

Lecture #15

Query Planning & Optimization



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

π _{ename}

2,000 + 4 writes

(10K/500 = 20 emps per dept)

↑

σ _{dname = 'Toy'}

1,000,000 + 2,000 writes

(FK join, 10K tuples in temp T2)

↑

σ _{EMP.did = DEPT.did}

50 + 50,000 + 1,000,000 writes

(write to temp file T1)

↑

×

5 tuples per page in T1

↗

↖

EMP

DEPT

Query

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

Total: 54K I/Os

Catalog

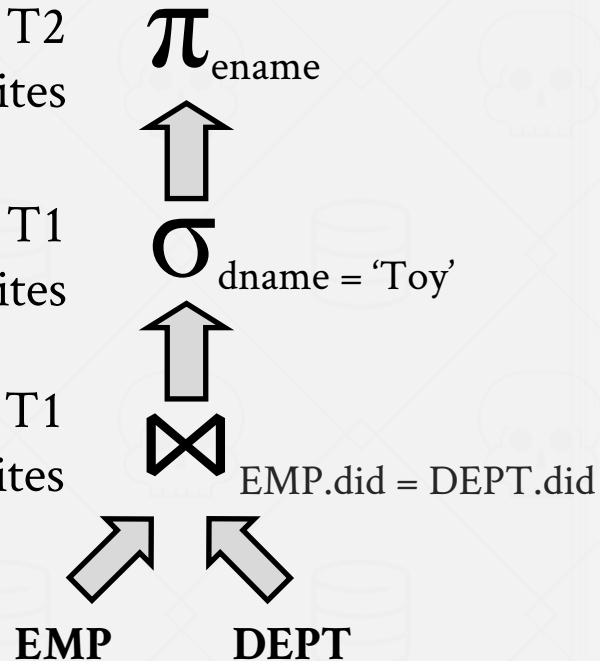
clustered	nonclustered	nonclustered
▲	△	△
EMP (ssn, ename, addr, sal, did)		
10,000 records		
1,000 pages		

clustered	nonclustered
▲	△
DEPT (did, dname, floor, mgr)	
500 records	
50 pages	

Read temp T2
4 reads + 1 writes

Read temp T1
2,000 reads + 4 writes

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



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

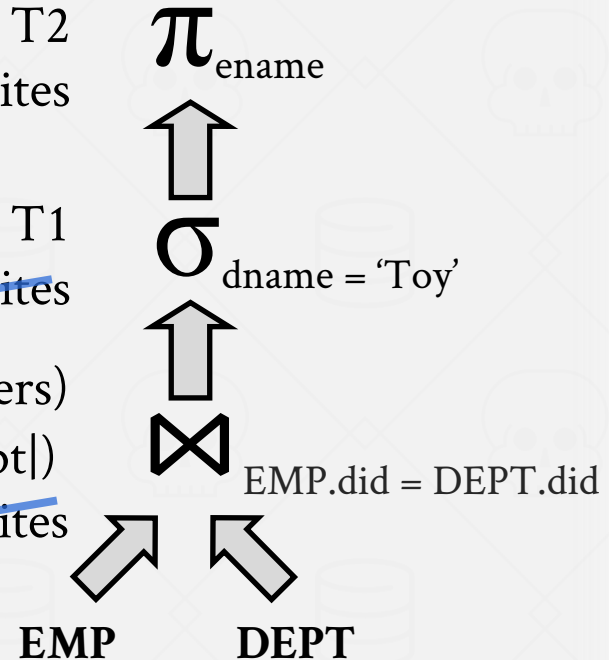
clustered  nonclustered 
 DEPT (did, dname, floor, mgr)
 500 records
 50 pages

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

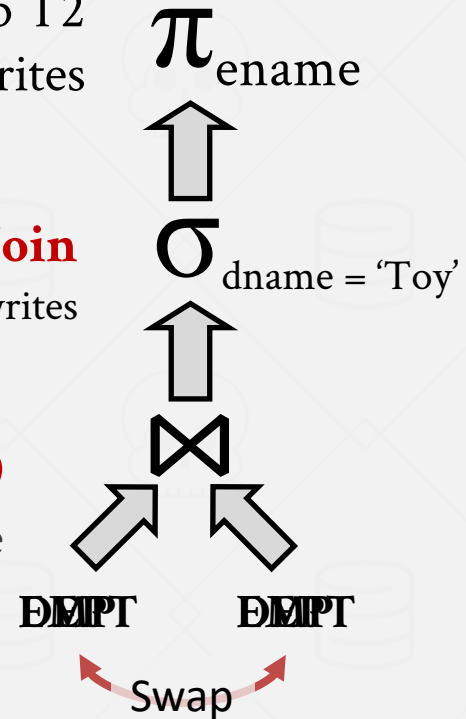
clustered nonclustered nonclustered
 ▲ △ △
 EMP (ssn, ename, addr, sal, did)
 10,000 records
 1,000 pages

clustered nonclustered
 ▲ △
 DEPT (did, dname, floor, mgr)
 500 records
 50 pages

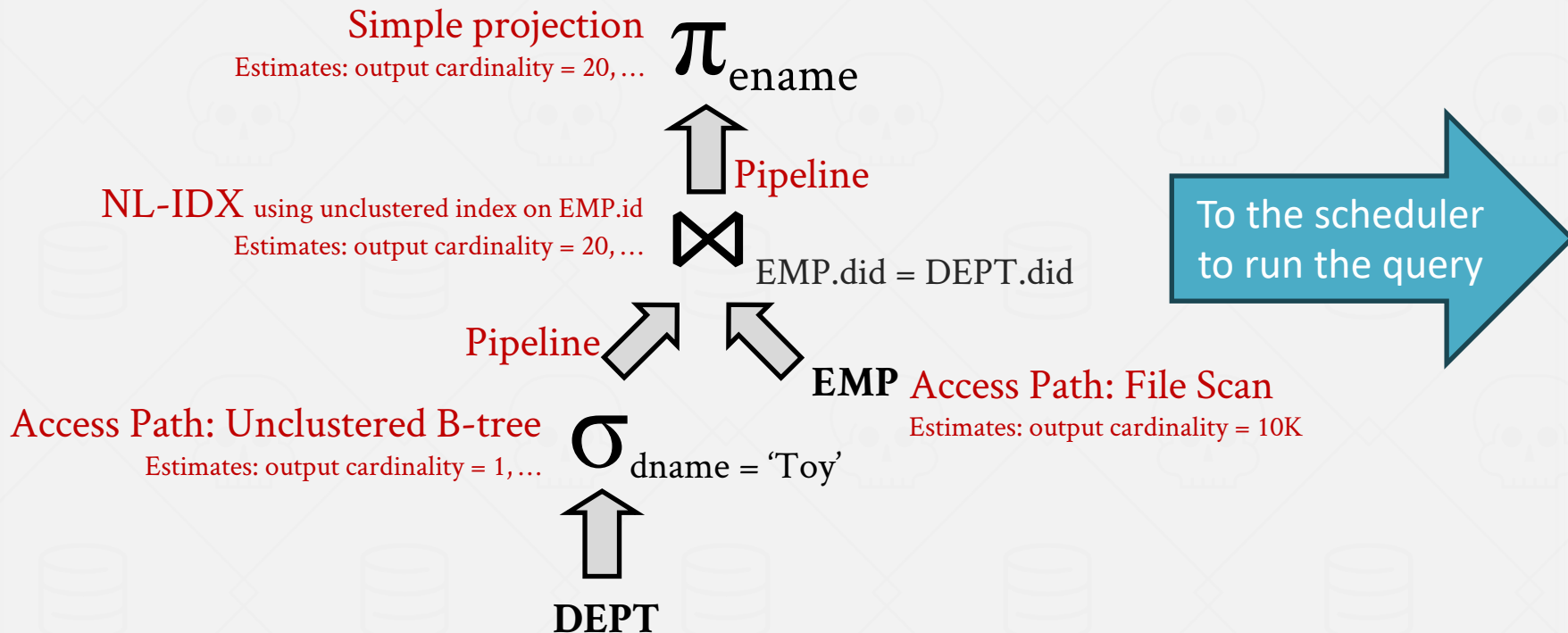
Read temp T2
4 reads + 1 writes

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

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



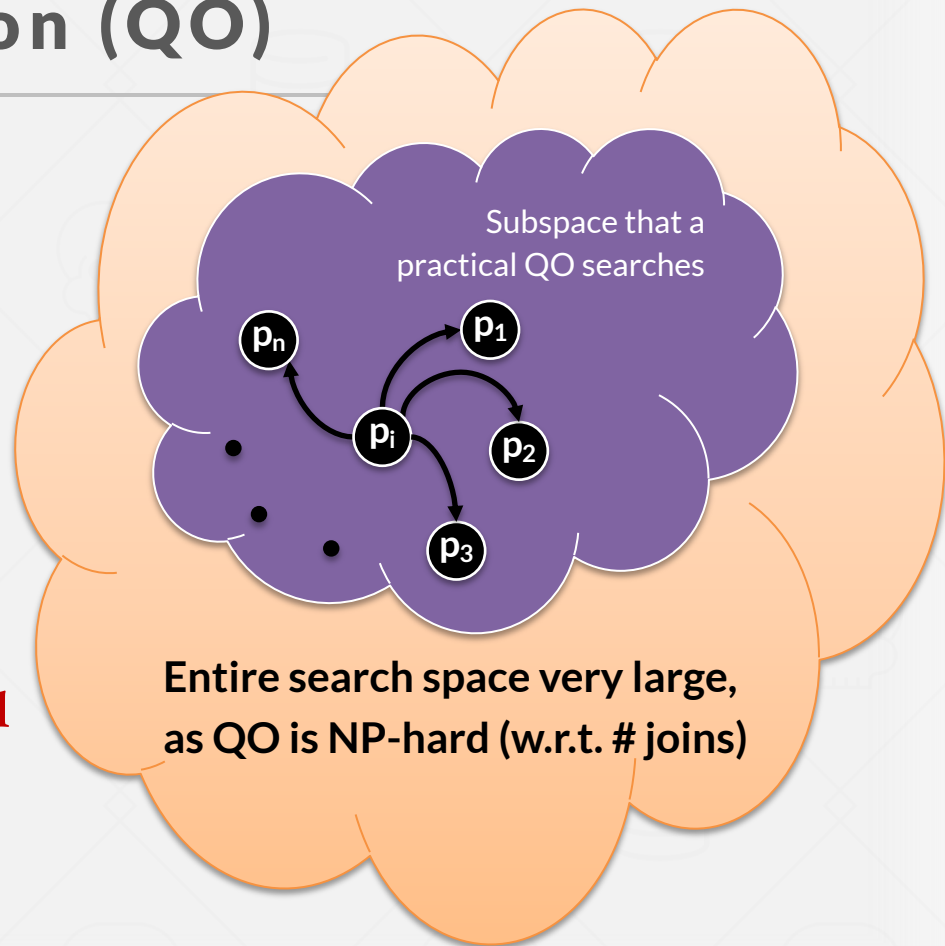
Annotated RA Tree a.k.a. The Physical Plan



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

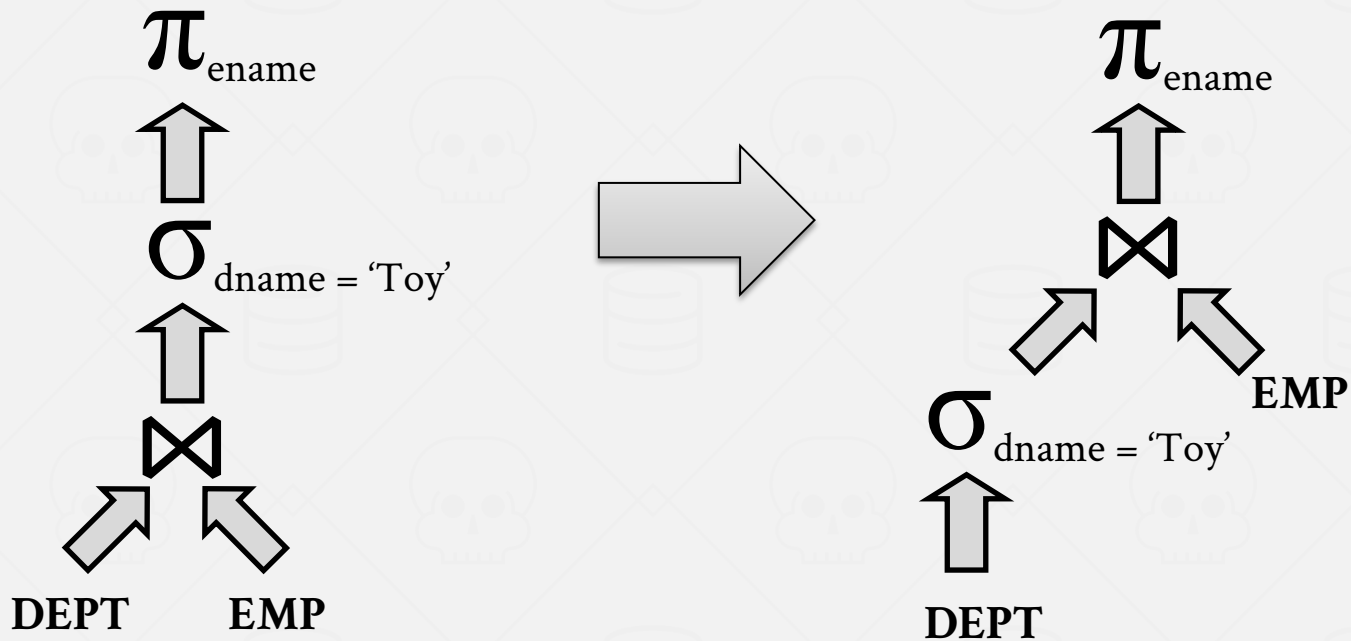
Heuristics / Rules

- 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 not need to examine data.

Cost-based Search

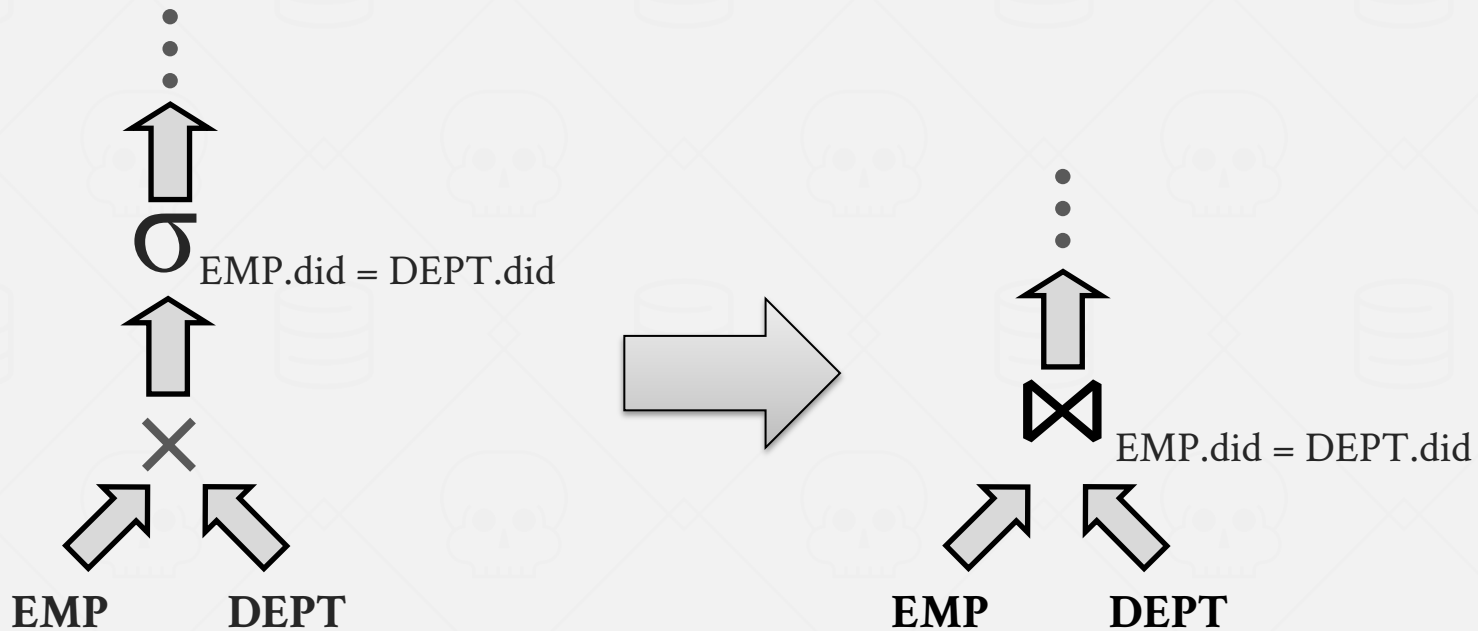
- Use a model to estimate the cost of executing a plan.
- Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

Predicate Pushdown



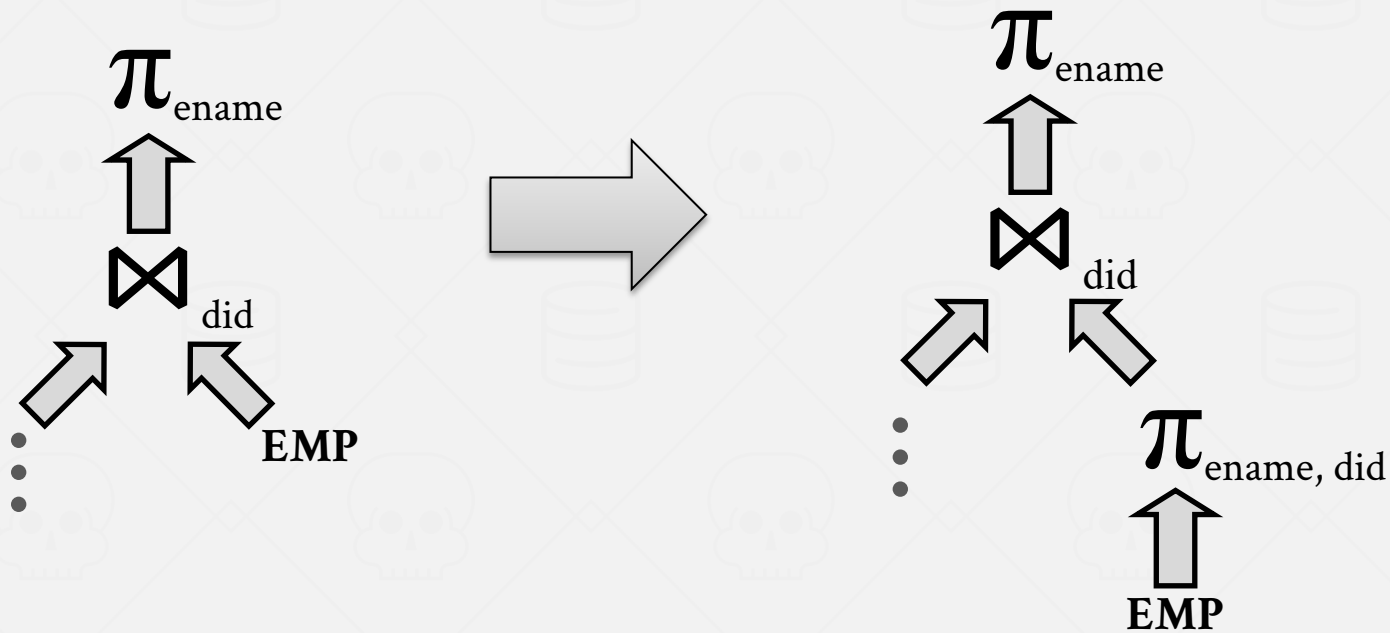
$\pi_{ename}(\sigma_{dname='Toy'}(DEPT \bowtie EMP))$
→
 rewrite
 →
 $\pi_{ename}(EMP \bowtie \sigma_{dname='Toy'}(DEPT))$

Replace Cartesian Product



$\dots (\sigma_{\text{DEPT.did} = \text{EMP.did}} (\text{DEPT} \times \text{EMP}))$
→
 rewrite
 →
 $\dots (\text{EMP} \bowtie_{\text{did}} \text{DEPT})$

Projection Pushdown



$$\pi_{\text{EMP.ename}} (\dots \bowtie_{\text{did}} \text{EMP})$$

rewrite

$$\pi_{\text{EMP.ename}} (\dots \bowtie_{\text{did}} (\pi_{\text{ename, did}} \text{EMP}))$$

Equivalence

$$\sigma_{P_1}(\sigma_{P_2}(R)) \equiv \sigma_{P_2}(\sigma_{P_1}(R)) \quad (\sigma \text{ commutativity})$$

$$\sigma_{P_1 \wedge P_2 \dots \wedge P_n}(R) \equiv \sigma_{P_1}(\sigma_{P_2}(\dots \sigma_{P_n}(R))) \quad (\text{cascading } \sigma)$$

$$\prod_{a_1}(R) \equiv \prod_{a_1}(\prod_{a_2}(\dots \prod_{a_k}(R)\dots)), a_i \subseteq a_{i+1} \quad (\text{cascading } \prod)$$

$$R \bowtie S \equiv S \bowtie R \quad (\text{join commutativity})$$

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad (\text{join associativity})$$

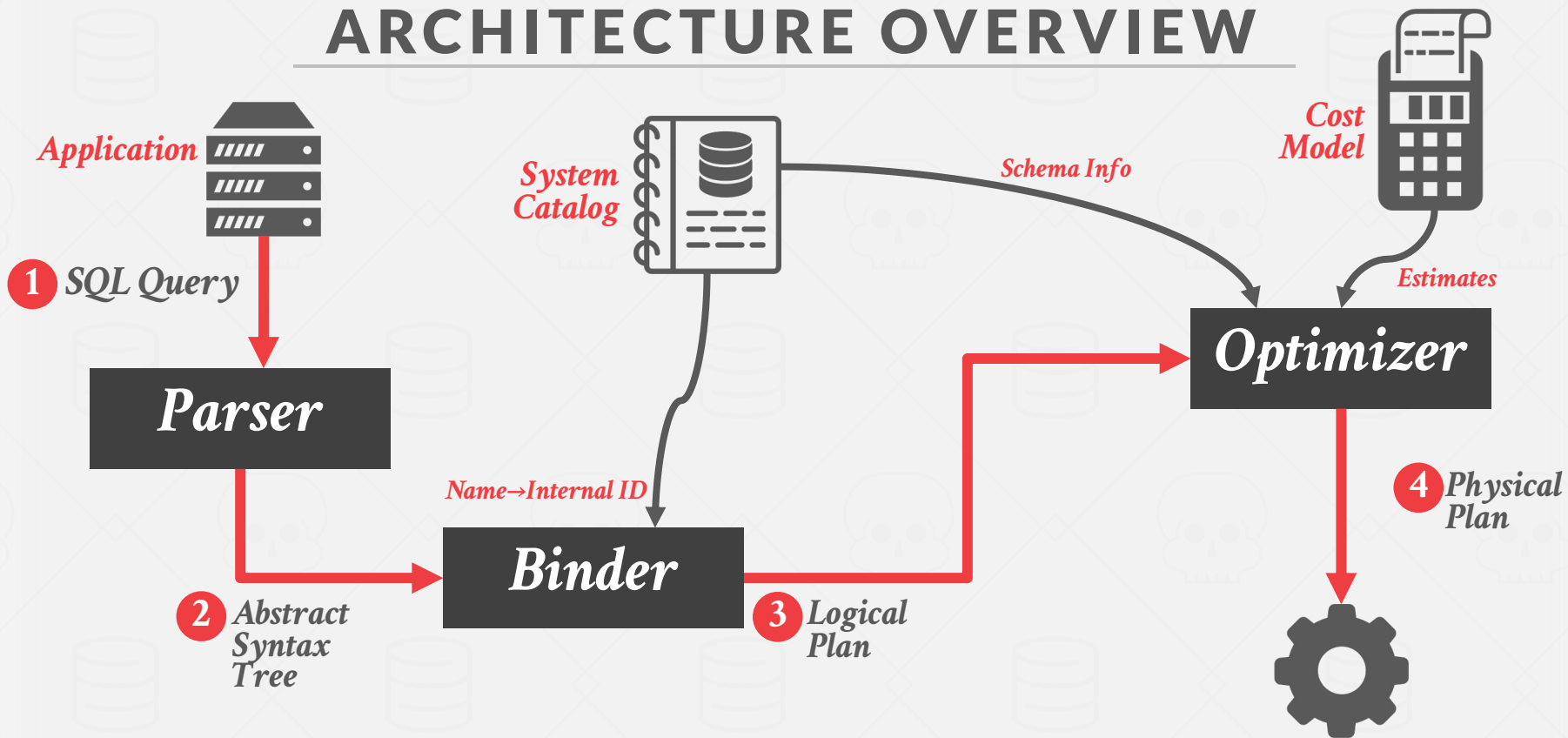
$$\sigma_P(R \bowtie S) \equiv (R \bowtie_P S), \text{ if } P \text{ is a join predicate}$$

$$\sigma_P(R \bowtie S) \equiv \sigma_{P_1}(\sigma_{P_2}(R) \bowtie_{P_4} \sigma_{P_3}(S)), \text{ where } P = p_1 \wedge p_2 \wedge p_3 \wedge p_4$$

$$\prod_{A_1, A_2, \dots, A_n}(\sigma_P(R)) \equiv \prod_{A_1, A_2, \dots, A_n}(\sigma_P(\prod_{A_1, \dots, A_n, B_1, \dots, B_M}(R))), \text{ where } B_1 \dots B_M \text{ are columns in } P$$

...

ARCHITECTURE OVERVIEW



QUERY OPTIMIZATION

Heuristics / Rules

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

- 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

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.

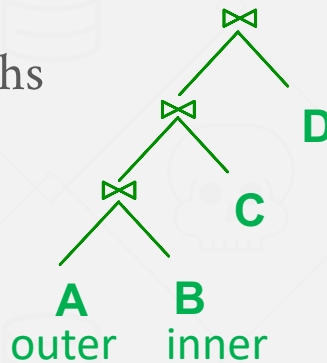
→ All combinations of join algorithms and access paths

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

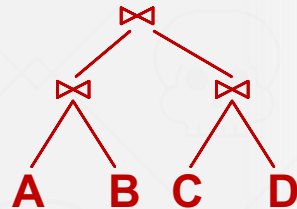


Selinger

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
⋮		⋮		⋮

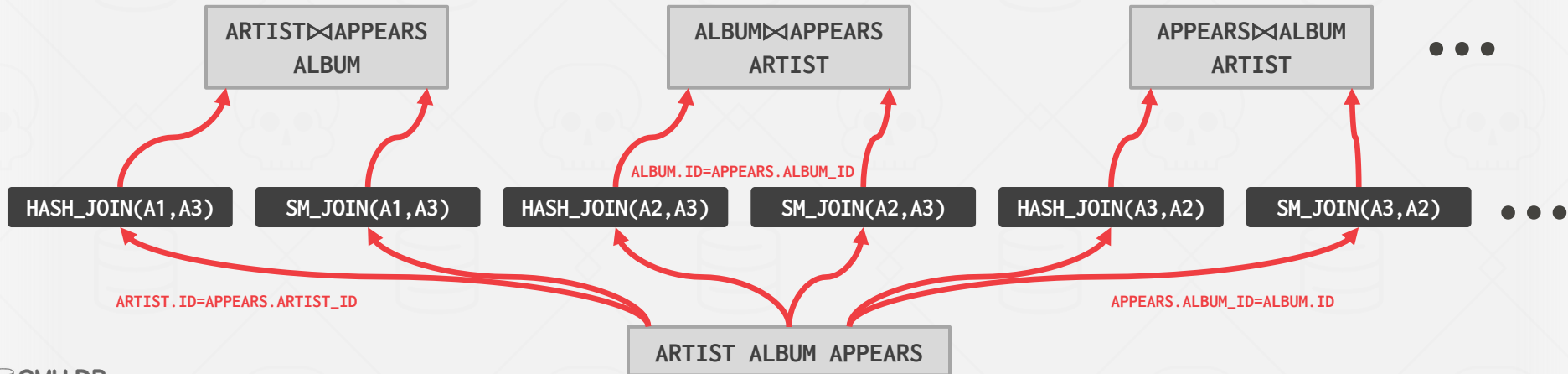
SYSTEM R OPTIMIZER

ARTIST \bowtie APPEARS \bowtie ALBUM

ARTIST ALBUM APPEARS

SYSTEM R OPTIMIZER

ARTIST ⋈ APPEARS ⋈ ALBUM



SYSTEM R OPTIMIZER

ARTIST \bowtie APPEARS \bowtie ALBUM

ARTIST \bowtie APPEARS
ALBUM

ALBUM \bowtie APPEARS
ARTIST

APPEARS \bowtie ALBUM
ARTIST

HASH_JOIN(A1, A3)

HASH_JOIN(A2, A3)

SM_JOIN(A3, A2)

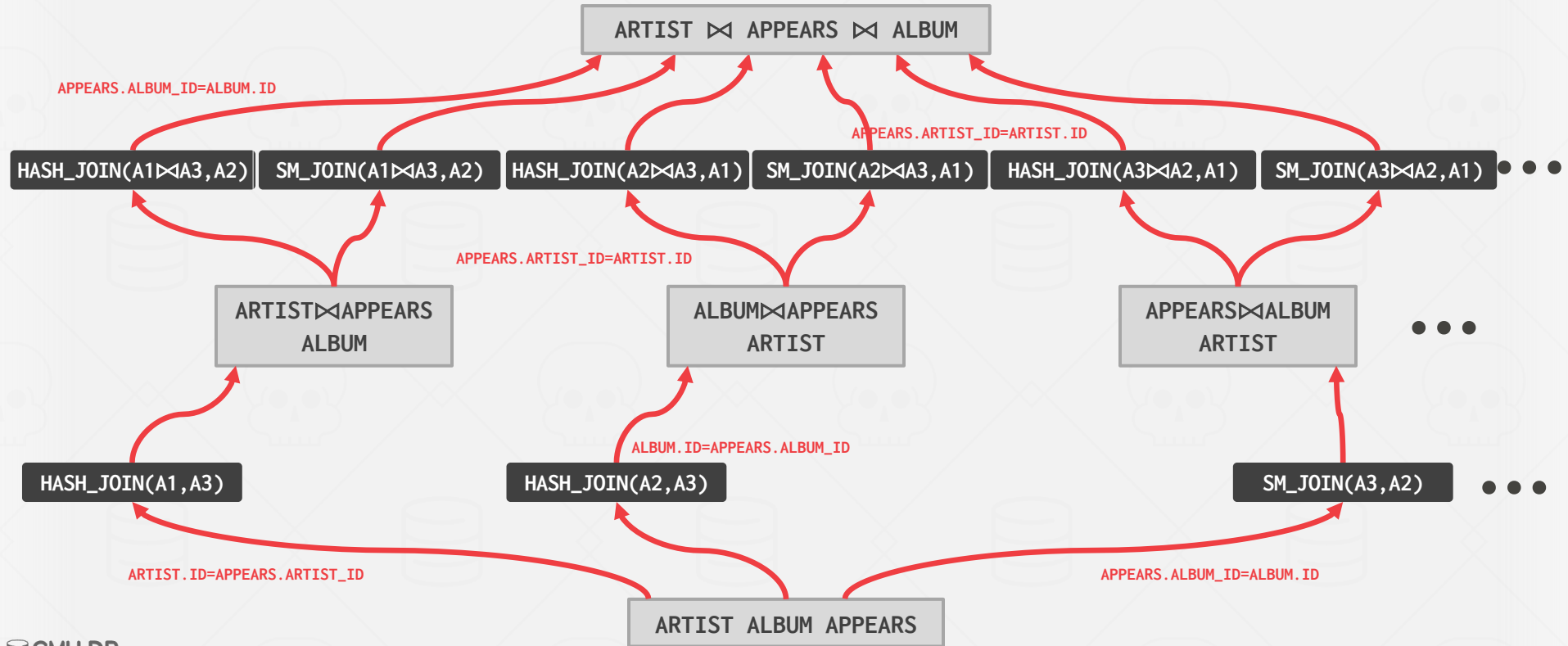
ARTIST.ID=APPEARS.ARTIST_ID

ALBUM.ID=APPEARS.ALBUM_ID

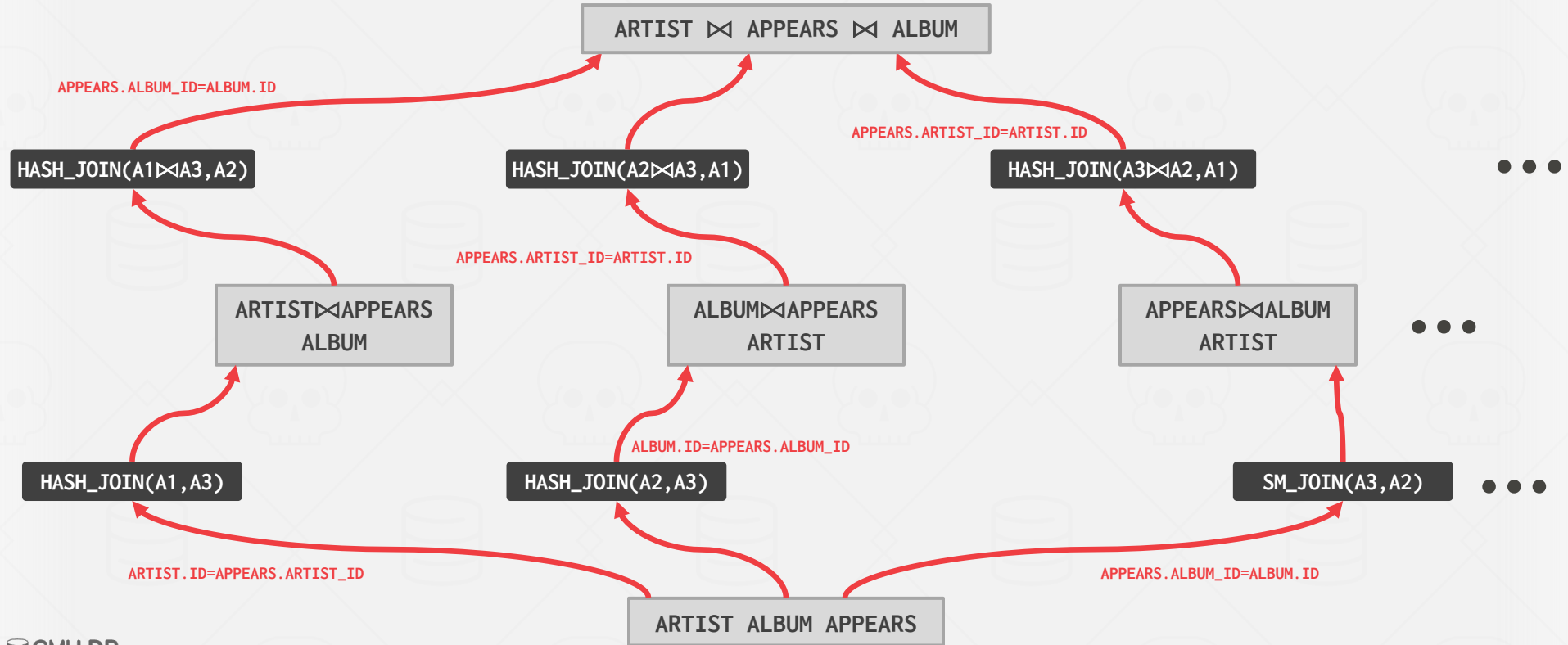
APPEARS.ALBUM_ID=ALBUM.ID

ARTIST ALBUM APPEARS

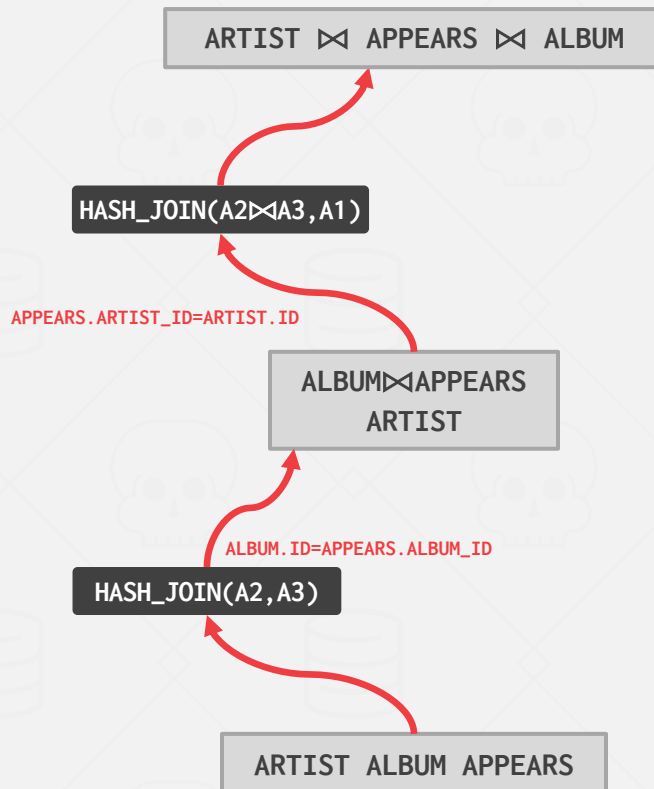
SYSTEM R OPTIMIZER



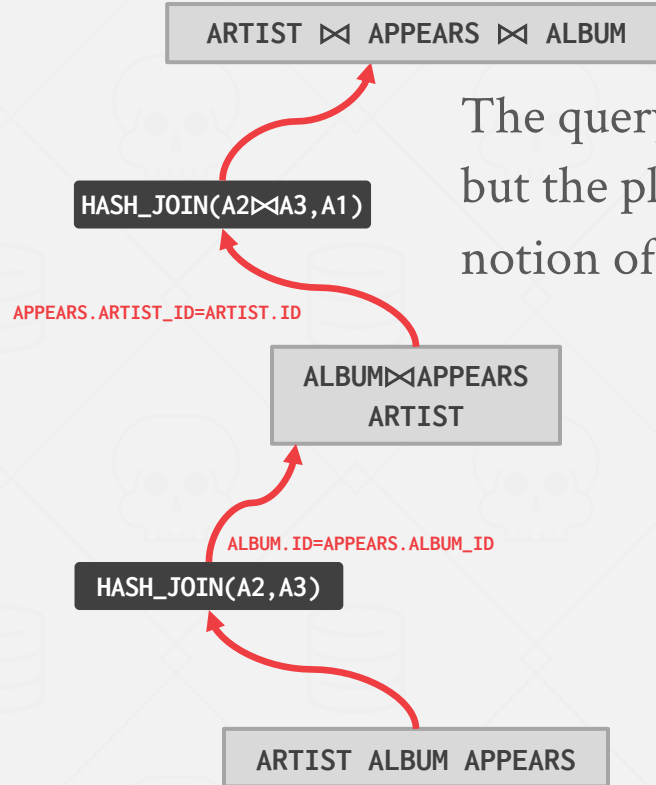
SYSTEM R OPTIMIZER



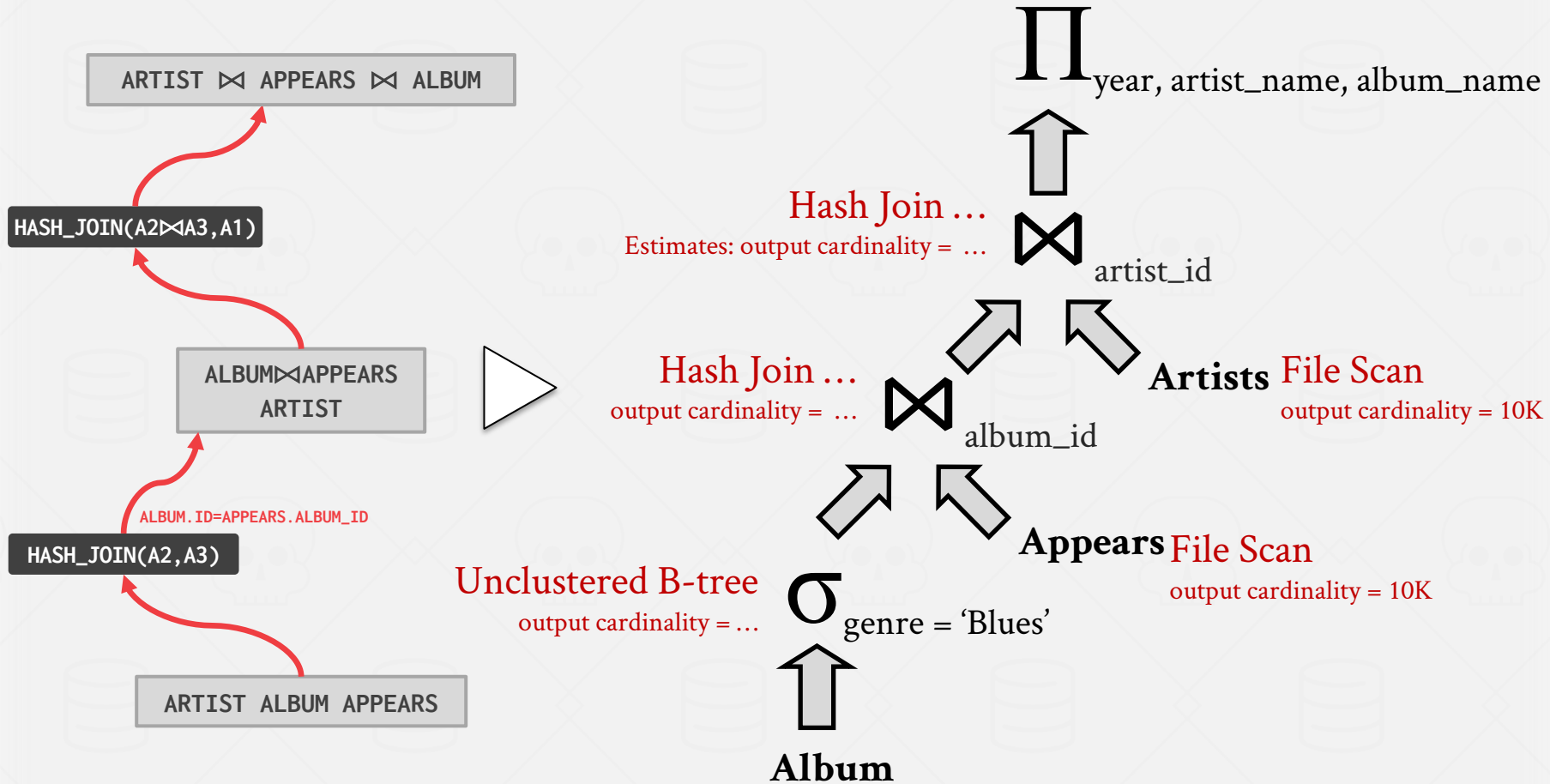
SYSTEM R OPTIMIZER



SYSTEM R OPTIMIZER



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



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)



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

ARTIST⋈APPEARS

ALBUM⋈APPEARS

ARTIST⋈ALBUM

ARTIST

ALBUM

APPEARS

TOP-DOWN OPTIMIZATION

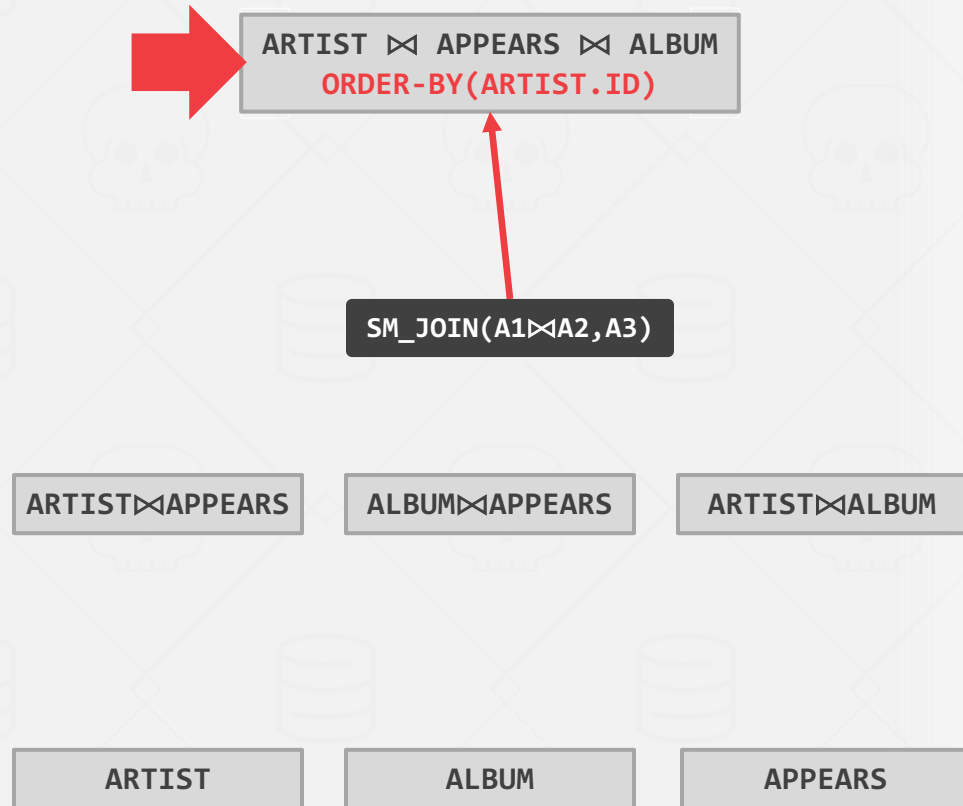
Invoke rules to create new nodes and traverse the tree.

→ **Logical**→**Logical**:

JOIN(A, B) to JOIN(B, A)

→ **Logical**→**Physical**:

JOIN(A, B) to HASH_JOIN(A, B)



TOP-DOWN OPTIMIZATION

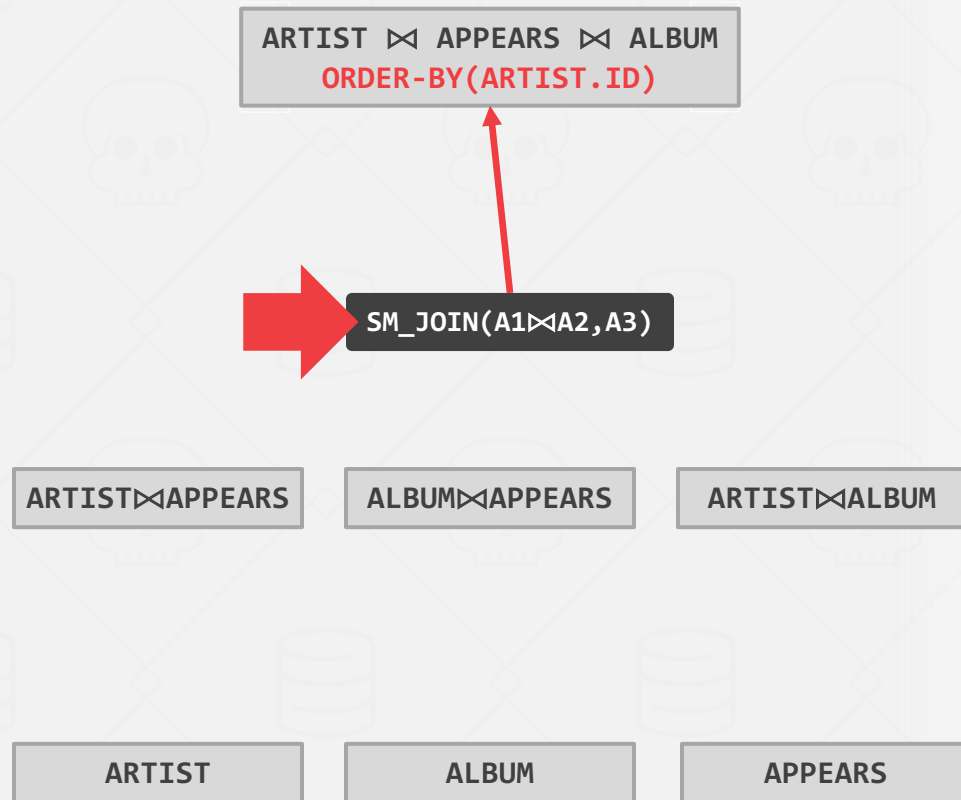
Invoke rules to create new nodes
and traverse the tree.

→ **Logical**→**Logical**:

JOIN(A, B) to JOIN(B, A)

→ **Logical**→**Physical**:

JOIN(A, B) to HASH_JOIN(A, B)



TOP-DOWN OPTIMIZATION

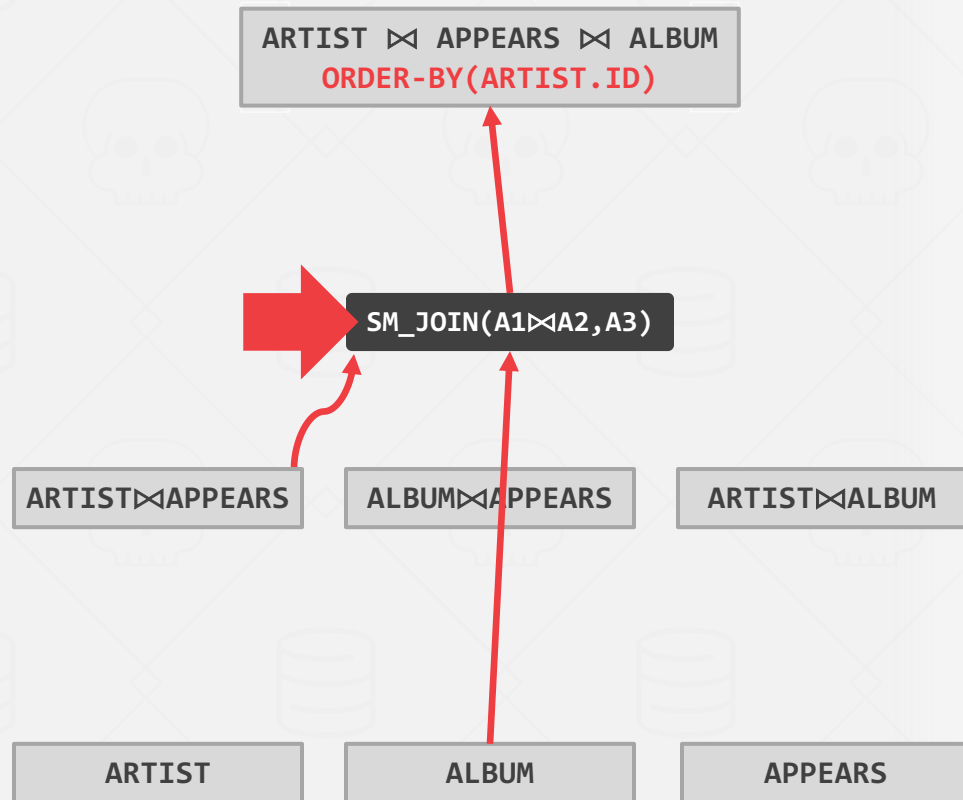
Invoke rules to create new nodes
and traverse the tree.

→ **Logical**→**Logical**:

JOIN(A, B) to JOIN(B, A)

→ **Logical**→**Physical**:

JOIN(A, B) to HASH_JOIN(A, B)



TOP-DOWN OPTIMIZATION

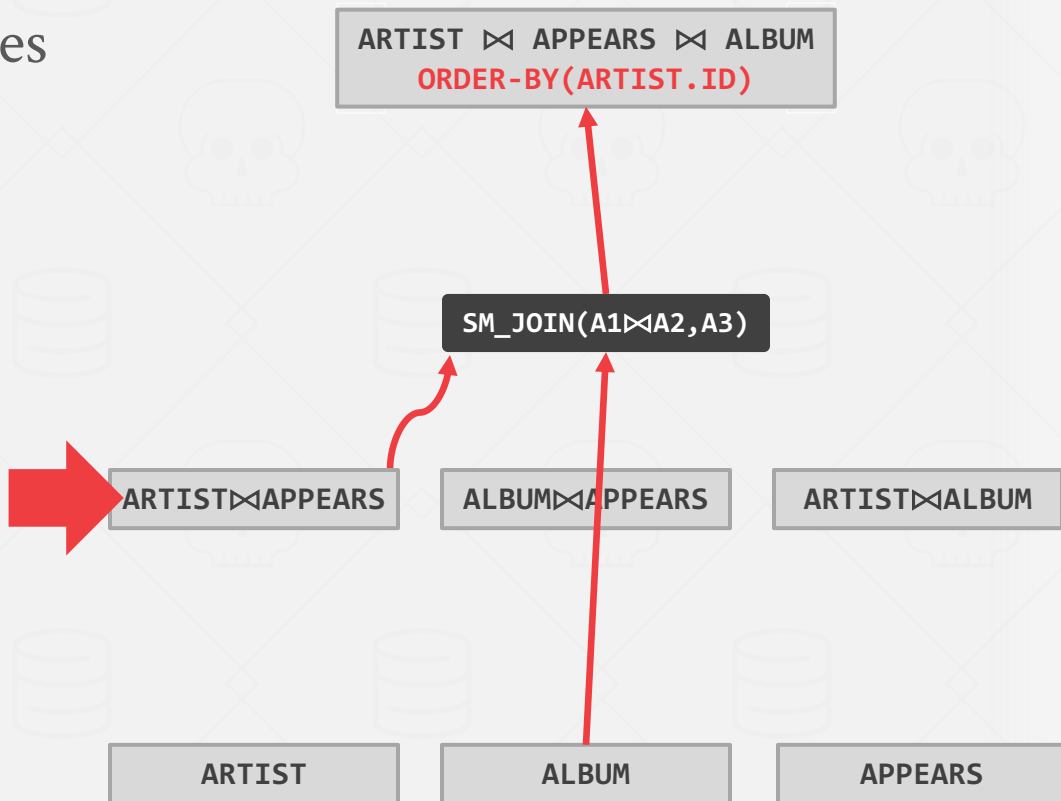
Invoke rules to create new nodes
and traverse the tree.

→ **Logical**→**Logical**:

JOIN(A, B) to JOIN(B, A)

→ **Logical**→**Physical**:

JOIN(A, B) to HASH_JOIN(A, B)



TOP-DOWN OPTIMIZATION

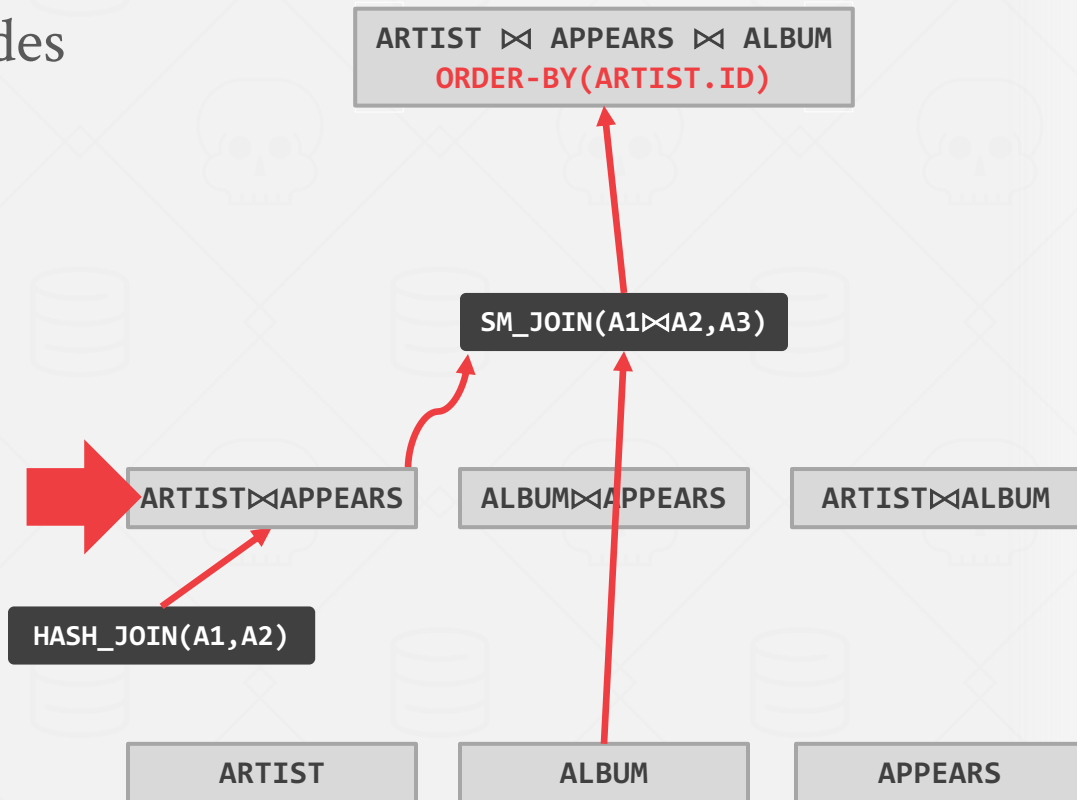
Invoke rules to create new nodes
and traverse the tree.

→ **Logical**→**Logical**:

JOIN(A, B) to JOIN(B, A)

→ **Logical**→**Physical**:

JOIN(A, B) to HASH_JOIN(A, B)



TOP-DOWN OPTIMIZATION

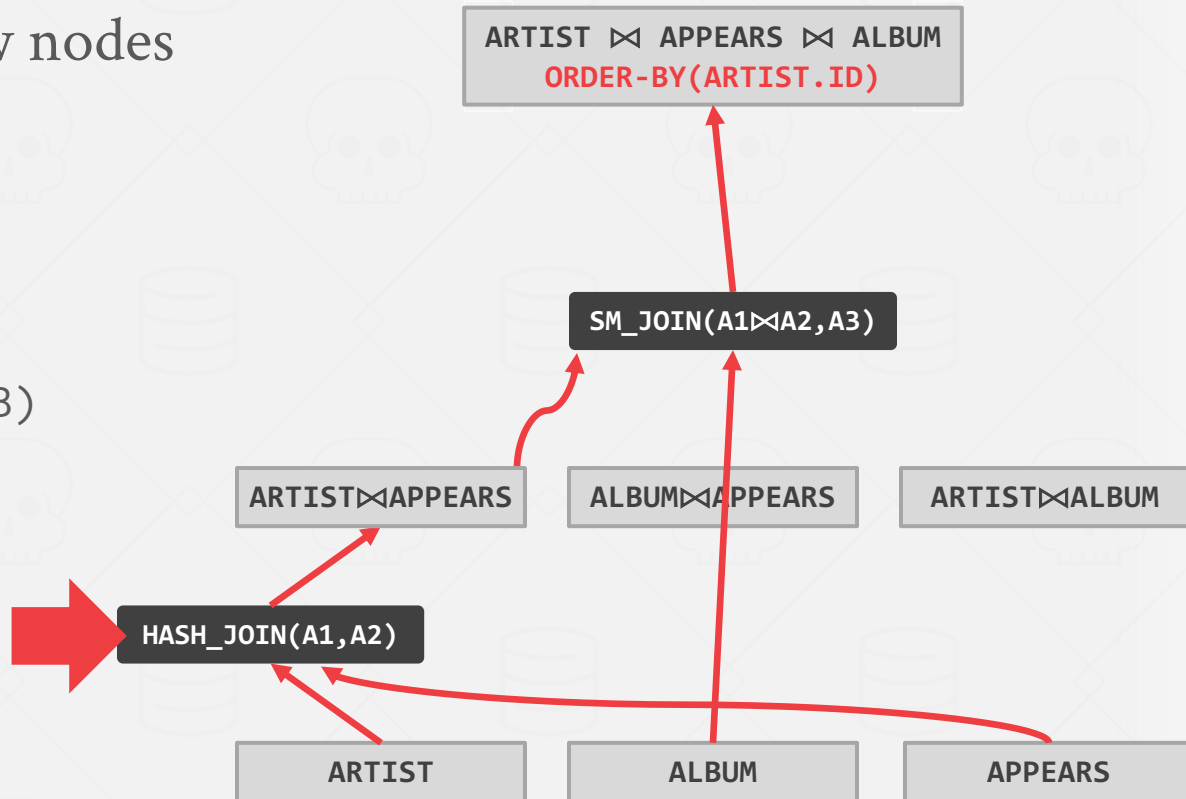
Invoke rules to create new nodes
and traverse the tree.

→ **Logical**→**Logical**:

JOIN(A, B) to JOIN(B, A)

→ **Logical**→**Physical**:

JOIN(A, B) to HASH_JOIN(A, B)



TOP-DOWN OPTIMIZATION

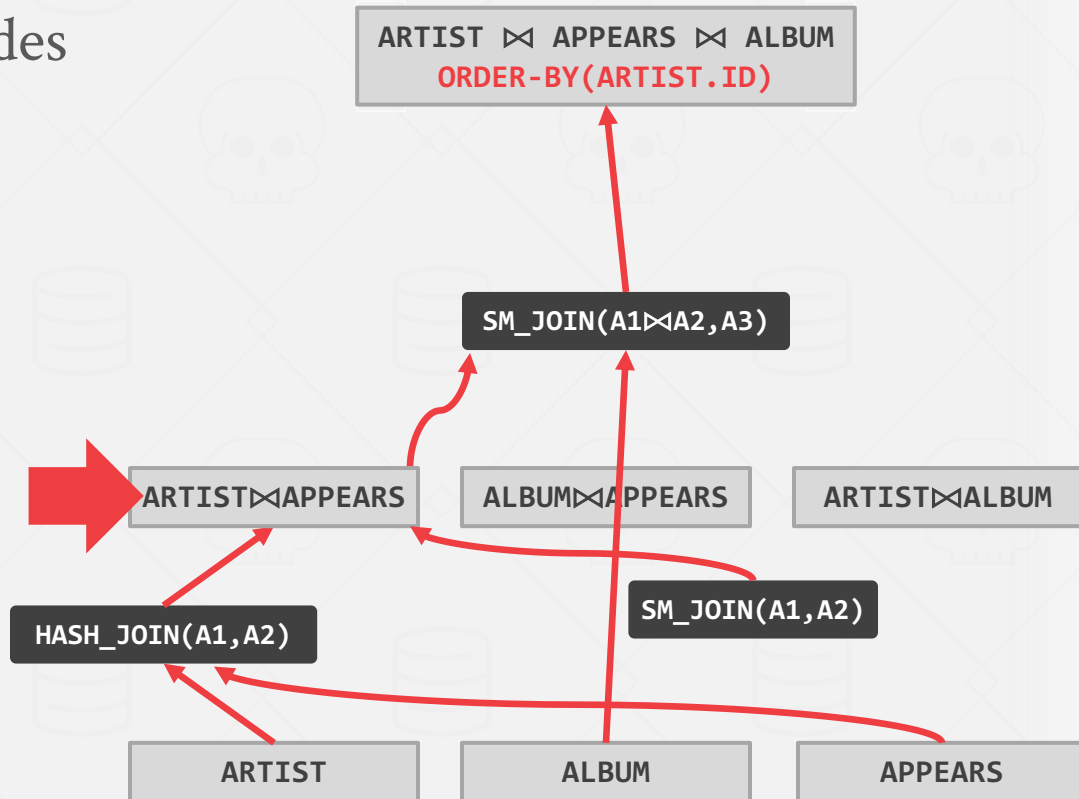
Invoke rules to create new nodes
and traverse the tree.

→ **Logical**→**Logical**:

JOIN(A, B) to JOIN(B, A)

→ **Logical**→**Physical**:

JOIN(A, B) to HASH_JOIN(A, B)



TOP-DOWN OPTIMIZATION

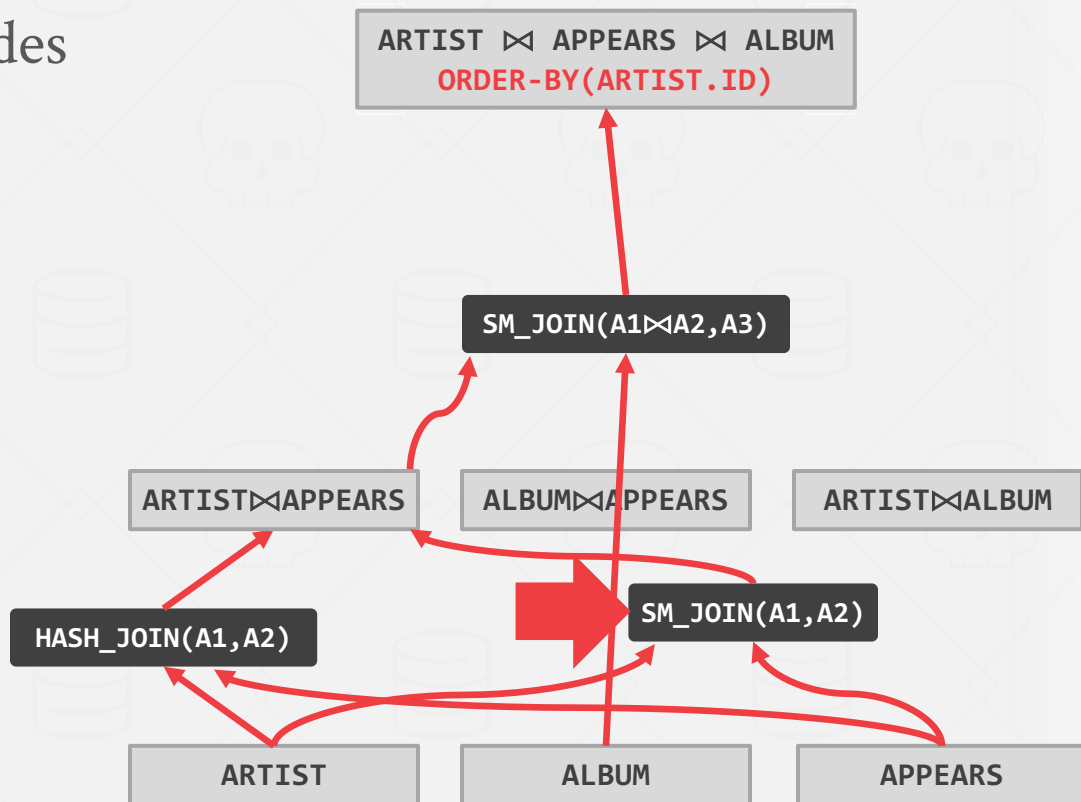
Invoke rules to create new nodes
and traverse the tree.

→ **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.

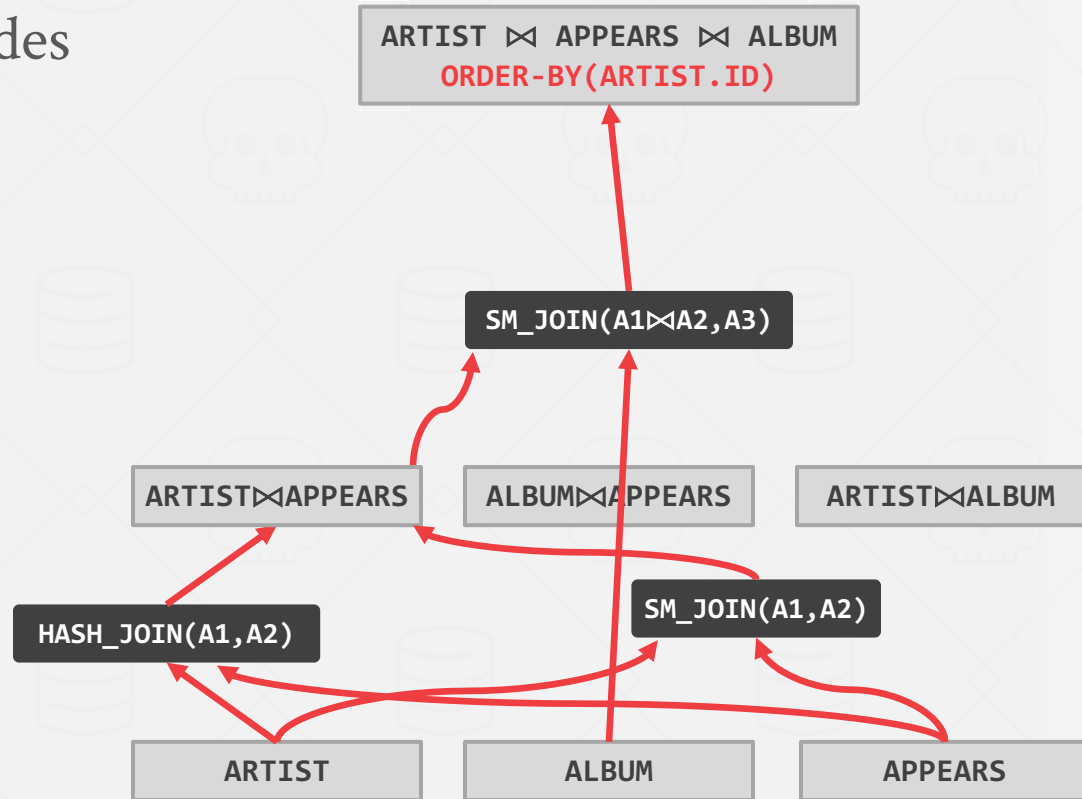
→ **Logical**→**Logical**:

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.

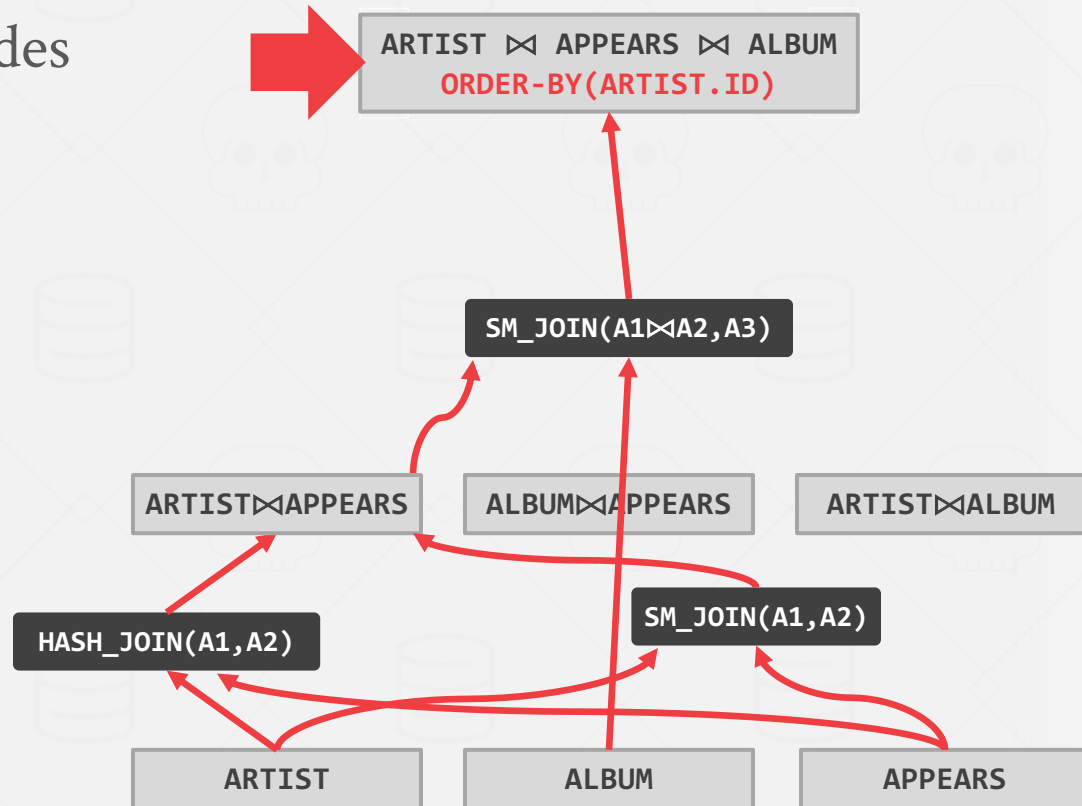
→ **Logical**→**Logical**:

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.

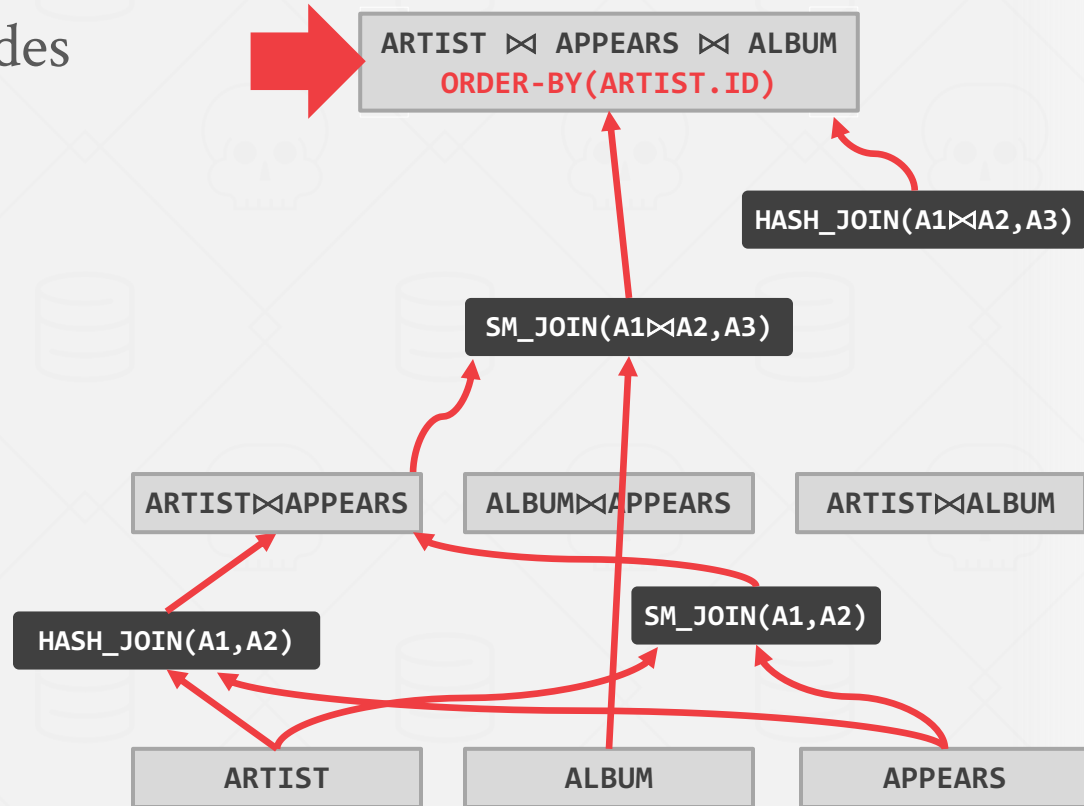
→ **Logical**→**Logical**:

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.

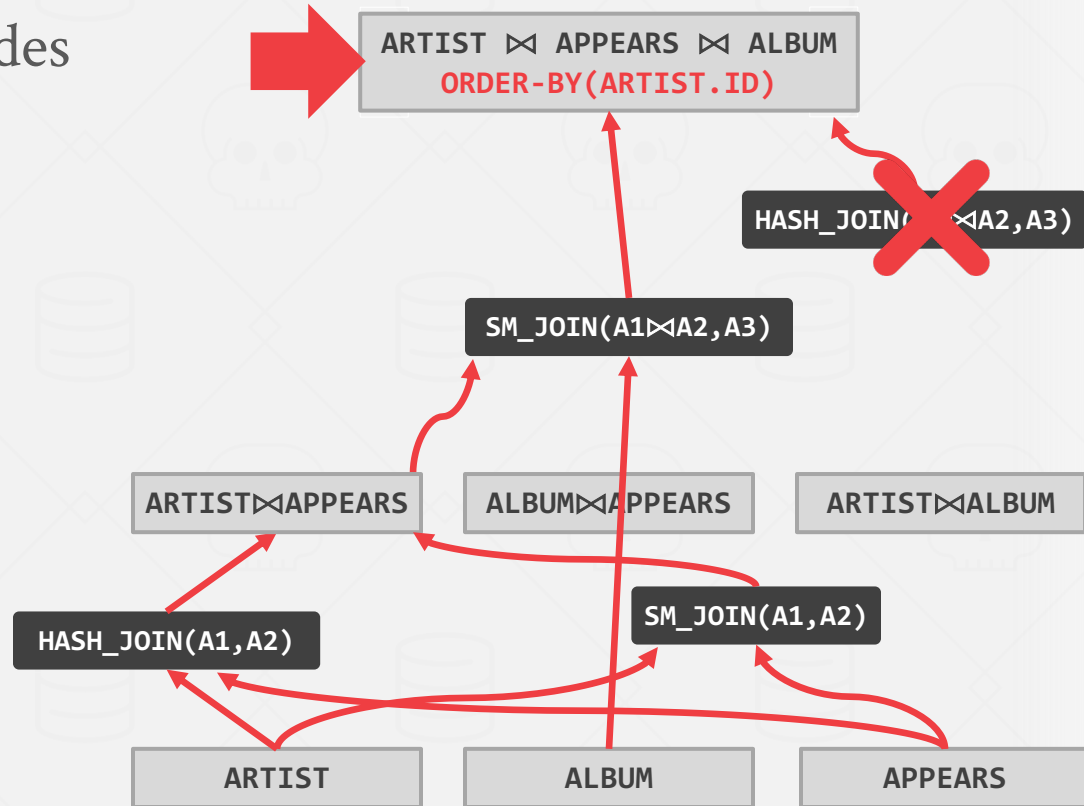
→ **Logical**→**Logical**:

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.

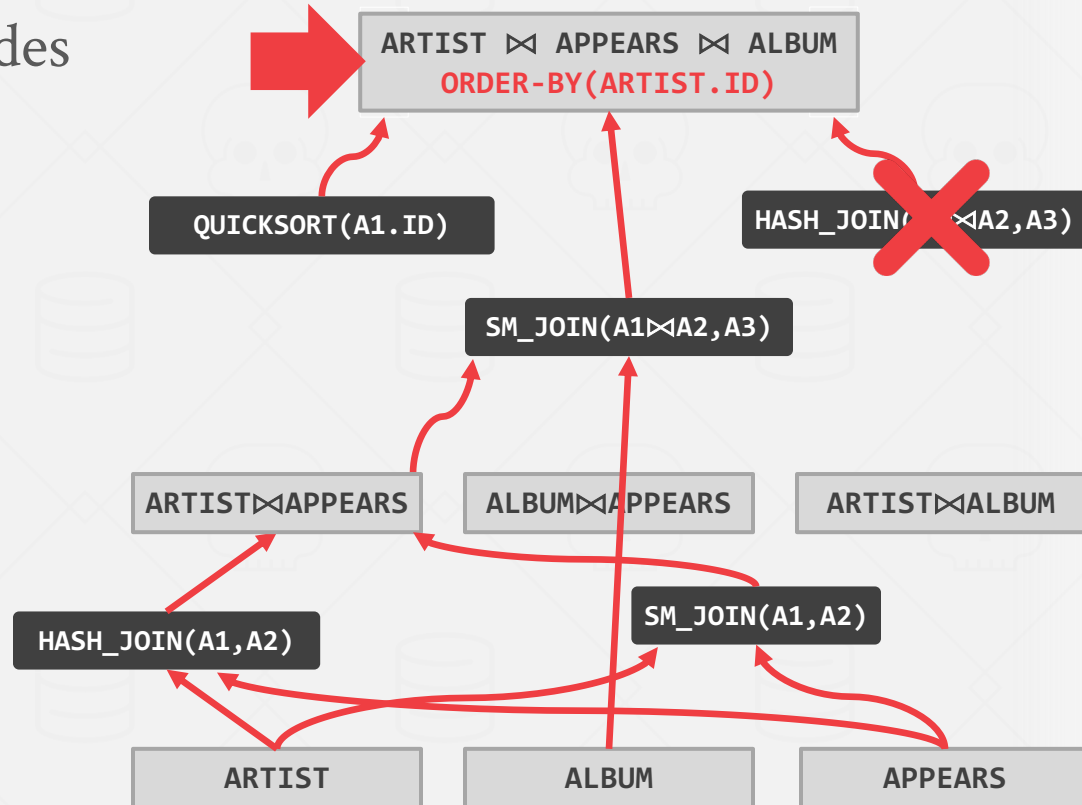
→ **Logical**→**Logical**:

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.

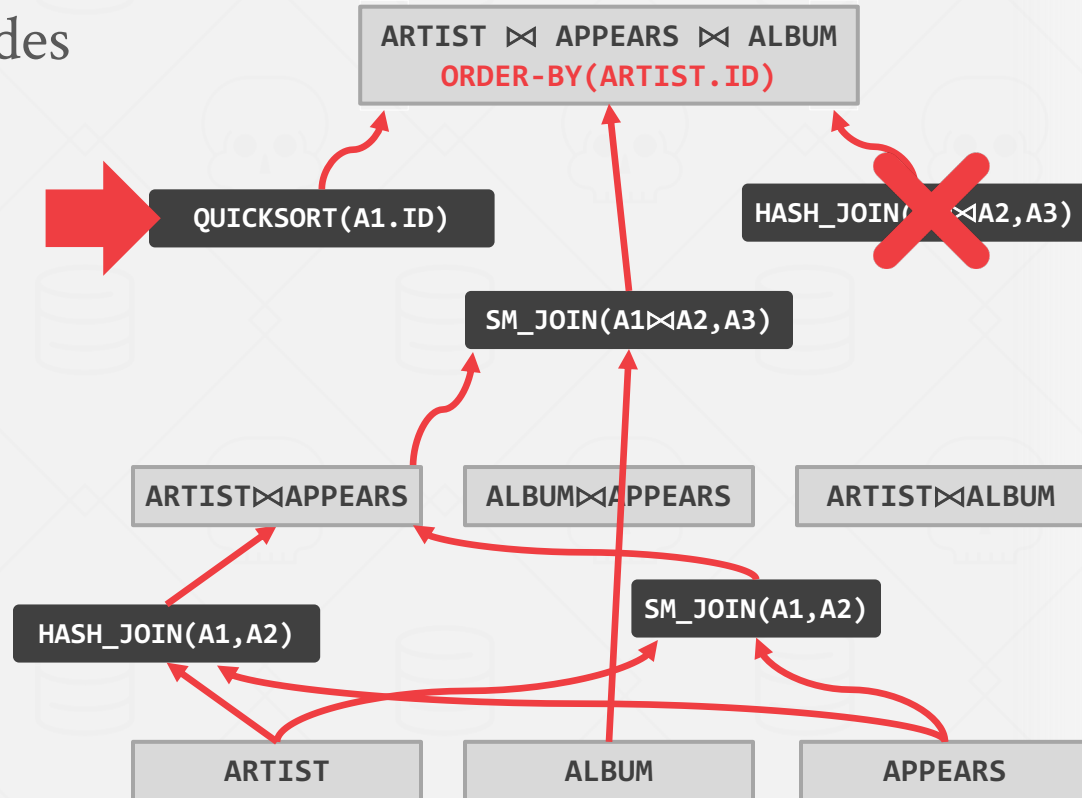
→ **Logical**→**Logical**:

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.

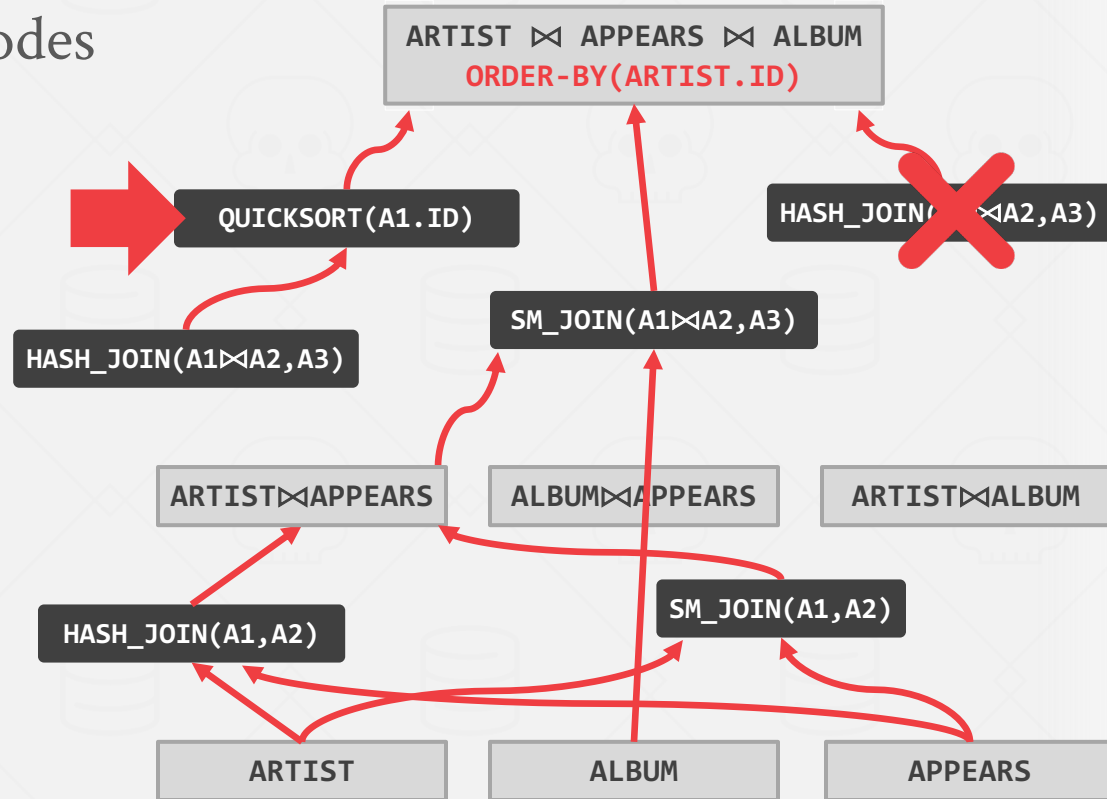
→ **Logical**→**Logical**:

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.

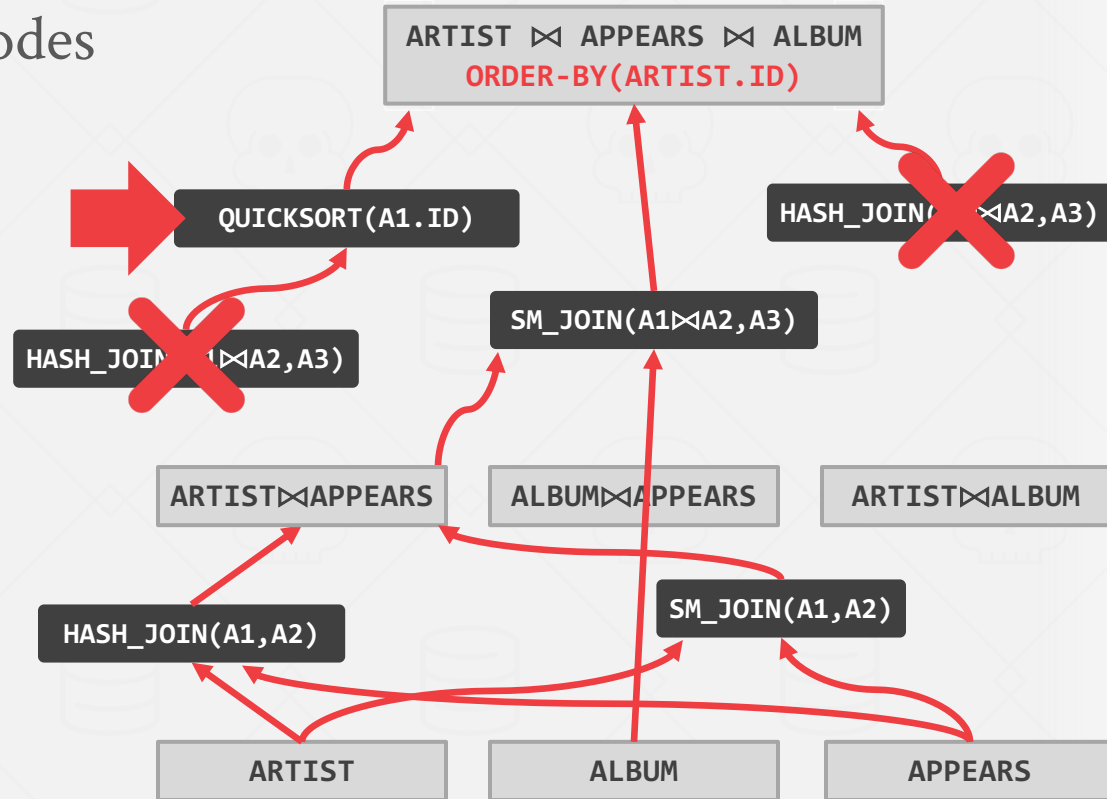
→ **Logical**→**Logical**:

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 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.

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;
```

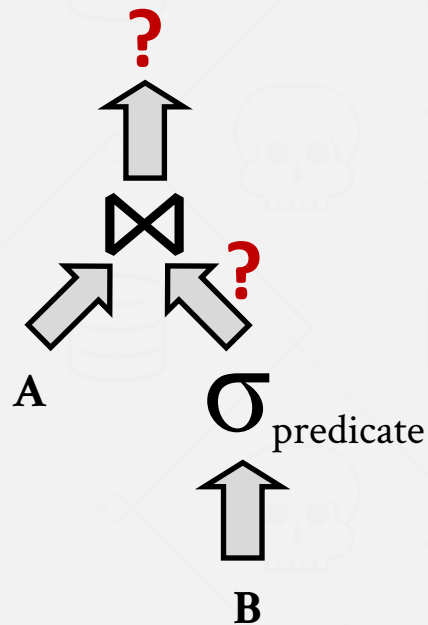
```
SELECT * FROM A WHERE RANDOM() IS NULL;
```

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 formulae 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

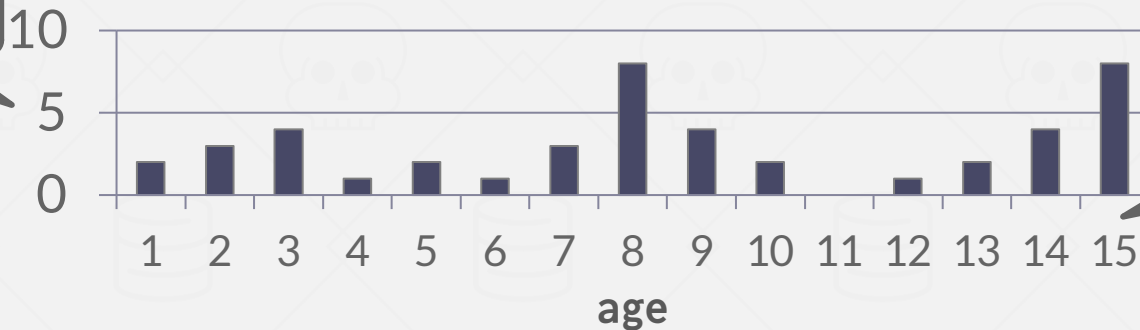
The selectivity (**sel**) of a predicate **P** is the fraction of tuples that qualify.

Equality Predicate: $A = \text{constant}$

$\rightarrow \text{sel}(A = \text{constant}) = \text{\#occurrences} / |R|$

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

of occurrences



Distinct values of attribute

SELECTION CARDINALITY

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

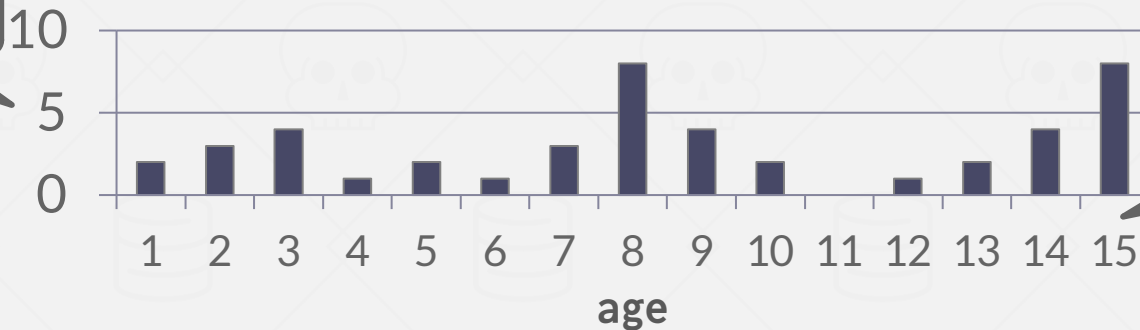
Equality Predicate: $A = \text{constant}$

→ $\text{sel}(A = \text{constant}) = \text{\#occurrences} / |R|$

→ Example: $\text{sel}(\text{age} = 9) =$

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

of occurrences



Distinct values of attribute

SELECTION CARDINALITY

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

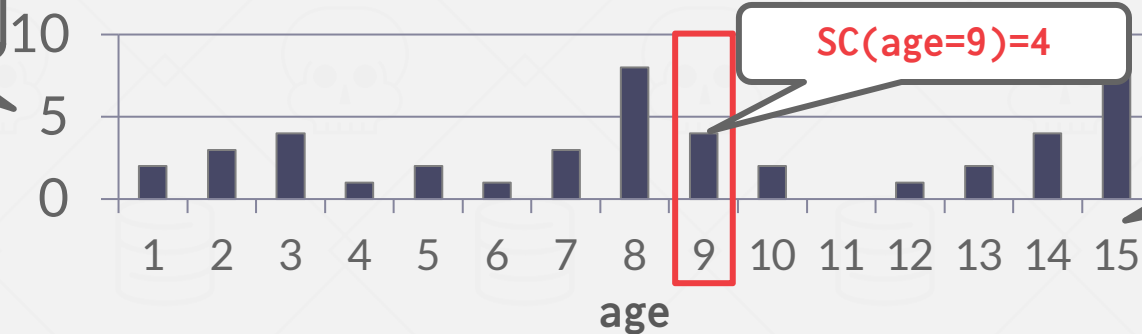
Equality Predicate: $A = \text{constant}$

→ $\text{sel}(A = \text{constant}) = \text{\#occurrences} / |R|$

→ Example: $\text{sel}(\text{age} = 9) =$

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

of occurrences



$SC(\text{age} = 9) = 4$

Distinct values of attribute

SELECTION CARDINALITY

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

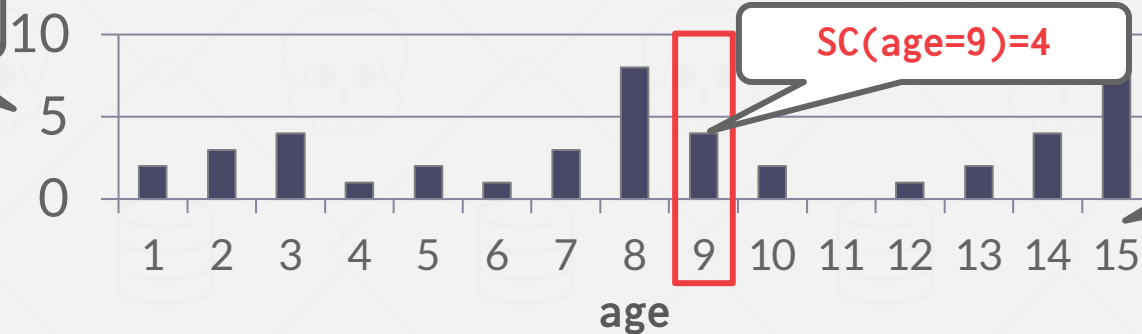
Equality Predicate: $A = \text{constant}$

→ $\text{sel}(A = \text{constant}) = \text{\#occurrences} / |R|$

→ Example: $\text{sel}(\text{age} = 9) = 4/45$

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

of occurrences



$SC(\text{age} = 9) = 4$

Distinct values of attribute

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 $1/100 = 0.01$

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

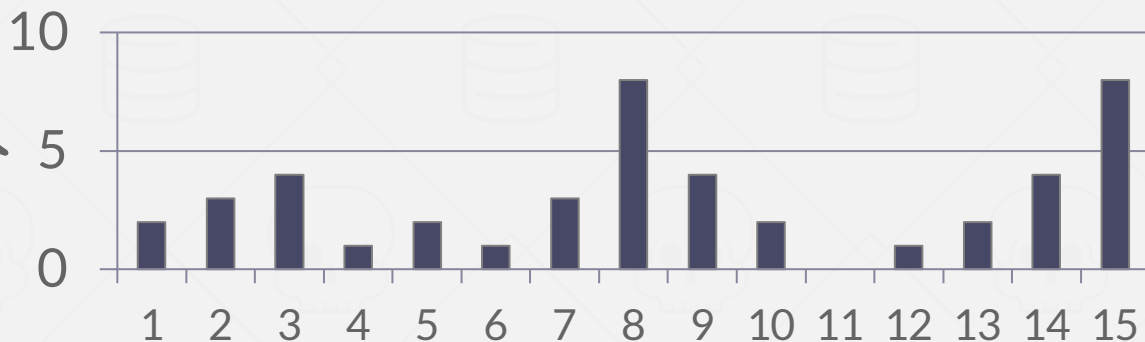
→ DBMS maintains a small subset of each table that it then uses to evaluate expressions to compute selectivity.

HISTOGRAMS

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

Histogram

of occurrences



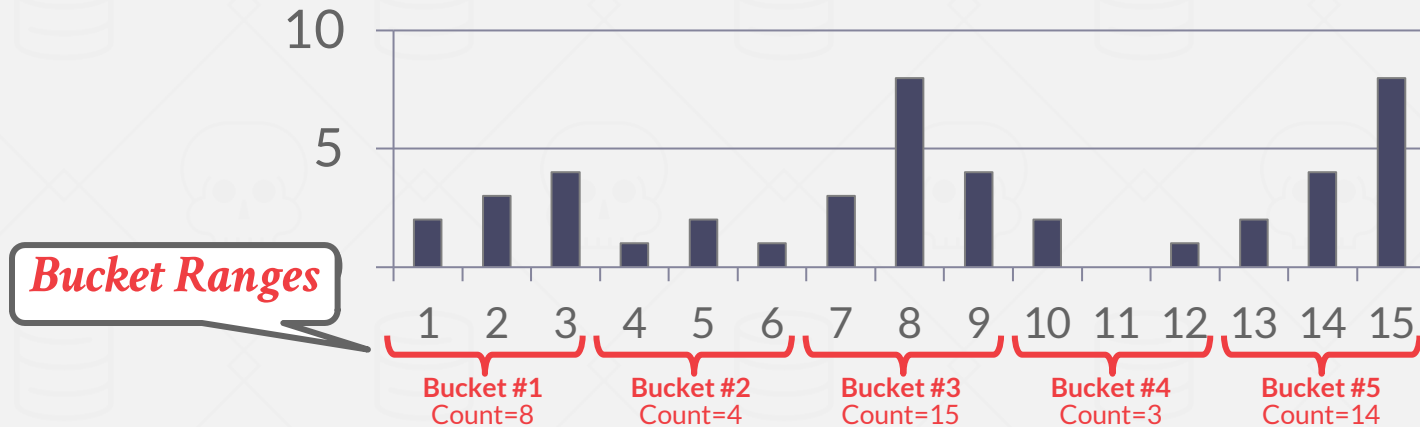
15 Keys \times 32-bits = 60 bytes

Distinct values of attribute

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).

Non-Uniform Approximation



EQUI-DEPTH HISTOGRAMS

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

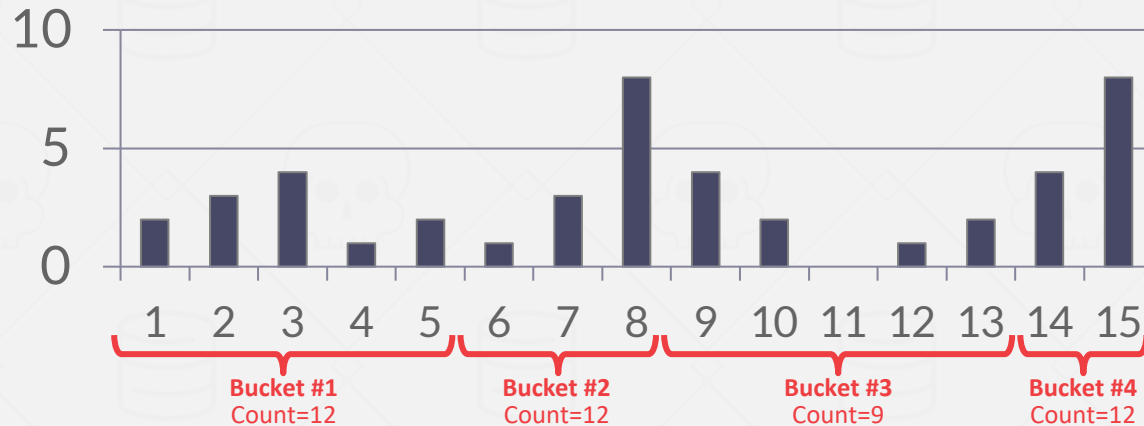
Histogram (Quantiles)



EQUI-DEPTH HISTOGRAMS

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

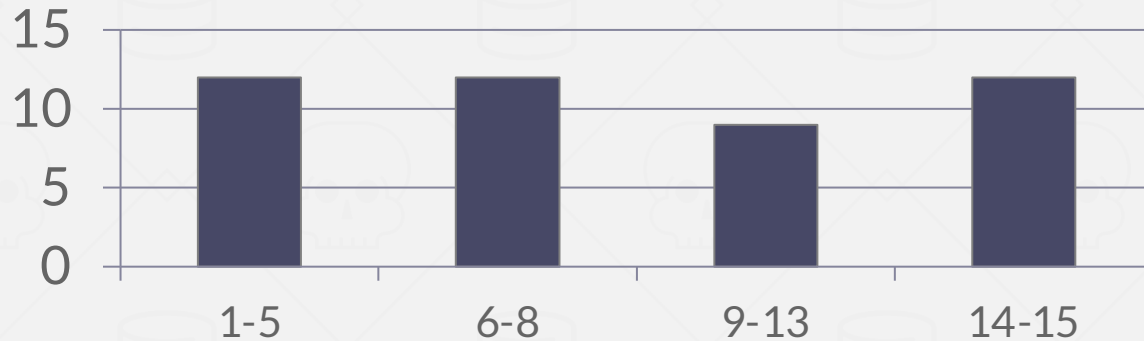
Histogram (Quantiles)



EQUI-DEPTH HISTOGRAMS

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

Histogram (Quantiles)



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.

Table Sample

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

$$\text{sel}(\text{age} > 50) = 1/3$$

```
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

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.

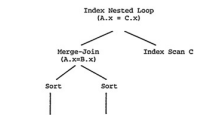
Essential Accepted Papers

An Overview of Query Optimization in Relational Systems

Surajit Chaudhuri
 Microsoft Research
 One Microsoft Way
 Redmond, WA 98052
 +1425-703-1938
 surajit@microsoft.com

1. OBJECTIVE

There has been extensive work in query optimization since the early 70's. It is hard to explore the breadth and depth of this large body of work in a short article. Therefore, I have decided to focus primarily on the optimization of SQL queries in relational database systems and present my biased and incomplete view of this field. The goal of this article is not to be comprehensive, but rather to explain the foundations and present samplings of significant work in this area. I would like to apologize to the many contributors in this area whose work I have failed to explicitly acknowledge due to oversight or lack of space. I take a liberty of stating technical precision for ease of presentation.



2. INTRODUCTION

Relational query languages provide a high-level "declarative" interface to access data stored in relational databases. Over time, SQL [14] has emerged as the standard for relational query languages. Two key components of the query evaluation component of a SQL database system are the query optimizer and the query execution engine.

The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of a SQL query as input and is responsible for generating an efficient execution plan for the given SQL query from the space of possible execution plans. The task of an optimizer is essentially to solve, for a given SQL query, there can be a large number of possible operator trees:

- The query execution engine implements a set of physical operators. An operator takes an input or more data streams and produces an output data stream. Examples of physical operators are (external) join, sequential scan, index scan, nested-loop join, and sort-merge join. It is clear that some operators as physical operators since they are not necessarily tied one-to-one with relational operators. The simplest way to think of physical operators is as pieces of code that are used to building blocks to make possible the execution of SQL operators. An abstract representation of such an execution is a physical operator tree, as illustrated in Figure 1. The edges in an operator tree represent the data flow among the physical operators. We use the terms physical operator tree and execution plan (or, simply plan) interchangeably. The execution engine is responsible for the execution of the plan that results in generating answers to the query. Therefore, the capabilities of the query execution engine determine the structure of the operator trees that are feasible. We refer the reader to [20] for an overview of query optimizer techniques.
- A space of plans (search space).
- A cost estimation technique to that a cost can be assigned to each plan in the search space. Intuitively, this is an estimation of the resources needed for the execution of the plan.
- An enumeration algorithm that can search through the execution space.

Permission to make digital or hard copies of all or part of this material is granted provided that the copies are not made for distribution, for advertising or promotional purposes, for creating new collective works, or for resale or resale for profit or commercial advertising and that the name of this conference and the full citation on this page appear in the copyright notice. For more information, contact the ACM Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, USA. Copyright 1993, 1995 ACM 0897-9175/93 \$6.50 \$1.00

The Volcano Optimizer Generator: Extensibility and Efficient Search

Goetz Graefe
 Portland State University
 graefe@cs.pdx.edu

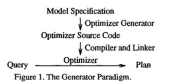
William J. McKenna
 University of Colorado at Boulder
 bill@cs.colorado.edu

Abstract
Emerging database application domains demand not only new functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific databases. Data model, logical algebra, physical algebra, and optimization rules are translated by the optimizer generator into optimizer source code. Compared with other earlier EXODUS optimizer generator prototypes, the physical engine is more extensible and powerful. It provides efficient support for non-normal cost models and for physical requesters such as sort orders. At the same time, it is much more efficient as it combines dynamic programming, which until now had been used only for relational table-join optimization, with goal-directed search and branch-and-bound pruning. Compared with other rule-based optimization systems, it provides complete data model independence and more natural extensibility.

1. Introduction

While extensibility is an important goal for requirement for many current database research projects and system prototypes, performance must not be sacrificed for two reasons. First, data volumes stored in database systems continue to grow, in many application domains far beyond the capabilities of most existing database systems. Second, in order to overcome acceptance problems in emerging database application areas such as scientific computing, database systems must achieve at least the same performance as the file systems currently in use. Additional software layers for database management must not be cumbersome by database performance advantages normally not used in these application areas. Optimization and parallelization are prime candidates to provide these performance advantages, and tools and techniques for optimization and parallelization are crucial for the wider use of extensible database technology.

Existing database management systems, namely the Volcano extensible, parallel query processor [4], the Relational CODSYS project [11] and optimization and parallelization in scientific databases [20] as well as to assist research efforts by other researchers, we have built a new extensible query optimization system. Our earlier experience with the EXODUS optimizer generator had been inconclusive while it had proved the feasibility of extending the optimizer generator paradigm, it was difficult to construct efficient, production-quality optimizers. Therefore, we designed a new optimizer generator, requiring several important improvements over the EXODUS prototype.



Access Path Selection in a Relational Database Management System

P. Griffiths Delinger
 W. M. Archibald
 D. C. Chamberlin
 T. A. Lorie
 T. G. Price

IBM Research Division, San Jose, California 95133

Abstract
In a high level query and data manipulation language such as SQL, requests are made non-procedurally, without reference to access paths. This paper describes how simple (single relation) and complex (join) queries are translated into a user specification of desired data as a Boolean expression of predicates. System R is an experimental database management system which does the job of the optimizer in the processing of SQL statements, and section 3 will describe the internal components physically stored table. In section 4 the optimizer core formulas are introduced for single table queries; and section 5 discusses the joining of two or more tables, and their corresponding costs. Nested queries (queries in predicates) are covered in section 6.

1. Introduction

System R is an experimental database management system based on the relational algebra of data which has been under development at the IBM San Jose Research Laboratory since 1973 [4]. The software was developed as a research vehicle in relational databases and is not generally available outside the IBM Research Division.

This paper assumes familiarity with relational data model terminology as described in Code #70 and data #2. The user interface in System R is the unified query, data definition, and manipulation language SQL [5]. Statements in SQL can be issued both from an online, casual-user-oriented, menu-driven interface and from programming languages such as C, PL, and COBOL.

In System R a user need not know how the tuples are physically stored. The user can access paths are optionally used, which allow the user to specify anything about the access paths to be used. For tuple retrieval, a predicate may have one or more join-join rules to be performed. The System R optimizer chooses both join order and access path for each table in the SQL statement.

Copyright © 1979 by the ACM, Inc. used by permission. Permission to make digital or hard copies of all or part of this material is granted provided that the copies are not made for distribution, for advertising or promotional purposes, for creating new collective works, or for resale or resale for profit or commercial advertising and that the name of this conference and the full citation on this page appear in the copyright notice. For more information, contact the ACM Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, USA. Copyright 1993, 1995 ACM 0897-9175/93 \$6.50 \$1.00

Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers

Umeshwar Dayal
 Computer Corporation of America
 Information Systems Center
 Cambridge, Massachusetts 02142-1489

Abstract
Existing query optimizers focus on Relational-Project-Join queries. In practice, however, query languages such as SQL and DAPLEX have many powerful features (e.g. control over duplicate, nested subqueries, grouping, aggregates, and quantifiers) that are not expressible as sequences of Relational, Project, and Join operations. Existing optimizers are severely limited in their strategies for processing such queries, typically they use only tuple substitution, and process nested subquery blocks top-down. Tuple substitution, however, is generally inefficient and especially so when the database is distributed. Hence, it is imperative to develop alternative strategies. This paper introduces new operators for these difficult features, and describes implementation methods for them. From the algebraic properties of these operators, new query processing tactics are derived. It shows how these new tactics can be deployed to greatly increase the space of interesting strategies for optimization, without seriously affecting the architecture of existing optimizers. The contribution of the paper is in demonstrating the feasibility and desirability of developing an integrated framework for optimizing all of SQL or other query languages that have similar features.

Consider, for example, the following relations:
 EMP (Emp#, Name, Dept#, Sal)
 DEPT (Dept#, Name, Loc, Mgr)
 and the following SQL query, which contains a nested subquery block:

```

  Query 1
  SELECT E.Name
  FROM EMP E
  WHERE E.Dept# IN
  (SELECT D.Dept#
  FROM DEPT D
  WHERE D.Loc = 'Dusser' AND
  D.Mgr = D.Mgr)
  
```

1. Introduction

Most research on query optimization has focused on conjunctive queries, i.e. queries that can easily be translated into Relational-Project-Join expressions of the relational algebra [CODD70]. However, practical query languages, such as SQL and DAPLEX, have many powerful features (e.g. control over duplicate values; E. Dept# in the list, then E.Name is inserted into the result. The system R optimizer follows the description quite literally, optimizing only the execution of the inner block (after the substitution, the inner block contains two relations and the optimizer does some nested SQL queries for efficiently evaluating them) [RELF79].

In [RELM81], Kim showed that some nested SQL queries could be transformed into equivalent "canonical" queries that did not contain nested subqueries. For example, query 1 could be transformed into query 2 (the queries are not strict equivalents, but more on this later):

```

  Query 2
  SELECT E.Name
  FROM EMP E
  WHERE E.Dept# = D.Dept# AND
  D.Loc = 'Dusser' AND
  D.Mgr = D.Mgr
  
```

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. ICFR 1993: 39-49

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 Query Triggers. VLDB 1987: 197-208

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