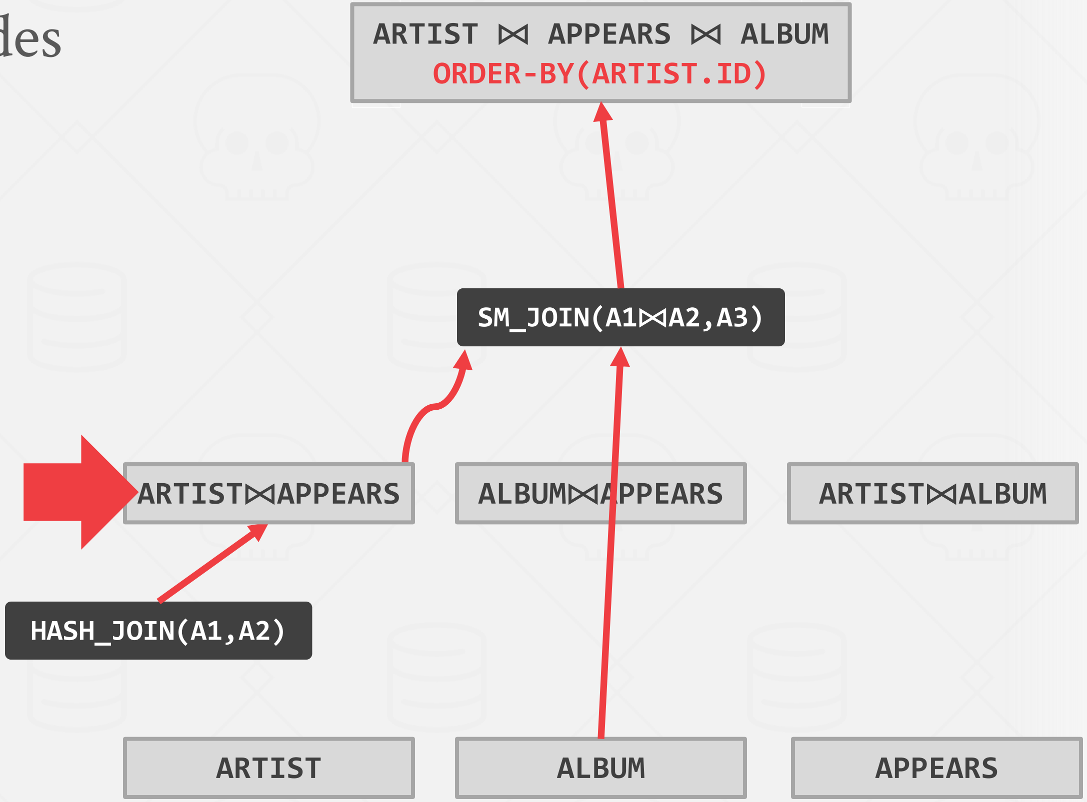


TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ **Logical→Logical:**
JOIN(A,B) to JOIN(B,A)

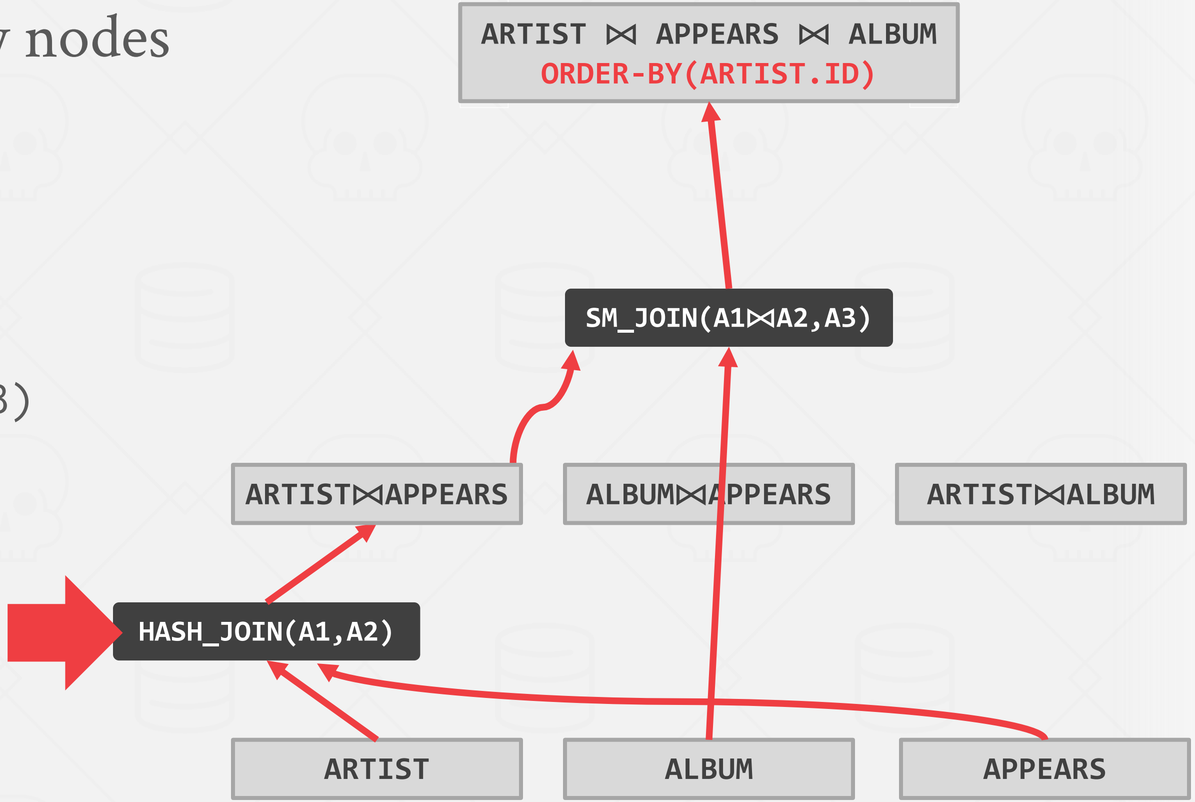
→ **Logical→Physical:**
JOIN(A,B) to HASH_JOIN(A,B)



TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

- → **Logical→Logical:**
  JOIN(A,B) to JOIN(B,A)
- → **Logical→Physical:**
  JOIN(A,B) to HASH_JOIN(A,B)

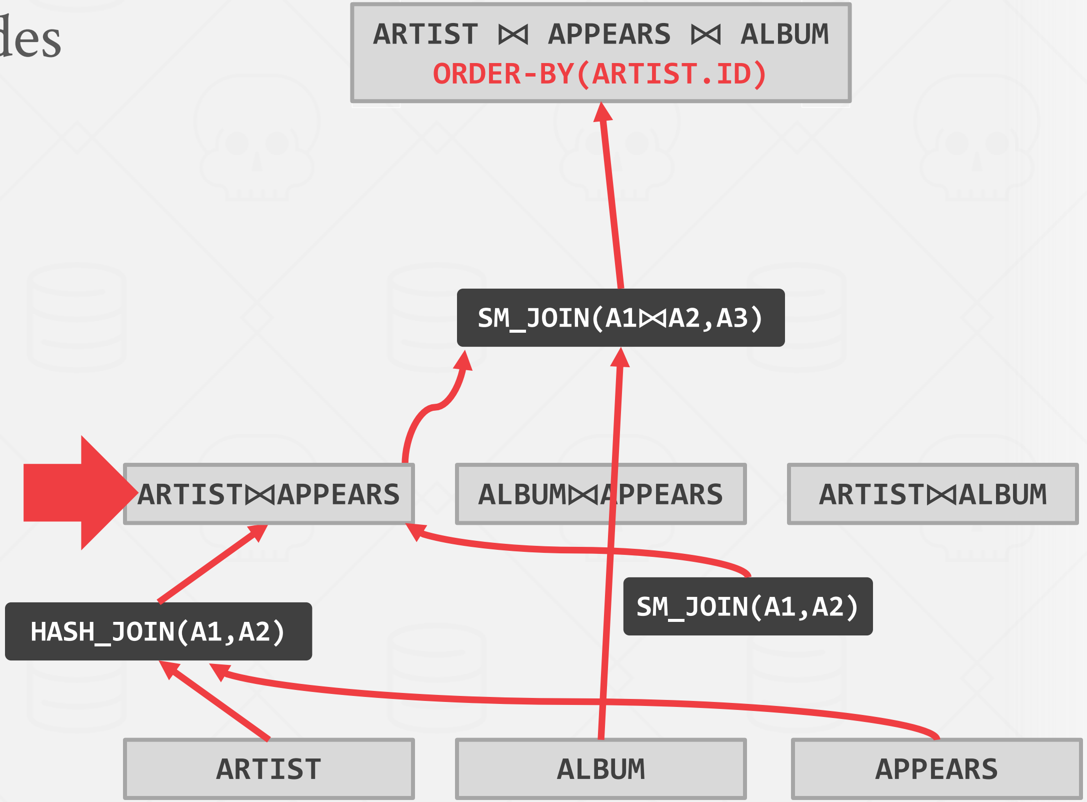


TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ **Logical→Logical:**
JOIN(A,B) to JOIN(B,A)

→ **Logical→Physical:**
JOIN(A,B) to HASH_JOIN(A,B)

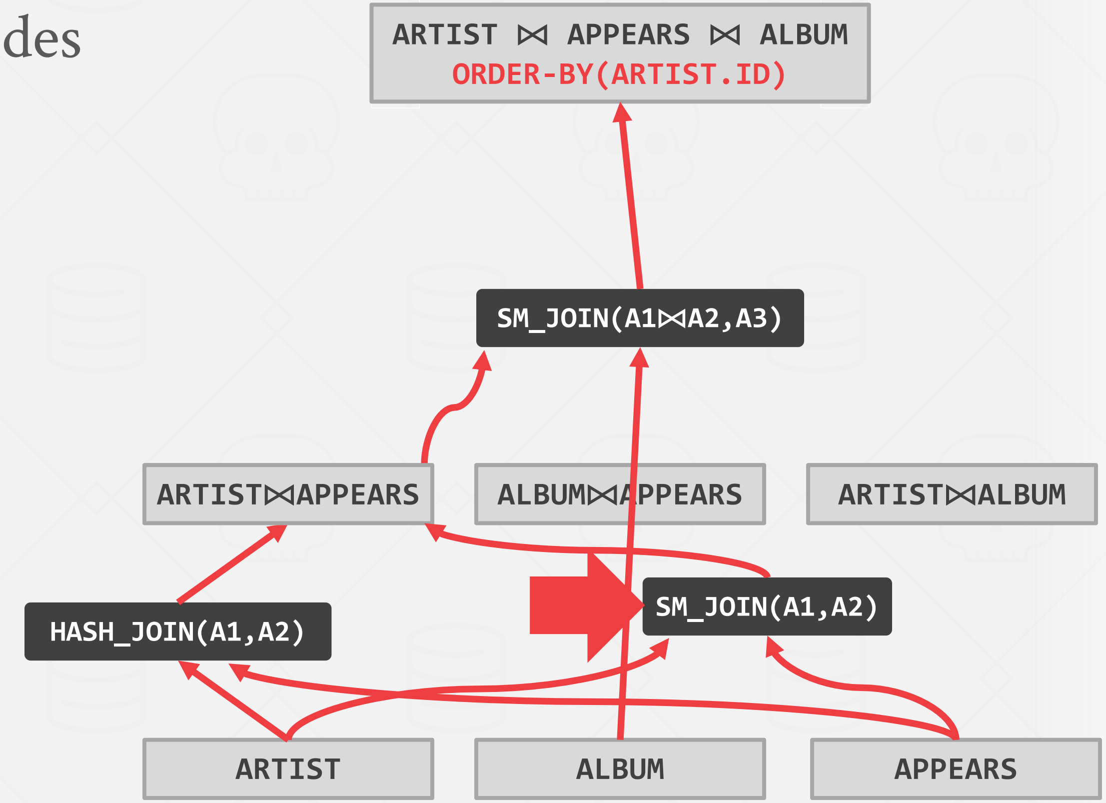


TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the tree.

→ **Logical→Logical:**
  JOIN(A,B) to JOIN(B,A)

→ **Logical→Physical:**
  JOIN(A,B) to HASH_JOIN(A,B)



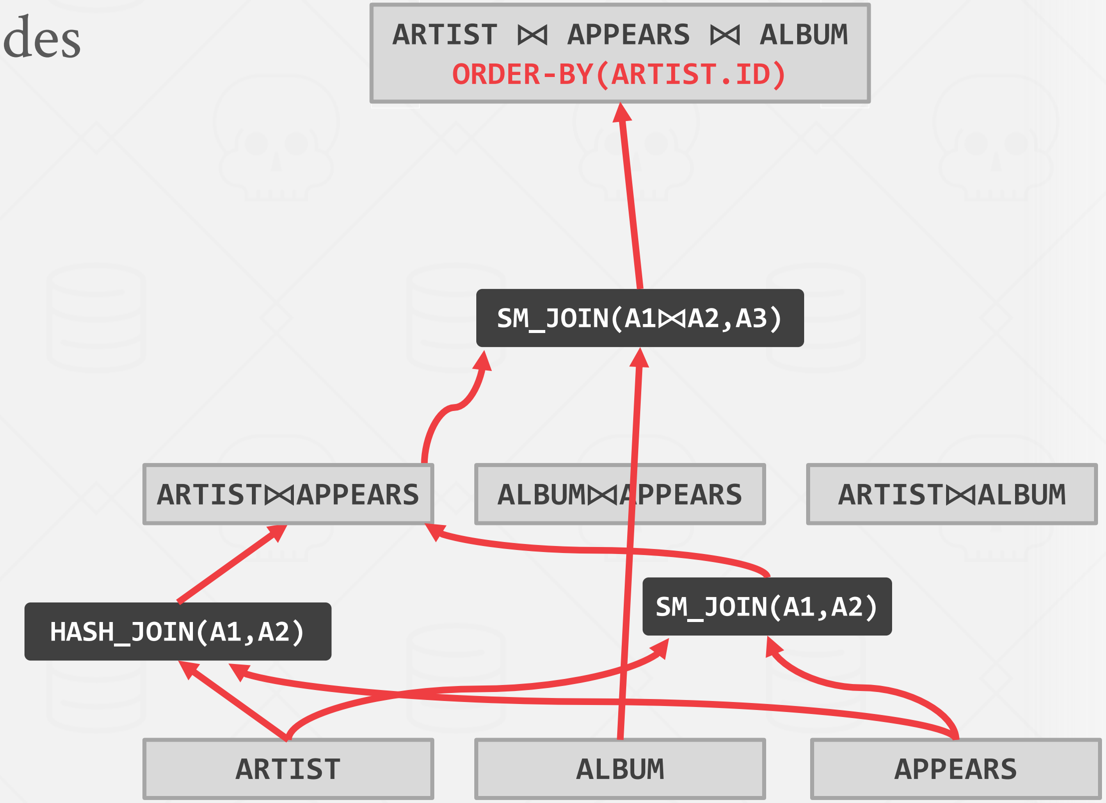
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the 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.



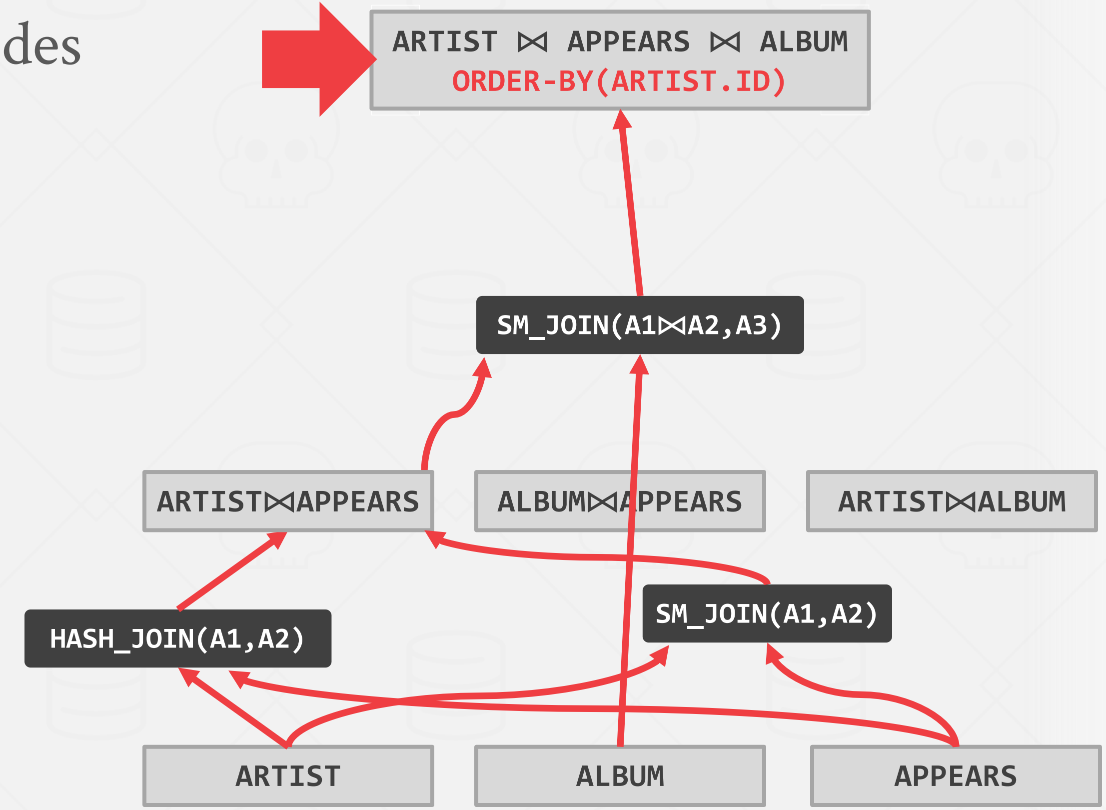
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the 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.



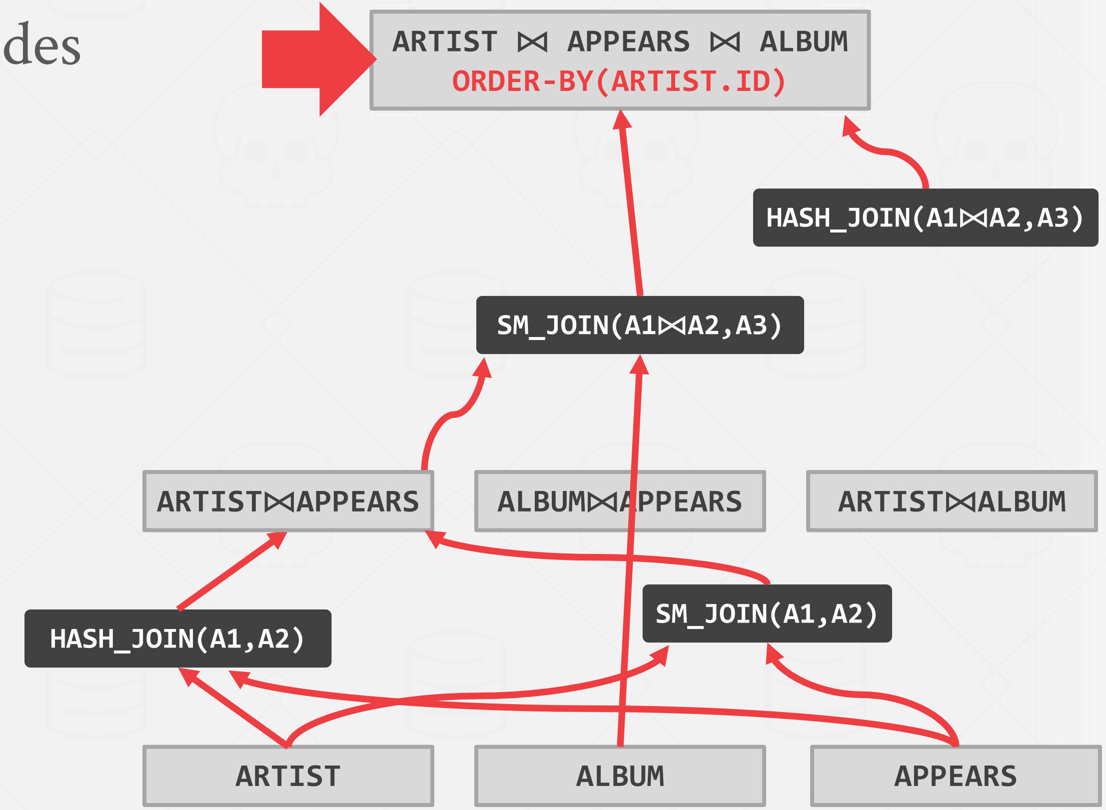
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the 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.

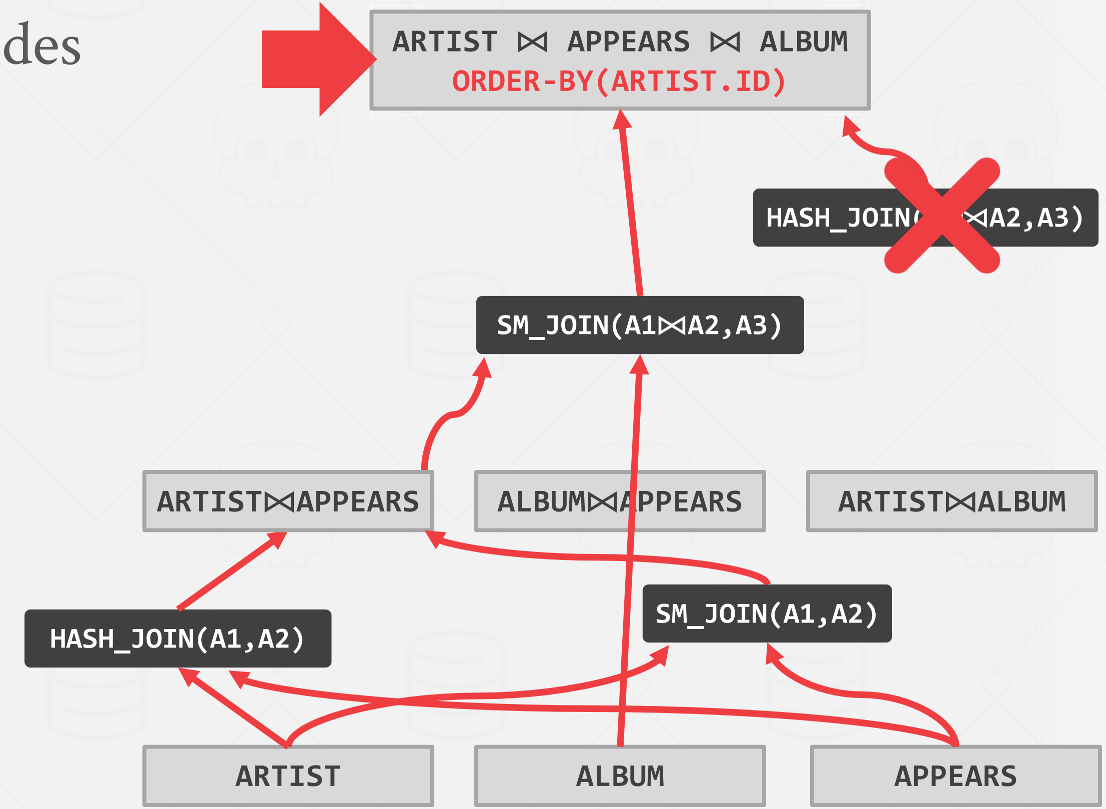


TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the 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.

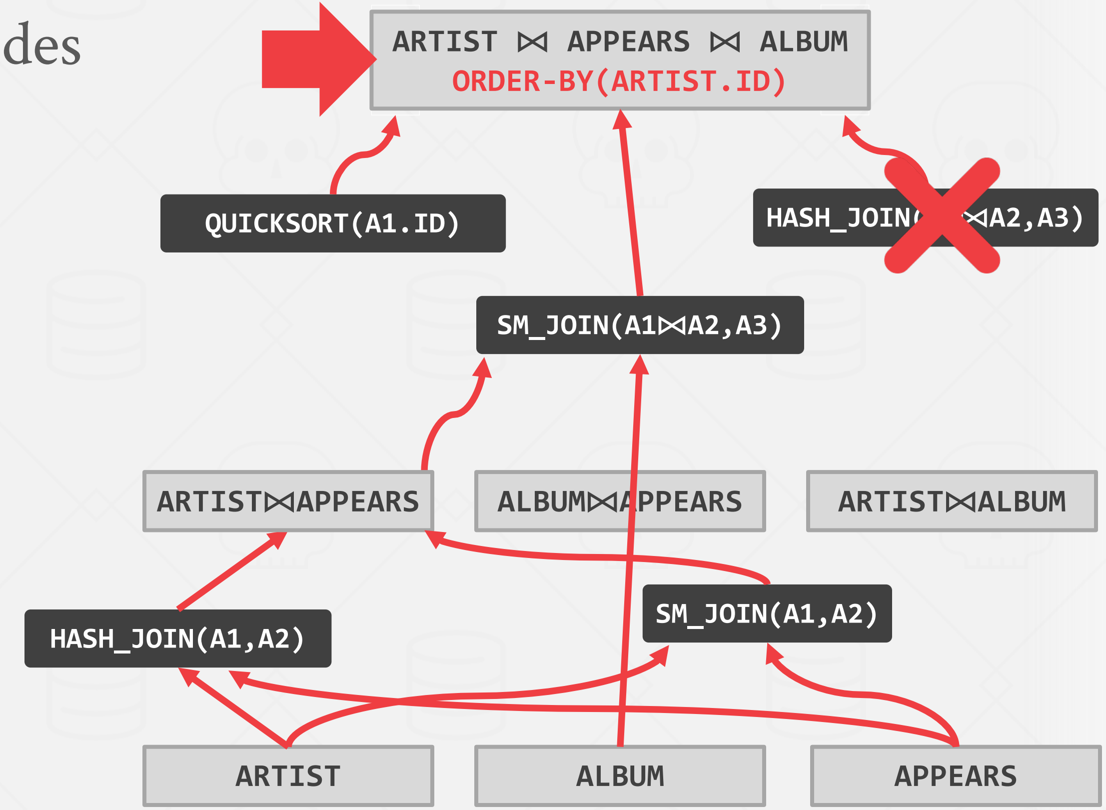


TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the 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.



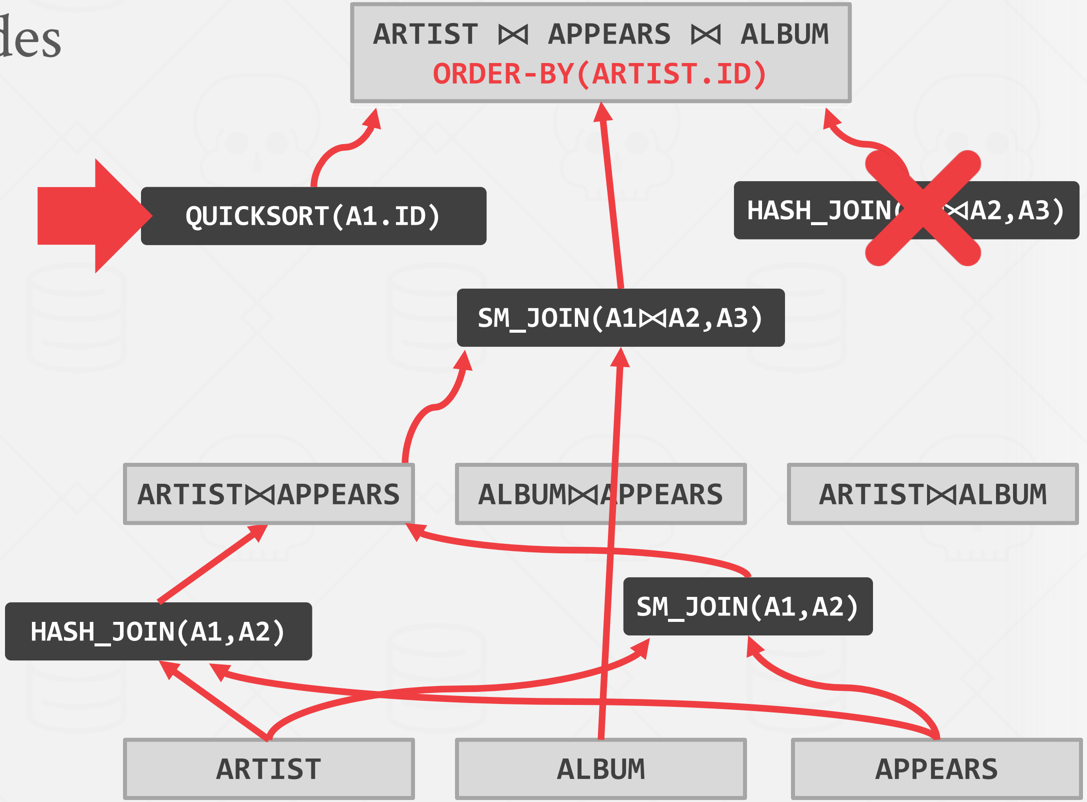
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the 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.



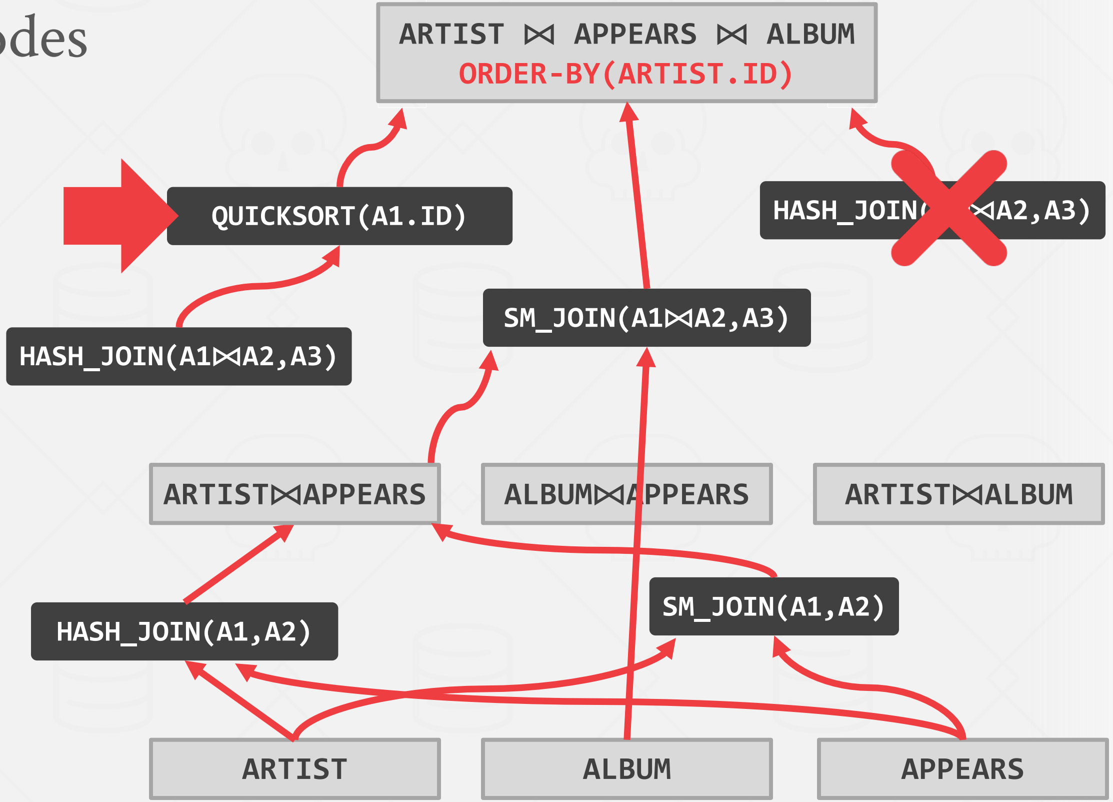
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the 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.



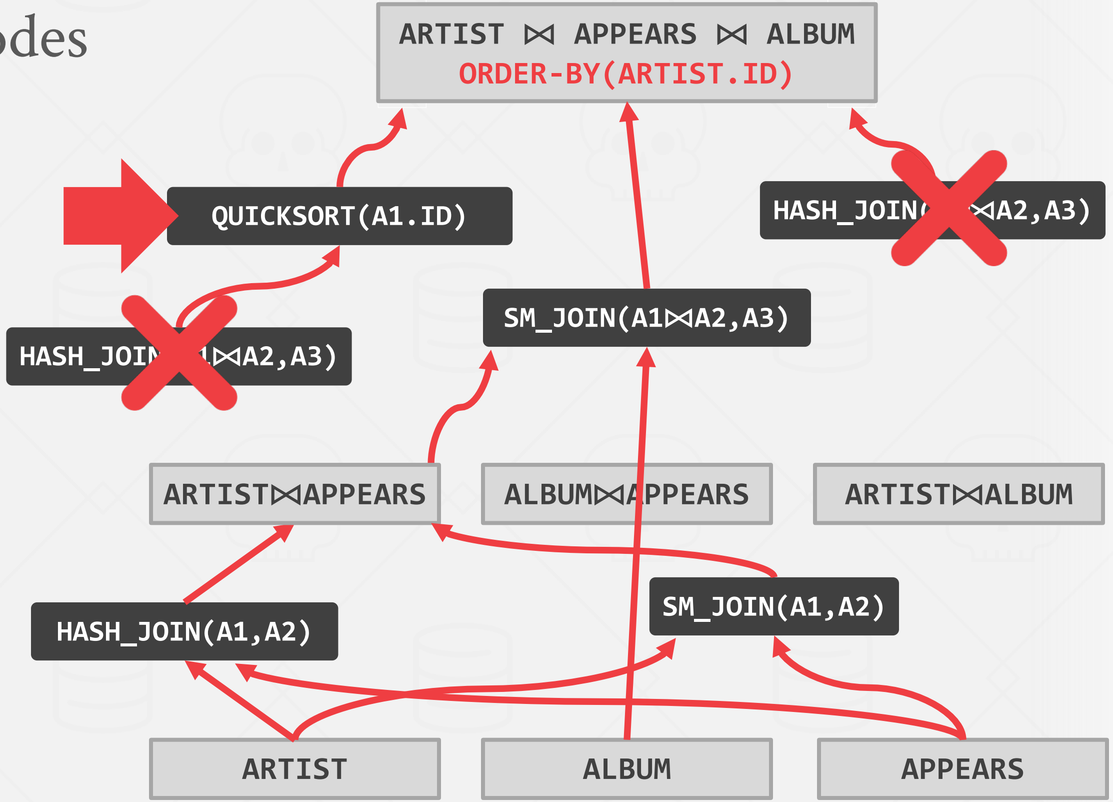
TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse the 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.



Life so far … single block QO

Often, we get 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 can't flatten to a single block, flattening to <u>fewer</u> 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.

Two Approaches:

→ Rewrite to de-correlate and/or flatten them.

→ 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'
 )
```

```
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;
```

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

EXPRESSION REWRITING

Impossible / Unnecessary Predicates

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

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

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

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;
```

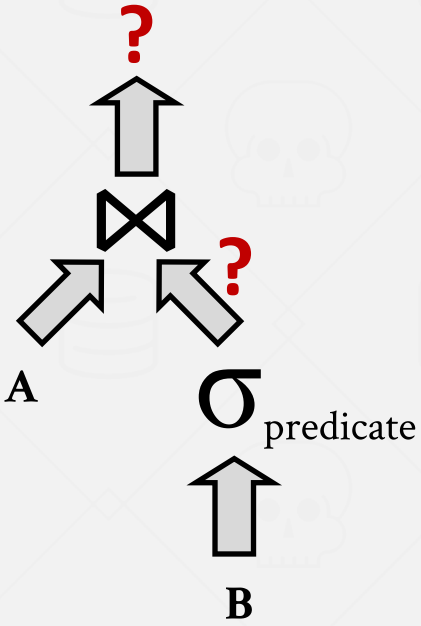
```
SELECT * FROM A WHERE RANDOM() IS NULL;
```

Merging Predicates

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

How do we calculate the cost of the plans?

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



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.

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`

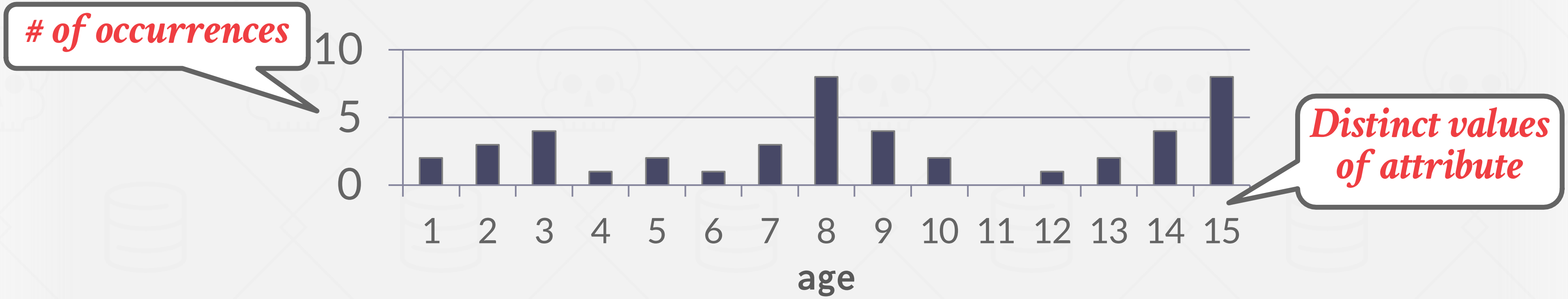
SELECTION CARDINALITY

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

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

Equality Predicate: A=constant

→ sel(A=constant) = #occurences/|R|



SELECTION CARDINALITY

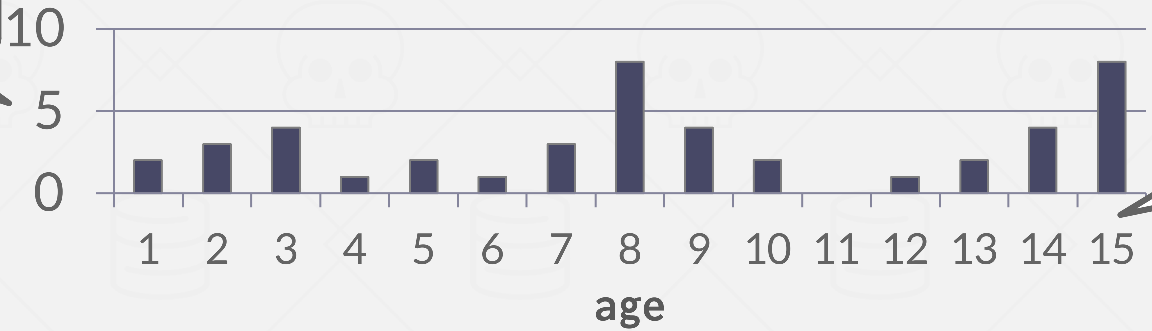
The **selectivity** (`sel`) of a predicate `P` is the fraction of tuples that qualify.

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

Equality Predicate: `A=constant`

- → `sel(A=constant) = #occurences/|R|`
- → Example: `sel(age=9) =`

# of occurrences



Distinct values of attribute

SELECTION CARDINALITY

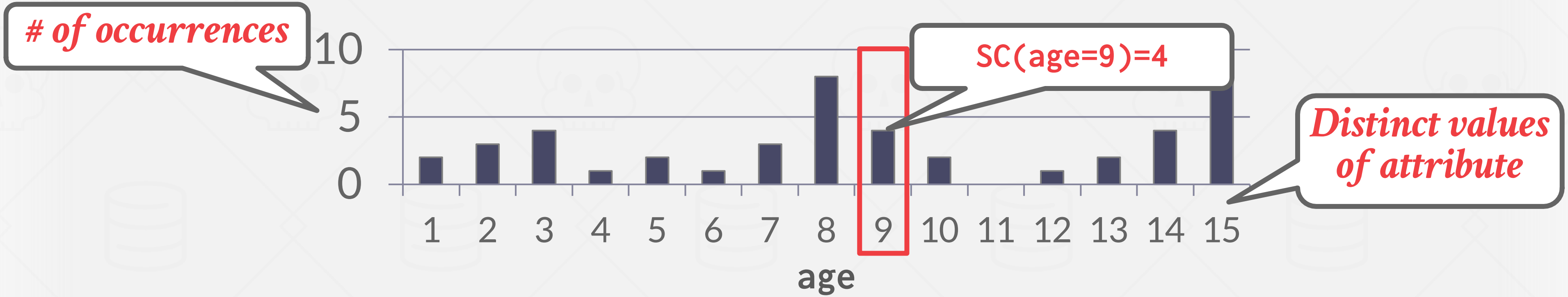
The **selectivity** (**sel**) of a predicate **P** is the fraction of tuples that qualify.

Equality Predicate: `A=constant`

→ `sel(A=constant) = #occurences/|R|`

→ Example: `sel(age=9) =`

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



SELECTION CARDINALITY

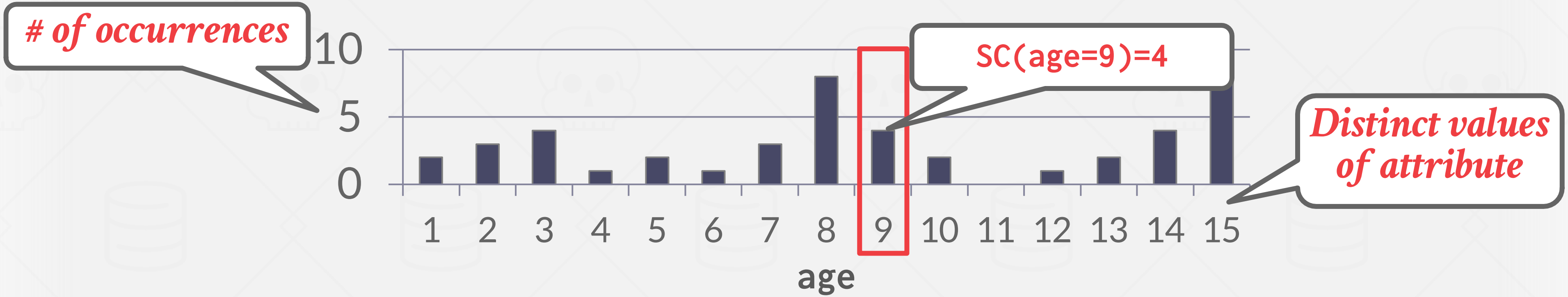
The **selectivity** (sel) of a predicate P is the fraction of tuples that qualify.

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

Equality Predicate: A=constant

→ sel(A=constant) = #occurences/|R|

→ Example: sel(age=9) = 4/45



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

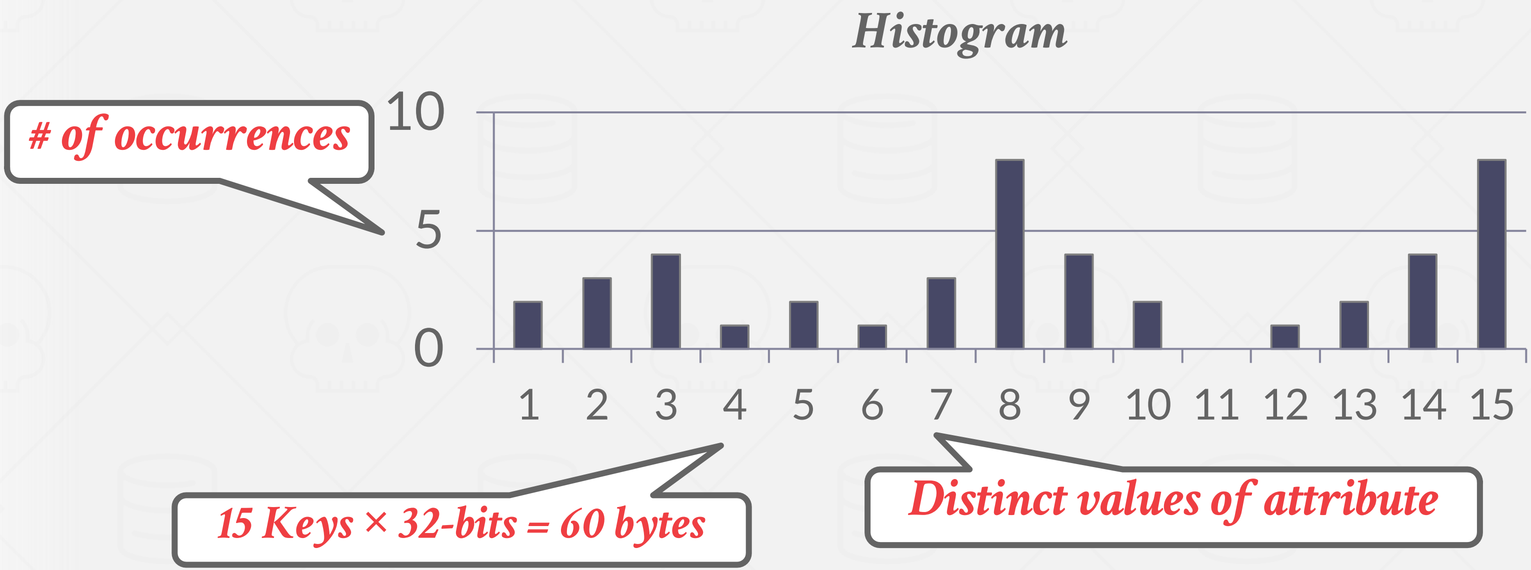
→ Probabilistic data structure that gives an approximate count for a given value.

Choice #3: Sampling

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

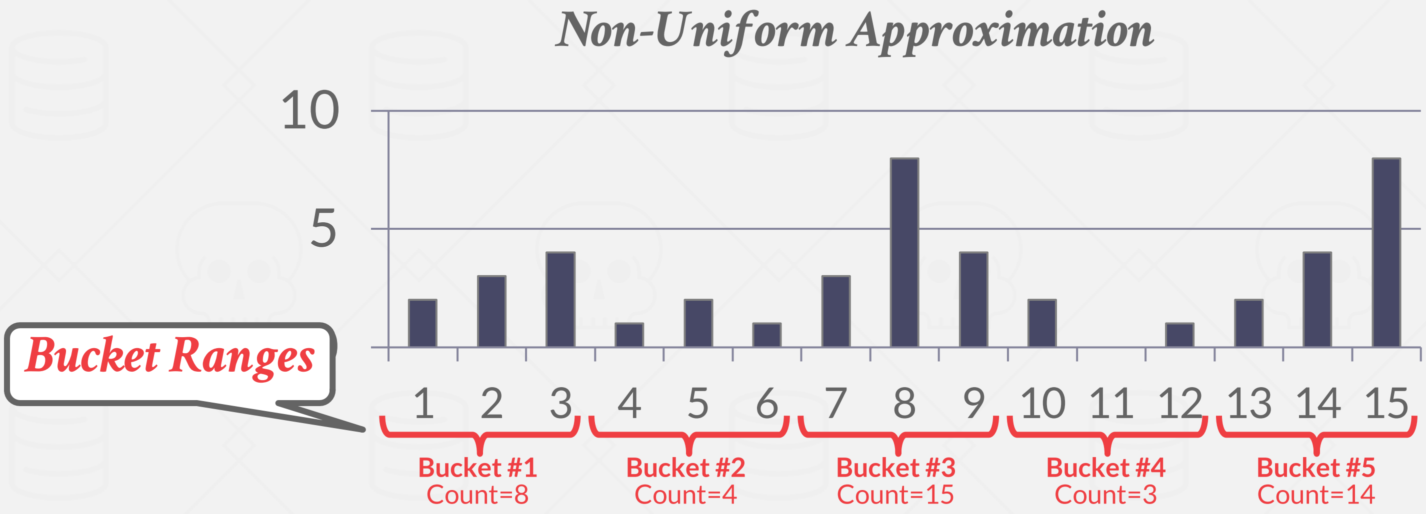
HISTOGRAMS

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



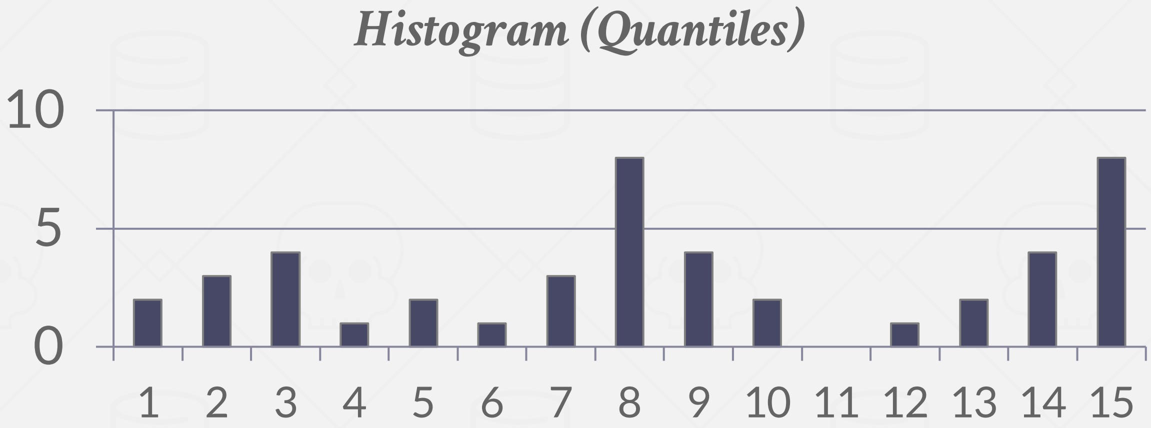
EQUI-WIDTH HISTOGRAM

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



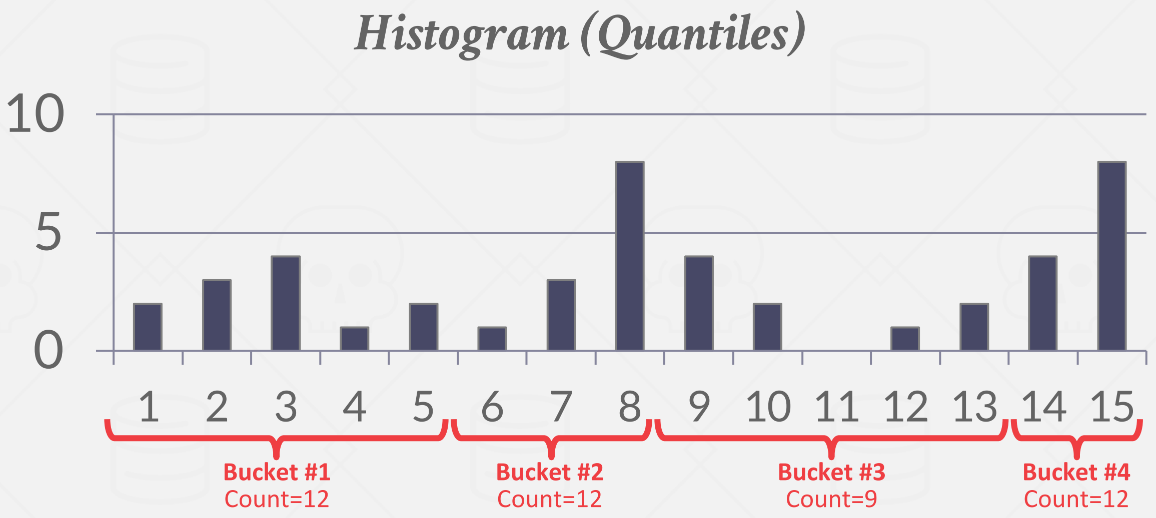
EQUI-DEPTH HISTOGRAMS

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



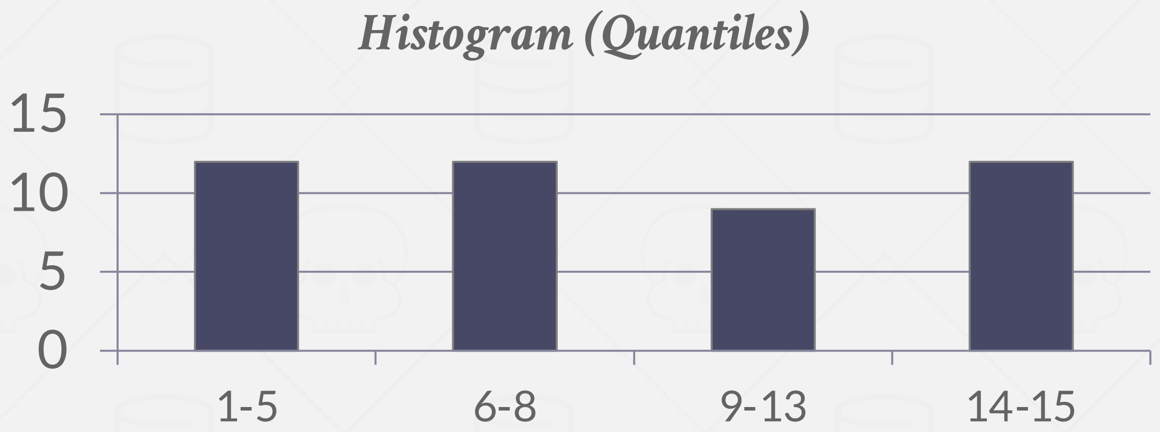
EQUI-DEPTH HISTOGRAMS

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 (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	61	Rested
1002	Kanye	45	Weird
1003	Tupac	25	Dead
1004	Bieber	28	Crunk
1005	Andy	41	Illin
1006	TigerKing	59	Jailed

⋮

1 billion tuples

Table Sample

1001	Obama	61	Rested
1003	Tupac	25	Dead
1005	Andy	41	Illin

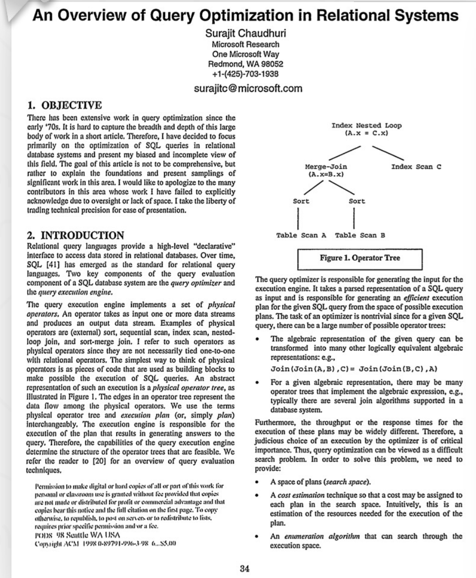
sel(age>50) = 1/3

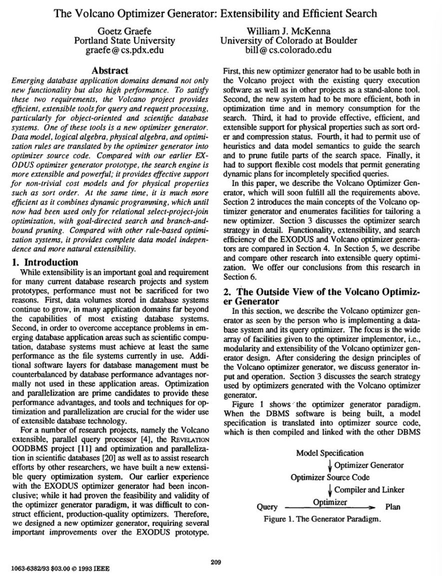
CONCLUSION

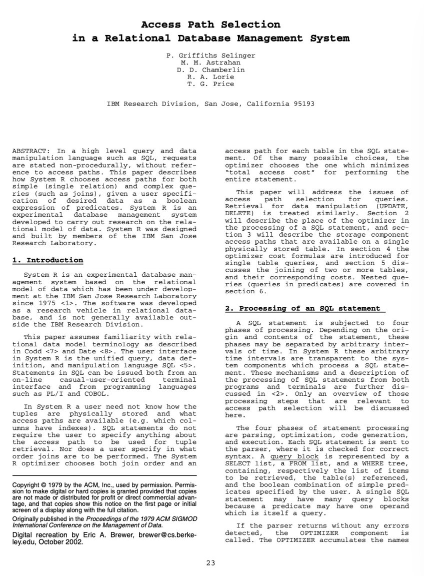
- Query optimization is critical for a database system.
- SQL -> logical plan -> physical plan.
- Flatten queries before going to the optimization part. Expression handling is also important.
- QO enumeration can be bottom-up or top-down.
- Need to cost each plan, so need cost-estimation methods.

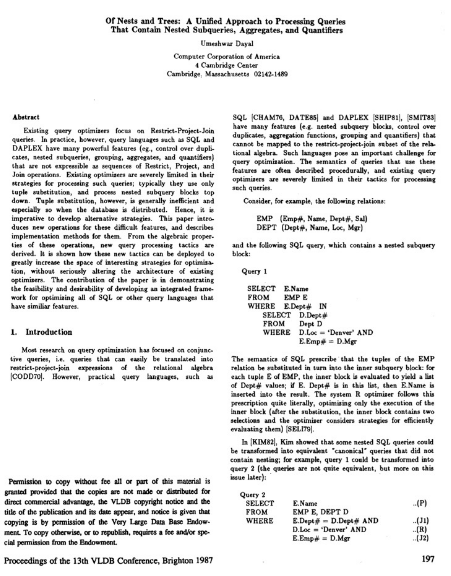
Bonus

Essential Query Optimization papers









Surajit Chaudhuri: An Overview of Query Optimization in Relational Systems. PODS 1998: 34-43

Goetz Graefe, William J. McKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE 1993: 209-218

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conference 1979: 23-34

Umeshwar Dayal: Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. VLDB 1987: 197-208

Bonus

Suggestions if you are going to build a QO

Rule 1: Read lots of papers, especially from the 80s & 90s.

→ Expect new combinations, only partially new core inventions.

Rule 2: Early on, test various workloads on the QO.

→ QOs harden over time as they "see" new workloads. Let them see more ASAP.

Rule 3: Throw away the initial one (or two) and start anew.

→ The hard part is going to be nitty-gritty details like data structures and pointers to shared objects; e.g., the list of predicates and the query graph structure, … You will NOT get this right in the first pass. Don't try to patch; be prepared to rewrite.

NEXT CLASS

Transactions!
→ aka the second hardest part about database systems