Carnegie Mellon University

DATABASE SYSTEMS

Query Planning – Pt.1

LECTURE #15 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO

ADMINISTRIVIA

Mid-term Exam grades posted
→ Come to Andy's OH to view your grade and solution.

Homework #4 is due Sunday Nov 2nd @ 11:59pm

Project #3 is due Sunday Nov 16th @ 11:59pm
→ Recitation Tuesday Oct 28th @ 8:00pm (see @195)

UPCOMING DATABASE TALKS

SingleStore (DB Seminar)
→ Monday Oct 27th @ 4:30pm
→ Zoom

Delta Lake (DB Seminar)
→ Monday Nov 3rd @ 4:30pm
→ Zoom

Apache Pinot @ Uber (DB Group)
→ Tuesday Nov 4th @ 12:00pm
→ GHC 8115

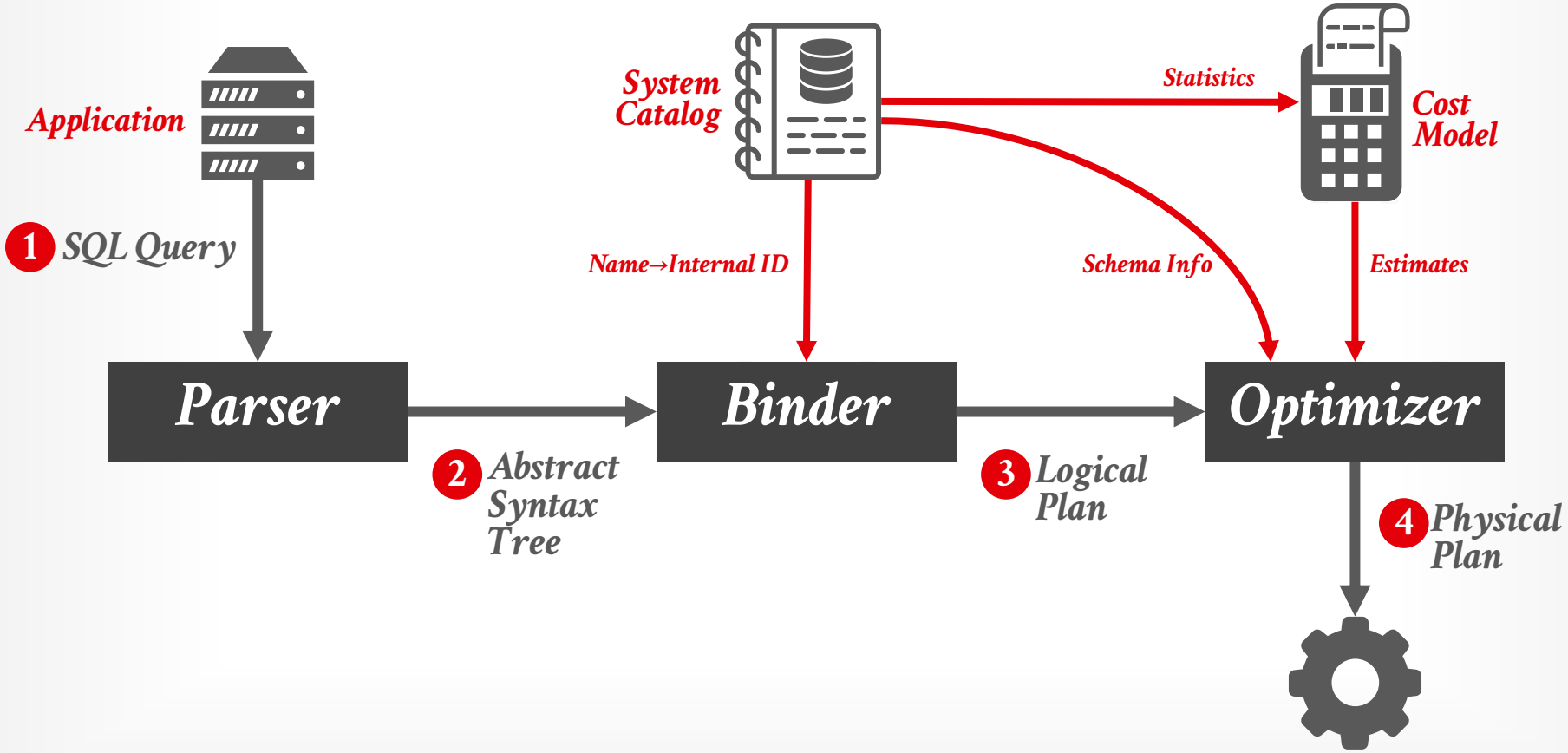
LAST CLASS

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?

DBMS OVERVIEW



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)

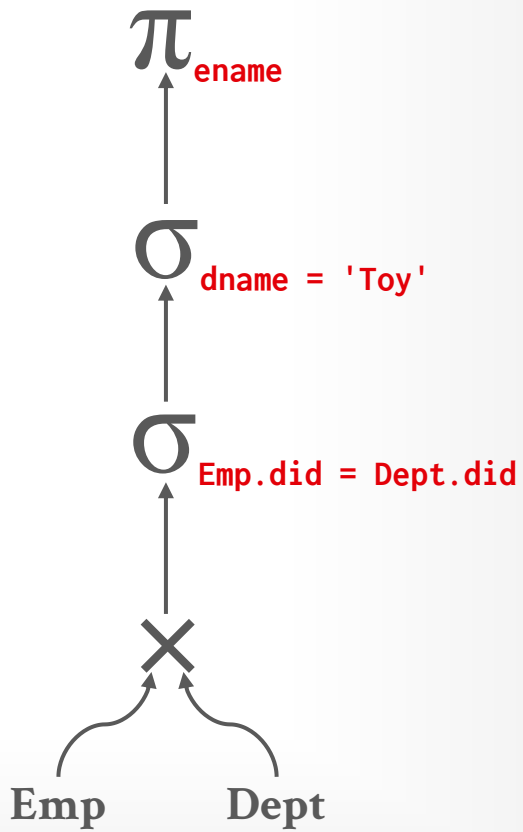
10,000 records
1,000 pages

---

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(ssn,ename,addr,sal,did)

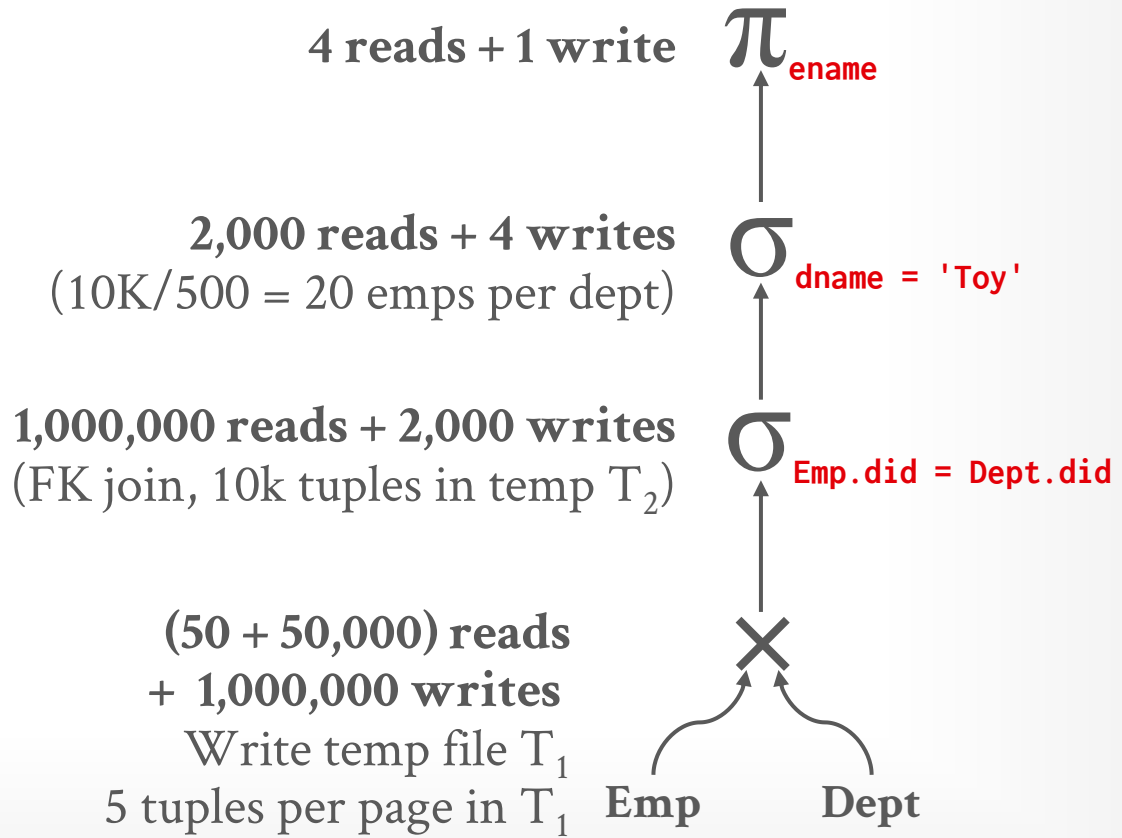
10,000 records
1,000 pages

clustered unclustered

Dept(did,dname,floor,mgr)

500 records
50 pages

Total: 2M I/Os



MOTIVATION

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

Catalog

clustered *unclustered* *unclustered*

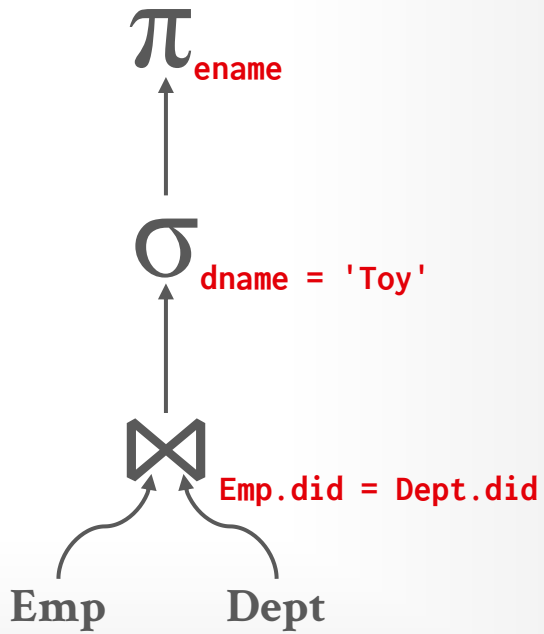
Emp(ssn,ename,addr,sal,did)

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(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 T_2

$\pi_{\texttt{ename}}$

2,000 reads + 4 writes
Read temp T_1, Write temp T_2

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

(50 + 50,000) reads
+ 2,000 writes
Page Nested-Loop Join
Write Temp T_1

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

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)

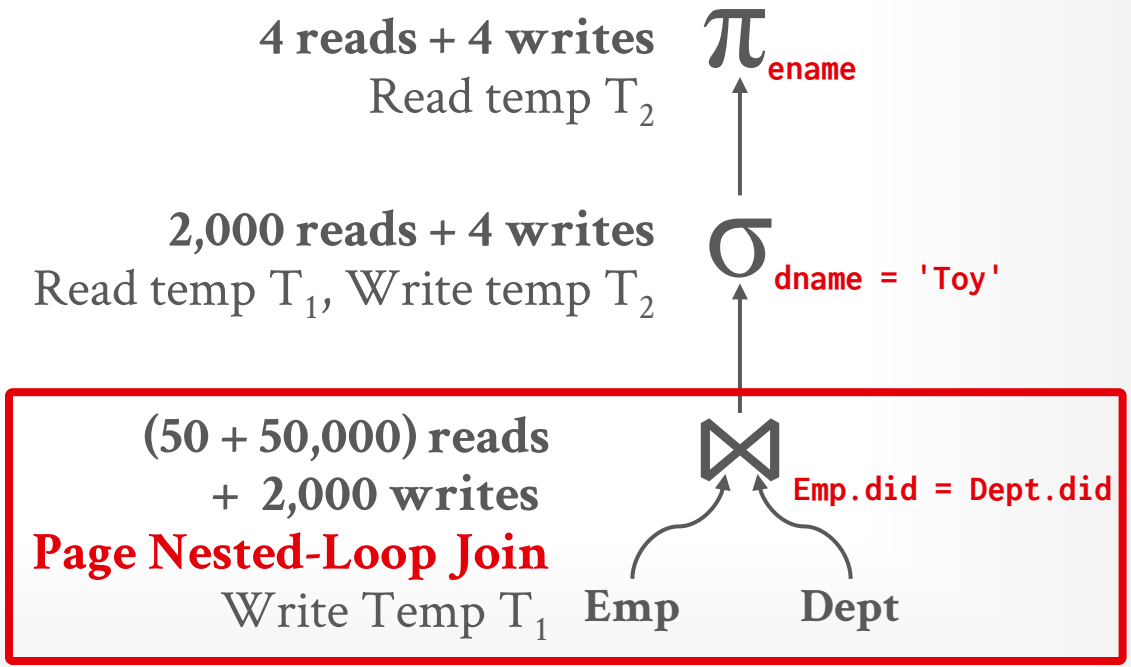
10,000 records
1,000 pages

clustered unclustered

Dept(did,dname,floor,mgr)

500 records
50 pages

Total: 54k I/Os



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: 7,159 I/Os

4 reads + 4 writes
Read temp T_2

$\pi_{\texttt{ename}}$

2,000 reads + 4 writes
Read temp T_1, Write temp T_2

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

$3\times(|Emp| + |Dept|) =$
3,150 reads + 2,000 writes
Sort-Merge Join (50 Buffers)
Write Temp T_1

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

Emp Dept

MOTIVATION

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

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*

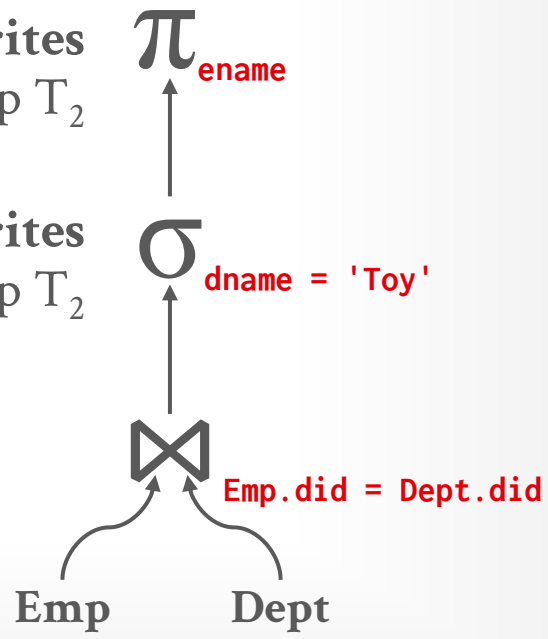
Dept(did,dname,floor,mgr)

500 records
50 pages

4 reads + 4 writes
Read temp T_2

2,000 reads + 4 writes
Read temp T_1, Write temp T_2

3×(|Emp| + |Dept|) =
3,150 reads + 2,000 writes
Sort-Merge Join (50 Buffers)
Write Temp T_1



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

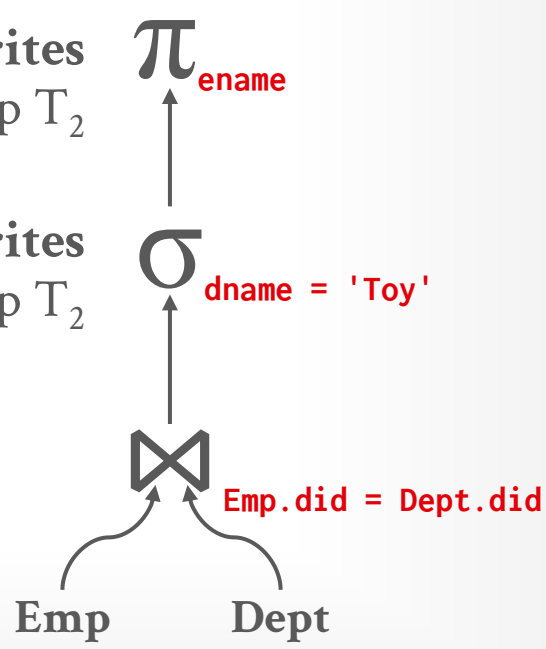
No Pipelining!

Materialization Model → Total: 7,159 I/Os

~~reads + 4 writes~~
Read temp T_2

~~2,000 reads + writes~~
Read temp T_1, Write temp T_2

3×(|Emp| + |Dept|) =
3,150 reads + ~~2,000 writes~~
Sort-Merge Join (50 Buffers)
Write Temp T_1



MOTIVATION

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

Catalog

clustered unclustered unclustered

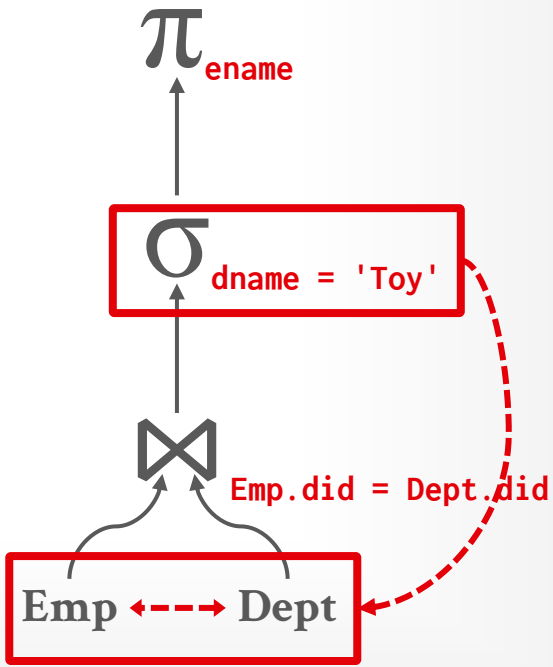
Emp(ssn,ename,addr,sal,did)

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*

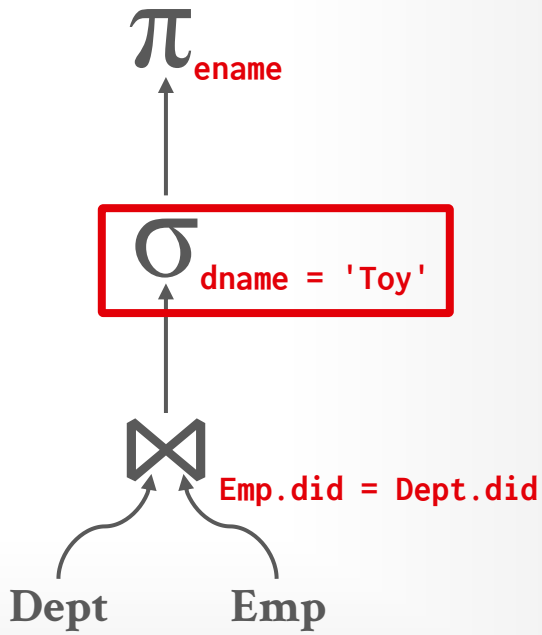
Emp(ssn,ename,addr,sal,did)

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(ssn,ename,addr,sal,did)

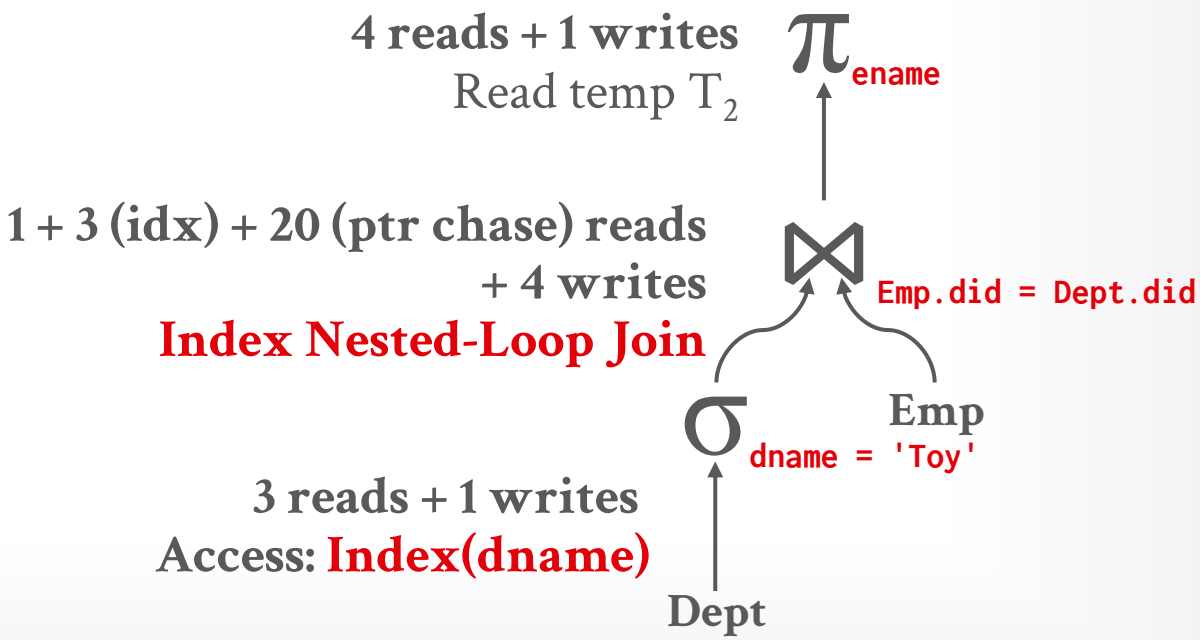
10,000 records
1,000 pages

clustered unclustered

Dept(did,dname,floor,mgr)

500 records
50 pages

Total: 37 I/Os



TODAY'S AGENDA

Background

Transformations

Heuristic / Ruled-based Optimization

Cost-based Optimization

QUERY OPTIMIZER

Given a query's **logical plan** as input, generate a semantically equivalent **physical execution plan**.

→ May have to consider a large search space of promising plans
→ Accurately determine whether one potential plan is better than another.
→ Efficiently search the solution space to find a physical plan with the lowest cost.

Ideally an optimizer should generate the best plan regardless of how the query is expressed.

LOGICAL VS. PHYSICAL PLANS

The optimizer applies transformations that map 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.

OPTIMIZATION GRANULARITY

Choice #1: Single Query

→ Much smaller search space.
→ DBMS (usually) does not reuse results across queries.
→ To account for resource contention, the cost model must consider what is currently running.

Choice #2: Multiple Queries

→ More efficient if there are many similar queries.
→ Search space is much larger.
→ Useful for data / intermediate result sharing.

OBSERVATION

We now have a high-level understanding of a query optimizer's role in a DBMS.

The quality of the plans that an optimizer generates is mostly based on three factors:

→ Transformations / Enumeration
→ Search Algorithm
→ Cost Model

TRANSFORMATIONS

Enumerate the different choices / forms of a query plan that are semantically equivalent and logically correct.

→ Need to ensure new query plans produces the same result as the original no matter the inputs.

The goal of each transformation is to:

→ Lower query execution cost.

→ Unlock additional transformations.

Exploit **relational algebra equivalencies** via query and database contents (logical + physical).

RELATIONAL ALGEBRA EQUIVALENCES

Two relational algebra expressions are equivalent if they generate the same set of tuples.

These equivalences allow the DBMS to manipulate and transform a query plan into different forms without effecting the correctness of its output.

→ This is how a heuristic-based optimizer identifies better query plans without a cost model.

RELATIONAL ALGEBRA EQUIVALENCES

Selections:

→ Perform filters as early as possible.

→ Breakup a complex predicate into conjunctive clauses and push down to lowest part of plan as possible.

$$\sigma_{p1 \wedge p2 \wedge \ldots pn}(R) = \sigma_{p1}(\sigma_{p2}(\ldots \sigma_{pn}(R)))$$

Simplify complex predicates:

→ `X=3 AND Y=X` → `X=3 AND Y=3`

→ `X=1+1` → `X=2`

→ `X=YEAR('10/27/2025')` → `X=2025`

RELATIONAL ALGEBRA EQUIVALENCES

Joins:

→ Commutative:

$$R \bowtie S = S \bowtie R$$

→ Associative:

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Number of join orderings for an ***n***-way binary join is **$(n-1)! \times C(n-1)$**, where **$C(n-1)$** is the **$(n-1)^{th}$** Catalan number.

→ ***n*!** different orders of leaf nodes (original relations)

→ **$C(n-1)$** possible shapes of a full binary tree with ***n*** leaves

TRANSFORMATIONS

Split Conjunctive Predicates

Predicate Pushdown

Replace Cartesian Products with Joins

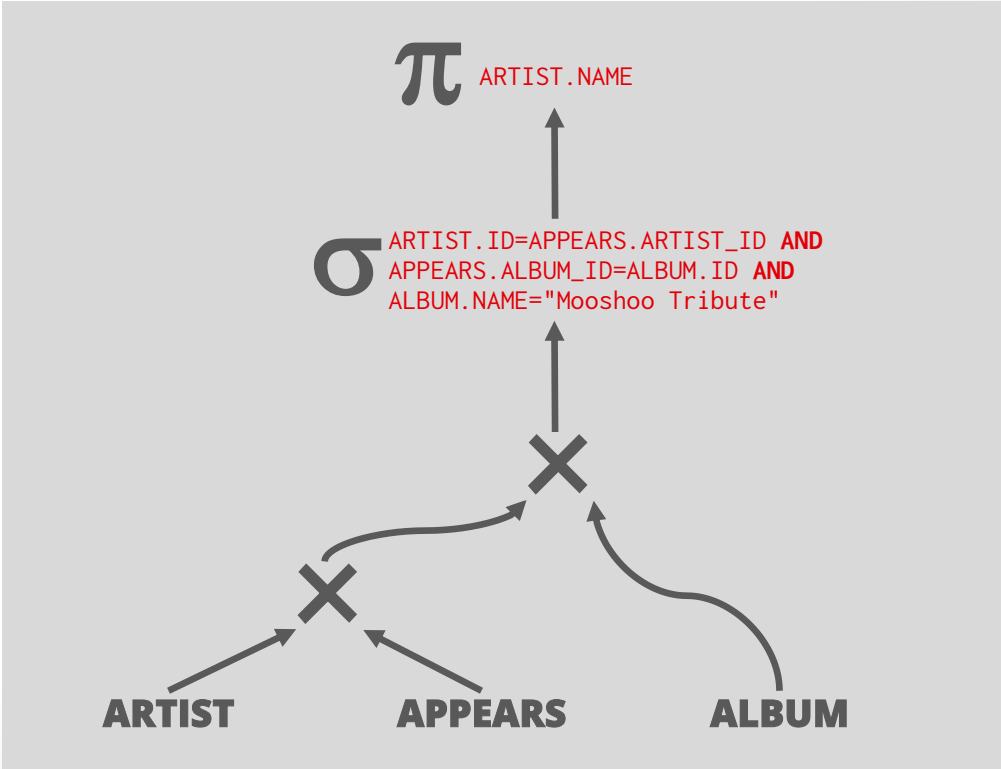
Projection Pushdown

Source: Thomas Neumann

SPLIT CONJUNCTIVE PREDICATES

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

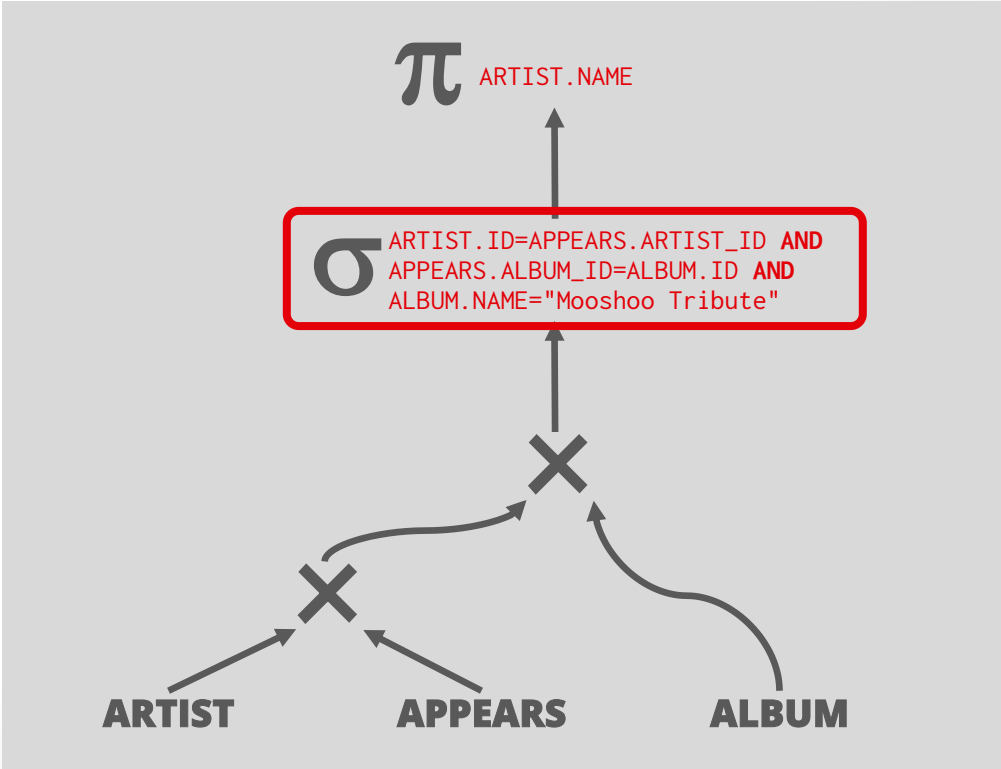
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



SPLIT CONJUNCTIVE PREDICATES

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

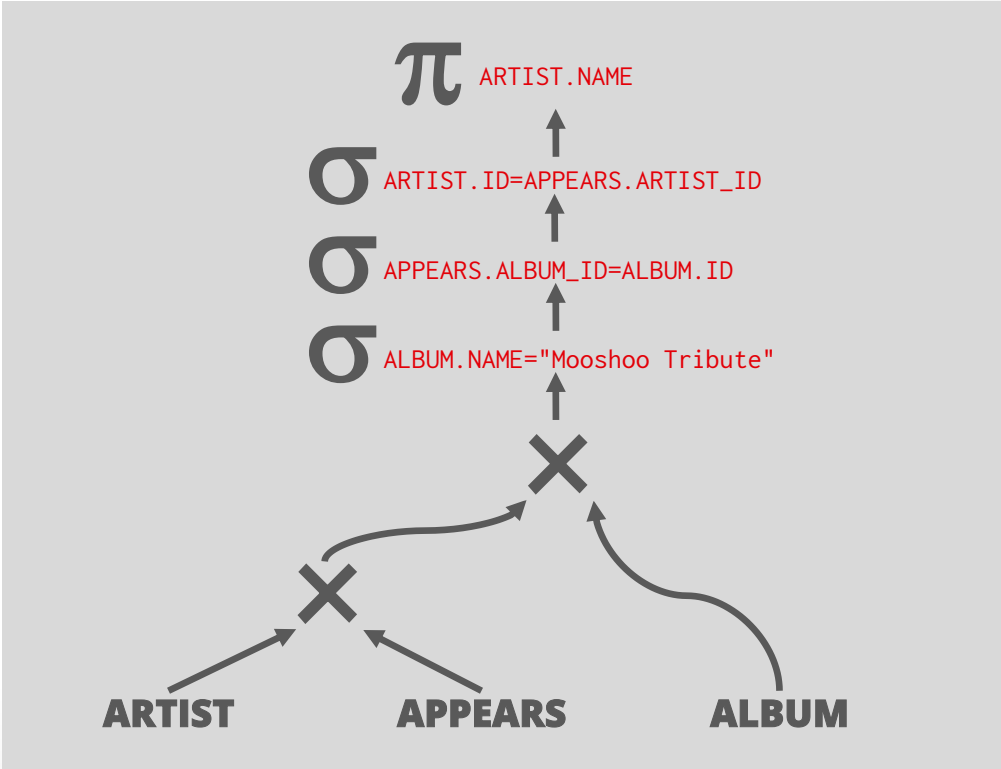
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



SPLIT CONJUNCTIVE PREDICATES

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

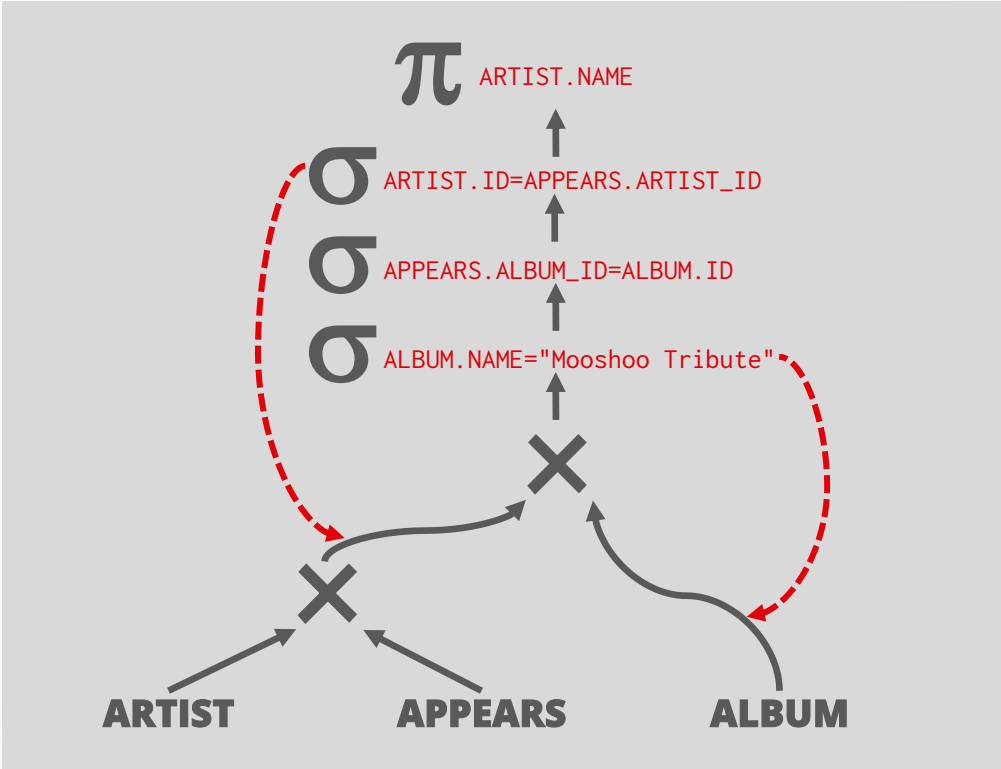
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



PREDICATE PUSHDOWN

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

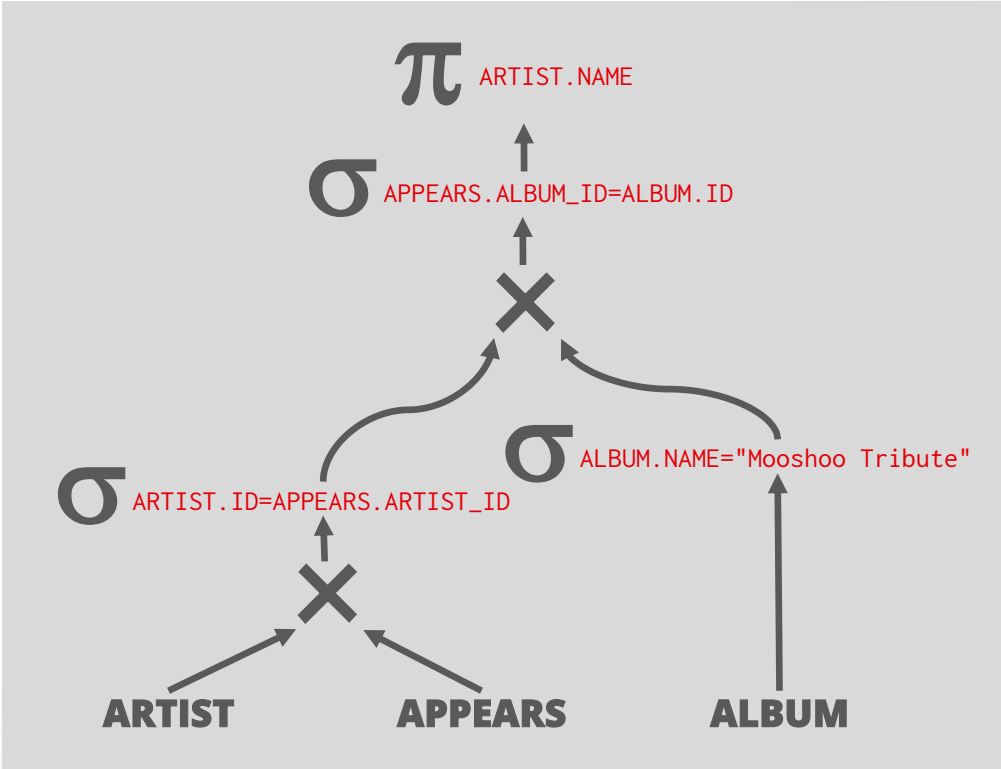
Move the predicate to the lowest point in the plan after Cartesian products.



PREDICATE PUSHDOWN

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

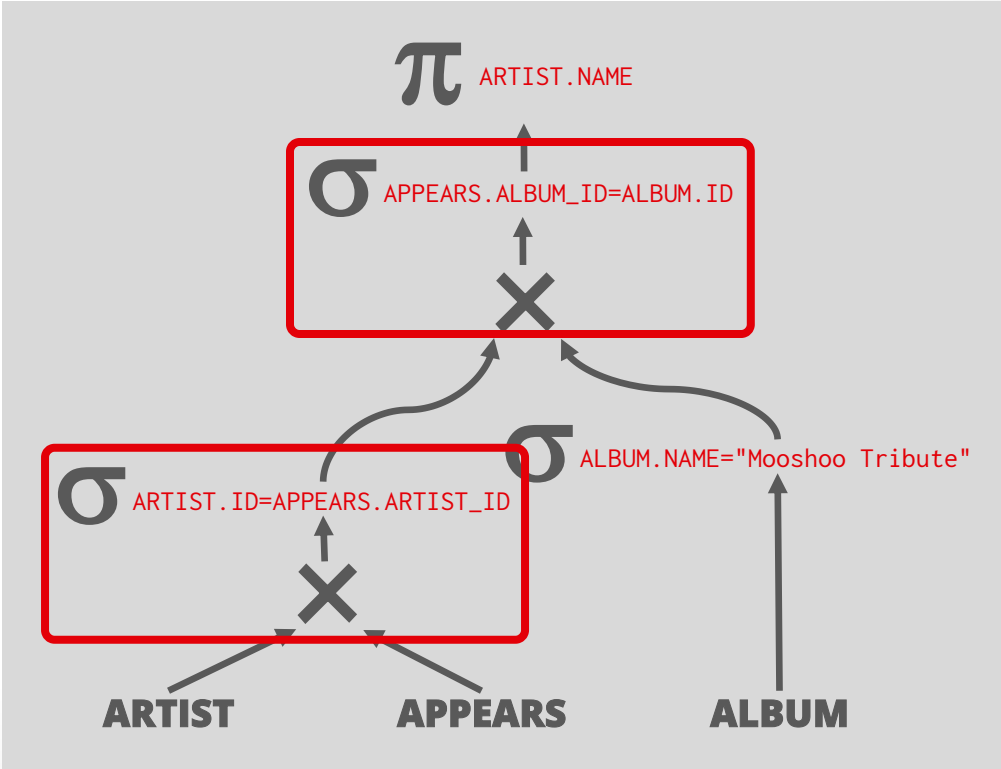
Move the predicate to the lowest point in the plan after Cartesian products.



REPLACE CARTESIAN PRODUCTS

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

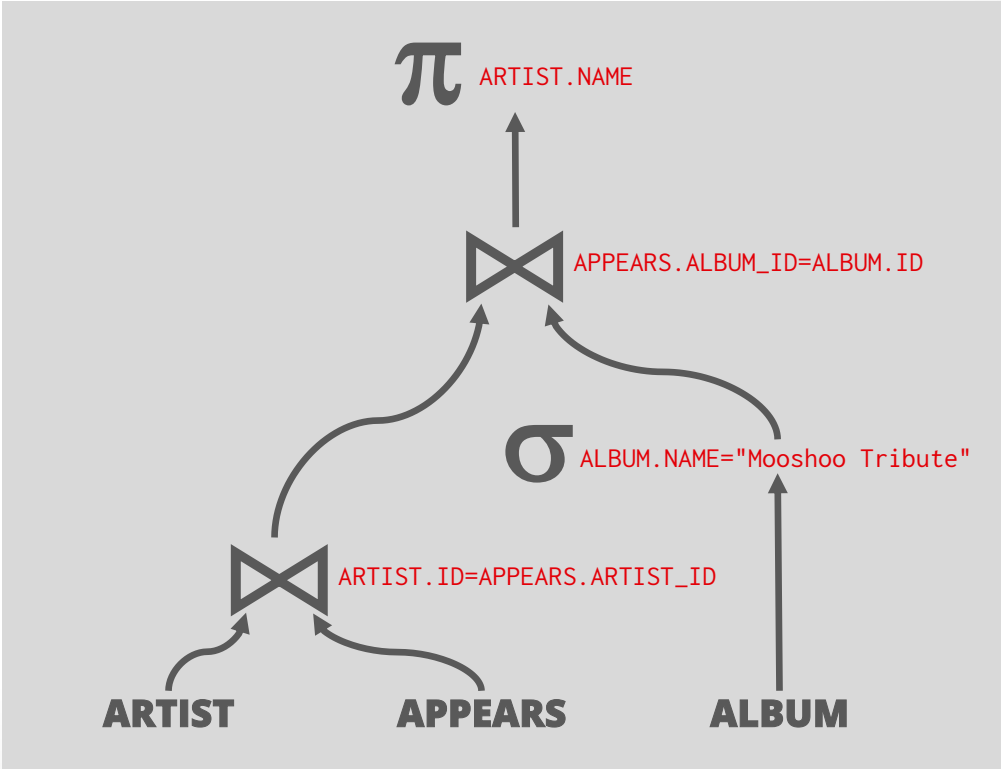
Replace all Cartesian Products with inner joins using the join predicates.



REPLACE CARTESIAN PRODUCTS

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

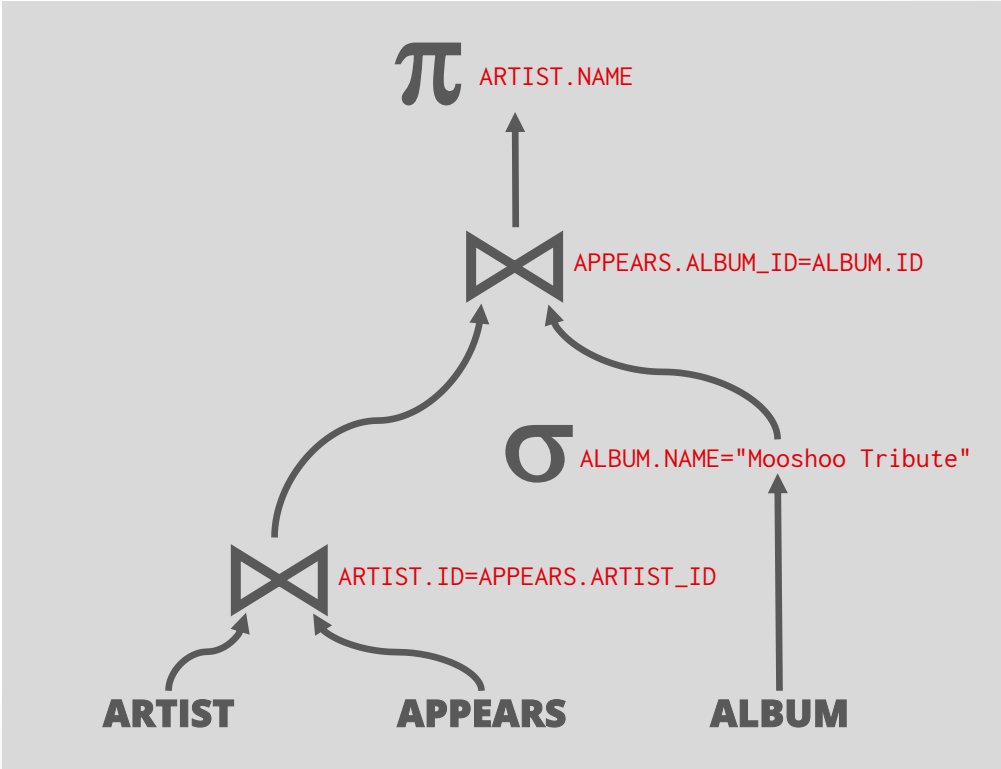
Replace all Cartesian Products with inner joins using the join predicates.



PROJECTION PUSHDOWN

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

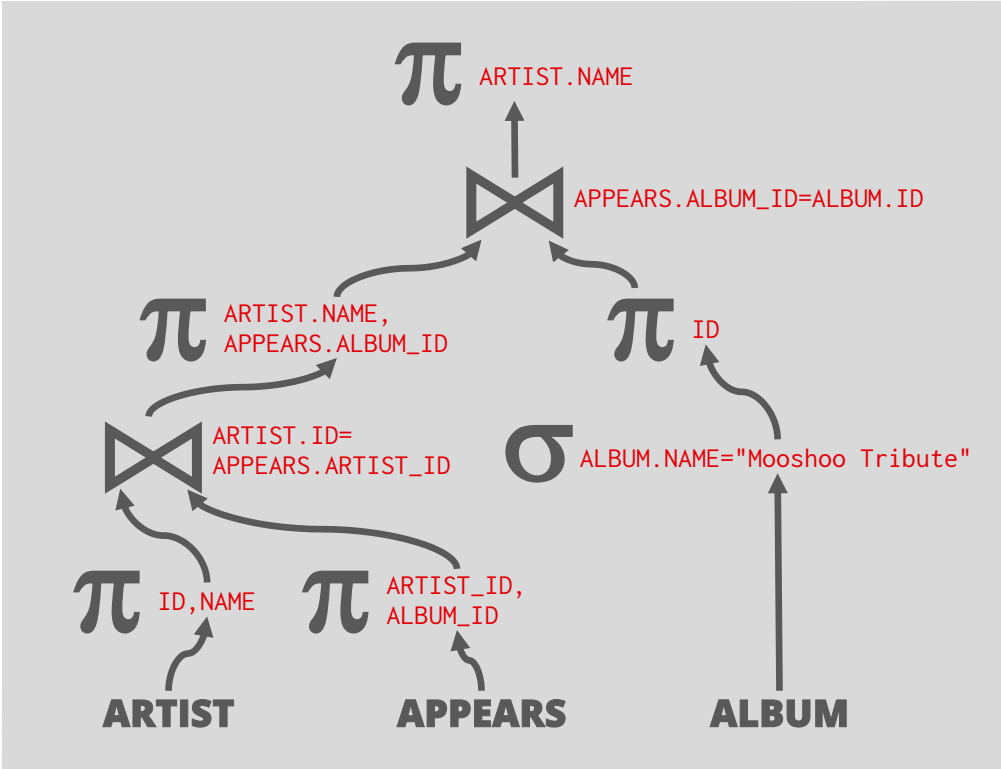
Eliminate redundant attributes before pipeline breakers to reduce materialization cost.



PROJECTION PUSHDOWN

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

Eliminate redundant attributes before pipeline breakers to reduce materialization cost.



SEARCH ALGORITHMS

Given a set of transformation rules, the optimizer searches for a good physical plan for a given query.

At search time, the optimizer will have a query's logical plan but it may not have all information available.

→ Prepared statements with input variables
→ Missing statistical information.

SEARCH ALGORITHMS

Heuristics / Rules

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

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.

HEURISTIC-BASED OPTIMIZATION

Define static rules that transform logical operators to a physical plan without a cost model.

→ Perform most restrictive selection early
→ Perform all selections before joins
→ Predicate/Limit/Projection pushdowns
→ Join ordering based on simple rules or cardinality estimates

Examples: INGRES (until mid-1980s) and Oracle (until early-1990s), MongoDB, most new DBMSs.

HEURISTIC-BASED OPTIMIZATION

Advantages:

→ Easy to implement and debug.
→ Works reasonably well and is fast for simple queries.

Disadvantages:

→ Relies on magic constants that predict the efficacy of a planning decision.
→ Nearly impossible to generate good plans when operators have complex inter-dependencies.

HEURISTIC-BASED OPTIMIZATION

Advantages:

→ Easy to implement and debug.

→ Works reasonably well and is fast fo

Disadvantages:

→ Relies on magic constants that predi decision.

→ Nearly impossible to generate good complex inter-dependencies.

Stonebraker gave the story of the query optimizer as an example. Relational queries were often highly complex. Let's say you wanted your database to give you the name, salary, and job title of everyone in your Chicago office who did the same kind of work as an employee named Alien. (This example happens to come from Oracle's 1981 user guide.) This would require the database to find information in the employee table and the department table, then sort the data. How quickly the database management system did this depended on how cleverly the system was constructed. "If you do it smart, you get the answer a lot quicker than if you do it stupid, Stonebraker said.

He continued. "Oracle had a really stupid optimizer. They did the query in the order that you happened to type in the clauses. Basically, they blindly did it from left to right. The Ingres program looked at everything there and tried to figure out the best way to do it." But Ellison found a way to neutralize this advantage, Stonebraker said. "Oracle was really shrewd. They said they had a syntactic optimizer, whereas the other guys had a semantic optimizer. The truth was, they had no optimizer and the other guys had an optimizer. It was very, very, very creative marketing. . . . They were very good at confusing the market."

"What he's using is semantics himself," Ellison said. Just because Oracle did things differently, "Stonebraker decided we didn't have an optimizer. [He seemed to think] the only kind of optimizer was his optimizer, and our approach to optimization wasn't really optimization at all. That's an interesting notion, but I'm not sure I buy that."

SEARCH ALGORITHMS

Heuristics / Rules

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

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

Apply transformation rules to enumerate different variations of a query's plan estimate their costs to guide the search process.

→ Single relation.
→ Multiple relations.
→ Nested sub-queries.

The optimizer chooses the best plan it has seen for the query until it reaches a search termination condition.

SEARCH TERMINATION

Approach #1: Wall-clock Time

→ Stop after the optimizer runs for some length of time.

Approach #2: Cost Threshold

→ Stop when the optimizer finds a plan that has a lower cost than some threshold.

Approach #3: Exhaustion

→ Stop when there are no more enumerations of the target plan. Usually done per sub-plan/group.

Approach #4: Transformation Count

→ Stop after a certain number of rules/transformations have been considered.

ACCESS PATH TRANSFORMATION

The optimizer chooses the access method(s) for those relations that minimizes the cost of retrieving a query's requested data from base relations.

→ Can also optimize predicate evaluation ordering.

Cost of access method depends on several factors:

→ Selectivity of predicate
→ Data structures (e.g., B+Tree vs. Hash Table)
→ Sort order of the table / index
→ Data accoutrements (e.g., **INCLUDE**, zone maps)
→ Compression / encoding

SINGLE-RELATION QUERY PLANNING

Generate multiple alternatives for retrieving data from a base relation for a given expression.

Available alternatives depend on query, database logical schema, and DBMS implementation.

→ Example: A rule determines whether an index qualifies based on a query's predicates (e.g., partial indexes).

Sequential Scan is always the fallback option.

→ Often worst choice in row stores but it is sometimes the only choice in column stores.

SINGLE-RELATION QUERY PLANNING

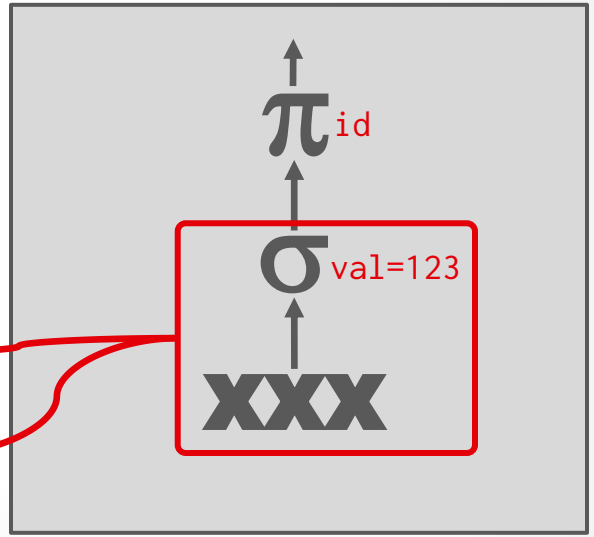
Access path selection for a single relation query is (relatively) easy because they are **sargable**.

Search Argument Able

```
SELECT id
  FROM xxx
 WHERE val >= 123
   AND val <= 456;
```

Pick the best access method (sequential scan vs. index) using a simple cost model.

```
CREATE TABLE xxx (
  id INT PRIMARY KEY,
  val INT
);
CREATE INDEX ON xxx (val);
```



MULTI-RELATION QUERY PLANNING

Approach #1: Bottom-Up / Forward Chaining

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

Approach #2: Top-Down / Backward Chaining

→ 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

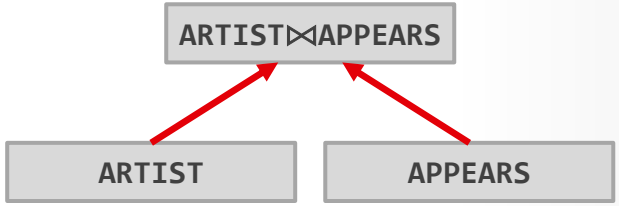
FORWARD VS. BACKWARD CHAINING

FORWARD VS. BACKWARD CHAINING

Bottom-Up / Forward Chaining:

→ Start from query plan roots, trigger all rules that match those operators, and adds their conclusion to the known facts. Repeats until full query is generated.

→ Breadth-first Search.

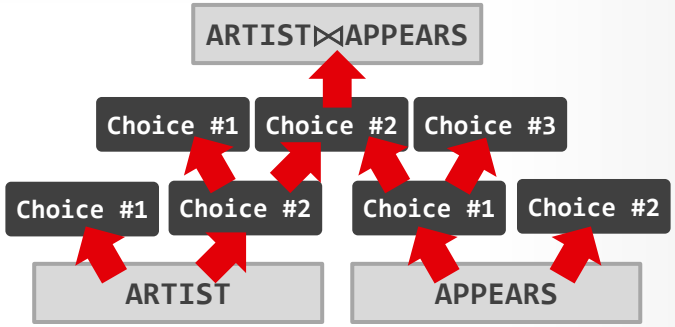


FORWARD VS. BACKWARD CHAINING

Bottom-Up / Forward Chaining:

→ Start from query plan roots, trigger all rules that match those operators, and adds their conclusion to the known facts. Repeats until full query is generated.

→ Breadth-first Search.

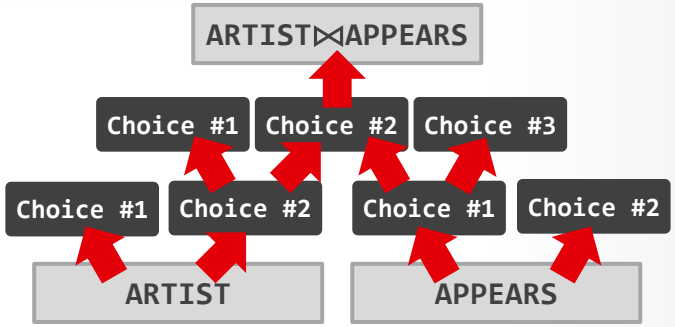


FORWARD VS. BACKWARD CHAINING

Bottom-Up / Forward Chaining:

→ Start from query plan roots, trigger all rules that match those operators, and adds their conclusion to the known facts. Repeats until full query is generated.

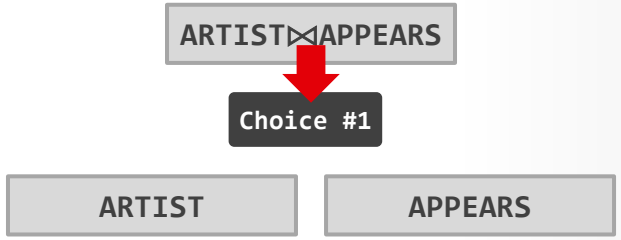
→ Breadth-first Search.



Top-Down / Backward Chaining:

→ Start from the query result and works backward to determine what operators to add to the query plan to achieve result.

→ Depth-first Search.

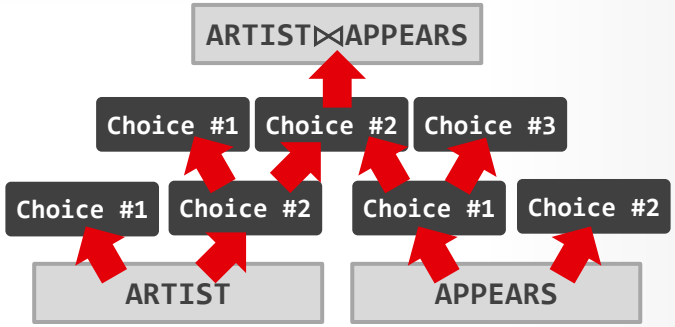


FORWARD VS. BACKWARD CHAINING

Bottom-Up / Forward Chaining:

→ Start from query plan roots, trigger all rules that match those operators, and adds their conclusion to the known facts. Repeats until full query is generated.

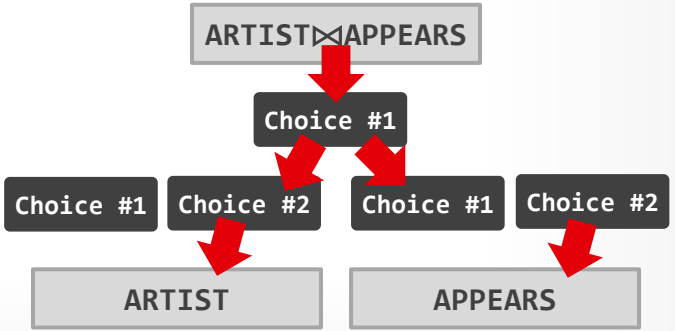
→ Breadth-first Search.



Top-Down / Backward Chaining:

→ Start from the query result and works backward to determine what operators to add to the query plan to achieve result.

→ Depth-first Search.



OBSERVATION

The optimizer can detect whether a query is targeting a database with a common design pattern and invoke transformations that push a query plan into an ideal form.

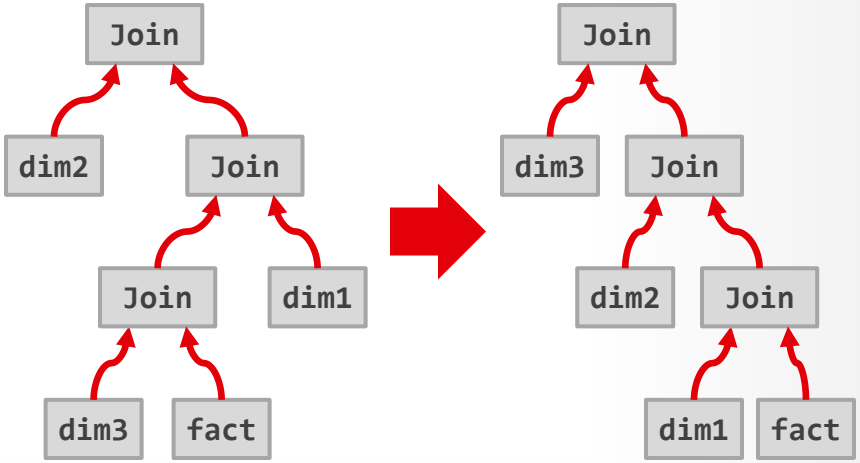
We saw this before with sargable queries where the optimizer can immediately select the best index.

STAR / SNOWFLAKE QUERIES

If a query joins a fact table with multiple dimension tables, then transform it to a left/right-deep join tree and order dimension tables from most to least selective.

Avoid wasting time exploring bushy plans or alternative join orderings for dimension tables.

```
SELECT * FROM fact AS F
  JOIN dim1 ON F.d1 = dim1.id
  JOIN dim2 ON F.d2 = dim2.id
  JOIN dim3 ON F.d3 = dim3.id;
```



Source: EQOP Book

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, Germans, DuckDB, Postgres, most open-source DBMSs.

SYSTEM R OPTIMIZER

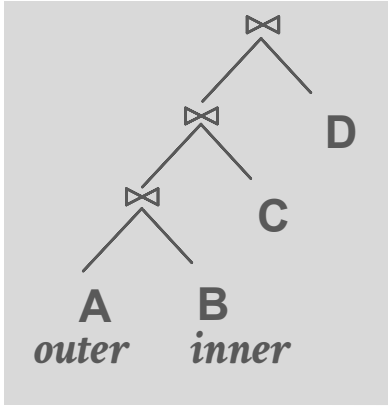
Break query into blocks and generate logical operators for each block.

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

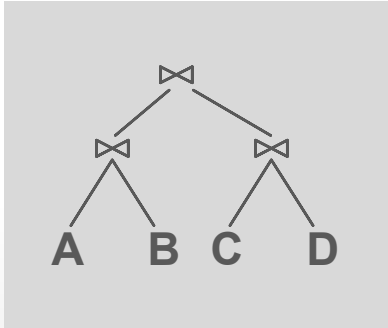
→ All combinations of join algorithms and access paths

If a block accesses multiple relations, iteratively construct a join tree that minimizes the estimated amount of work to execute the plan.

Left-Deep Tree



Bushy Tree



SYSTEM R OPTIMIZER

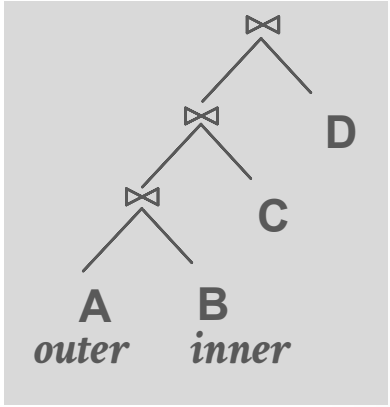
Break query into blocks and generate logical operators for each block.

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

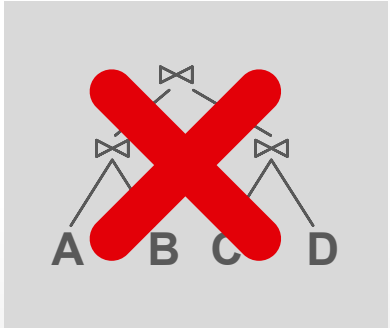
→ All combinations of join algorithms and access paths

If a block accesses multiple relations, iteratively construct a join tree that minimizes the estimated amount of work to execute the plan.

Left-Deep Tree



Bushy Tree



SYSTEM R OPTIMIZER: MULTI-RELATION QUERIES

```
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
```

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

ARTIST: Sequential Scan
APPEARS: Sequential Scan
ALBUM: Index Look-up on **NAME**

SYSTEM R OPTIMIZER: MULTI-RELATION QUERIES

```
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: MULTI-RELATION QUERIES

```
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

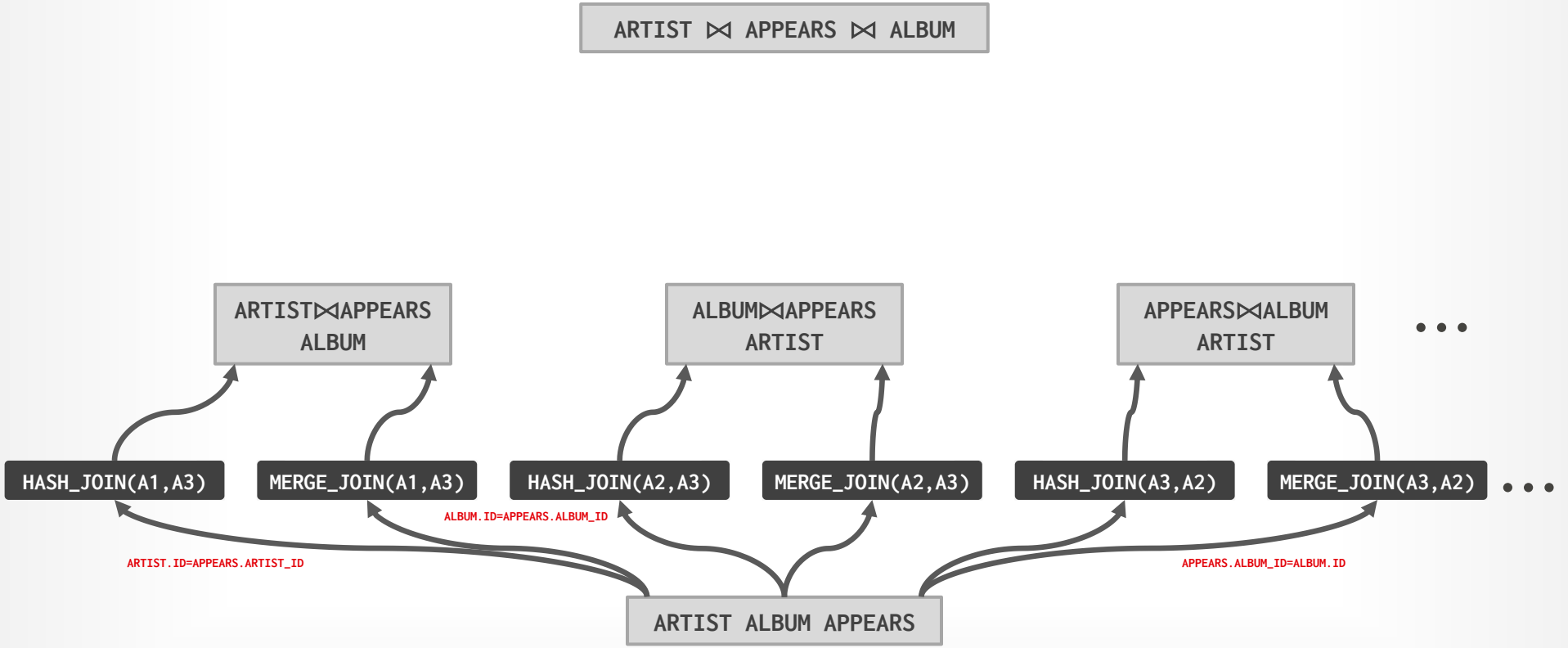
Step #3: Determine the join ordering with the lowest cost

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

Logical Op

Physical Op

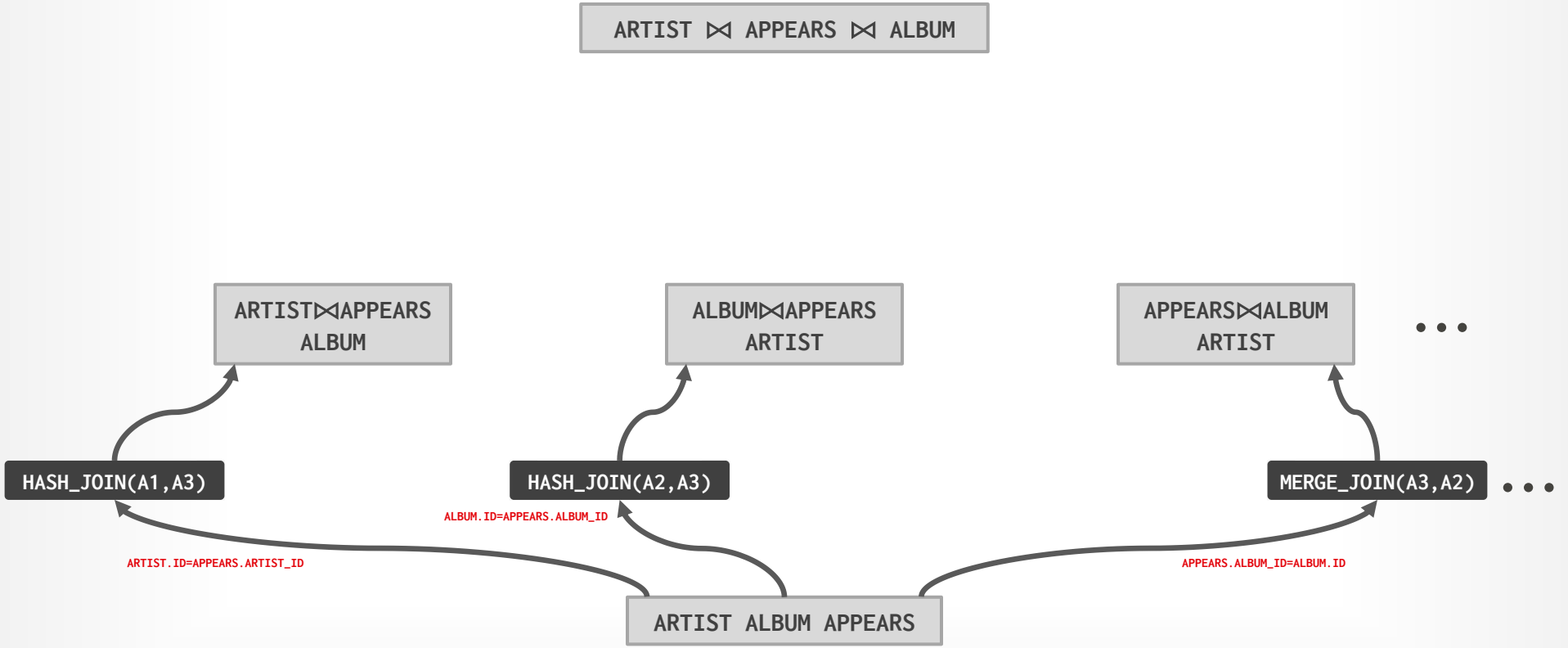
SYSTEM R OPTIMIZER



Logical Op

Physical Op

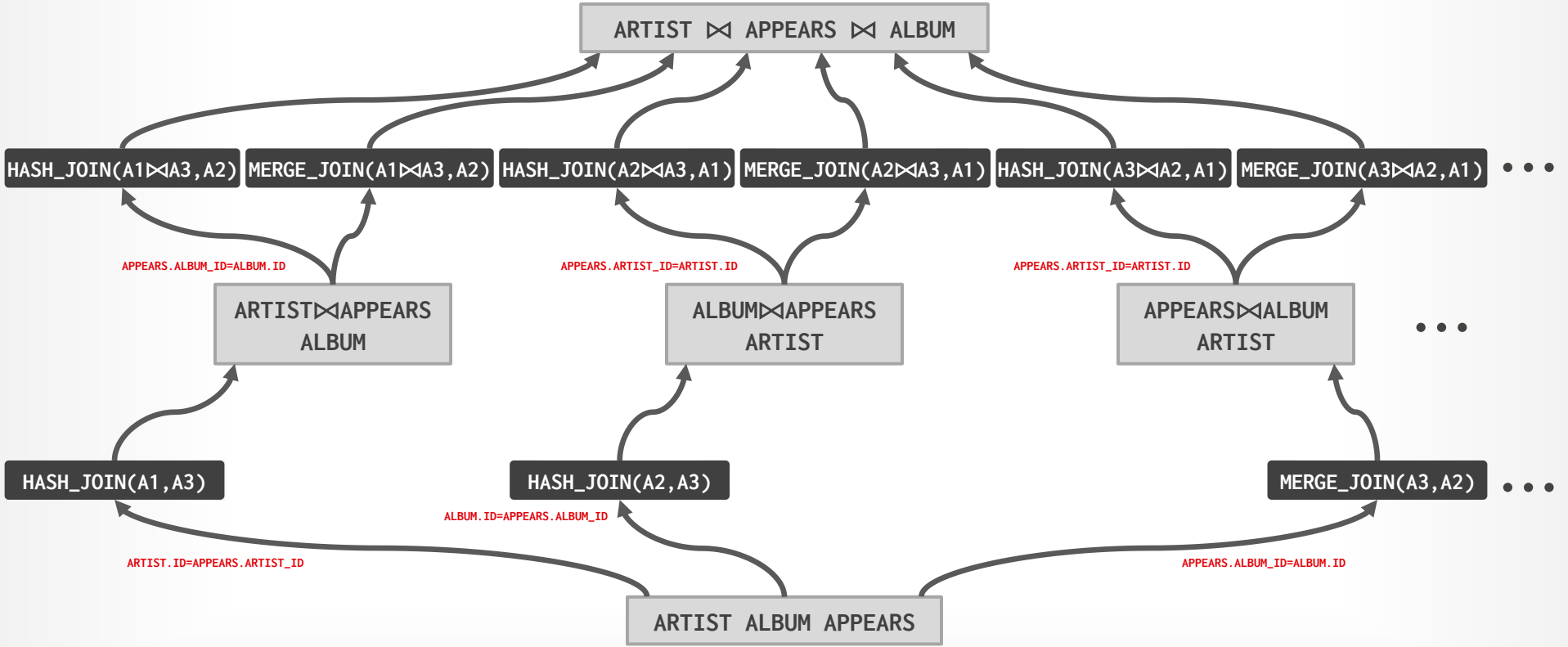
SYSTEM R OPTIMIZER



Logical Op

Physical Op

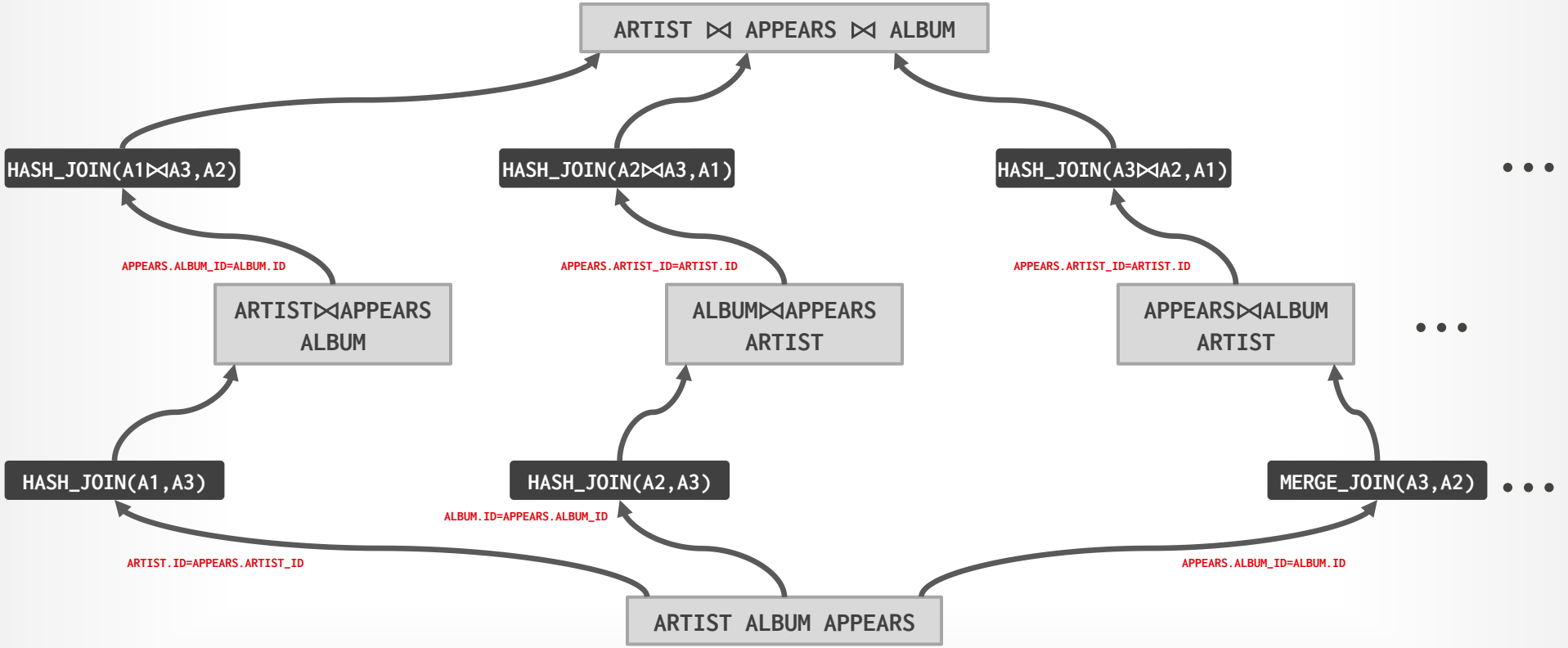
SYSTEM R OPTIMIZER



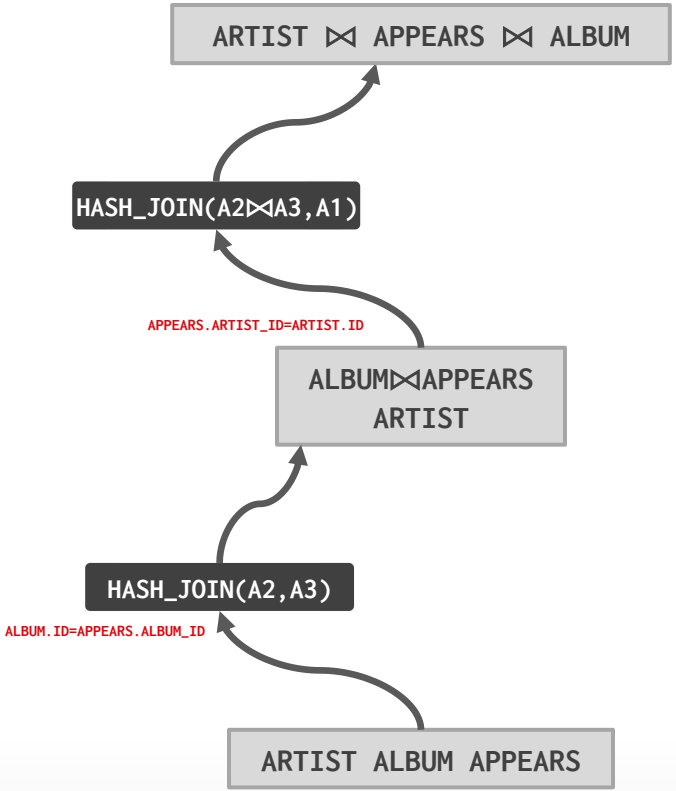
Logical Op

Physical Op

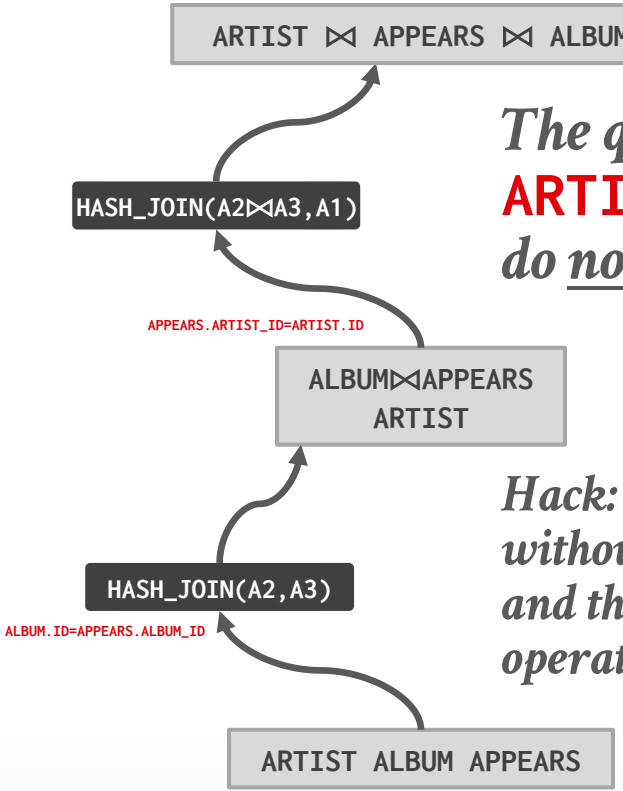
SYSTEM R OPTIMIZER



SYSTEM R OPTIMIZER



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.

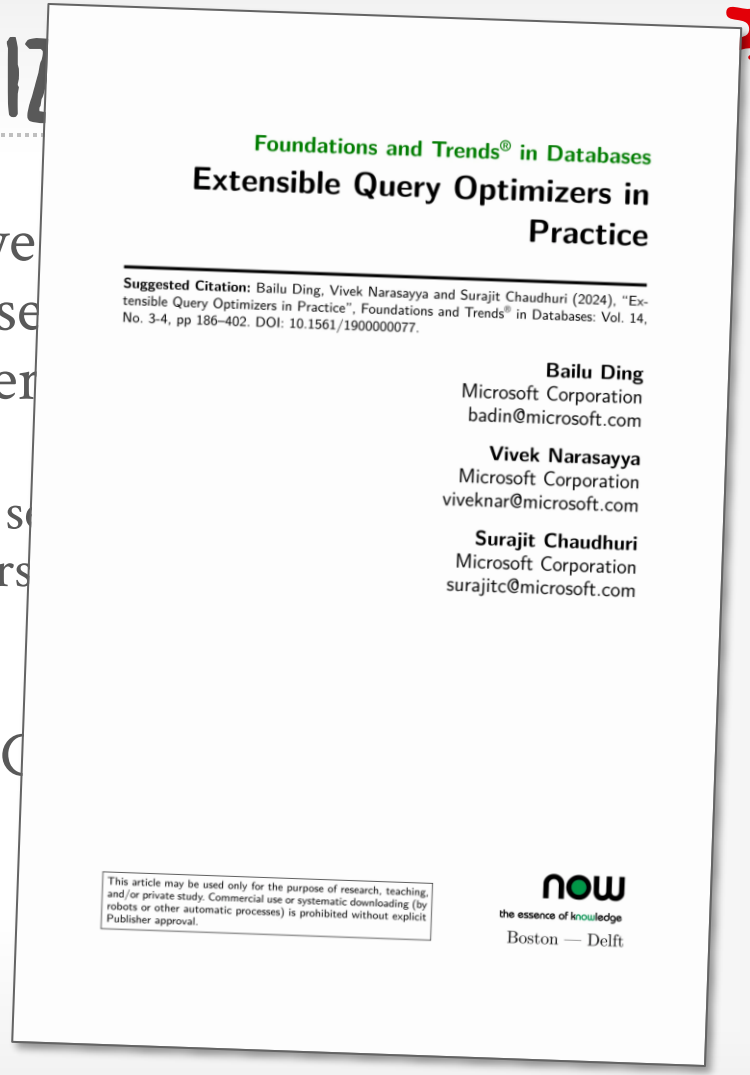
Examples: MSSQL, Greenplum, CockroachDB

TOP-DOWN OPTIMIZ

Start with a logical plan of what we
be. Perform a branch-and-bound se
plan tree by converting logical oper
operators.

→ Keep track of global best plan during s
→ Treat physical properties of data as firs
planning.

Examples: MSSQL, Greenplum, C



Foundations and Trends® in Databases

Extensible Query Optimizers in Practice

Suggested Citation: Bailu Ding, Vivek Narasayya and Surajit Chaudhuri (2024), "Extensible Query Optimizers in Practice", Foundations and Trends® in Databases: Vol. 14, No. 3-4, pp 186–402. DOI: 10.1561/1900000077.

Bailu Ding
Microsoft Corporation
badin@microsoft.com

Vivek Narasayya
Microsoft Corporation
viveknar@microsoft.com

Surajit Chaudhuri
Microsoft Corporation
surajitc@microsoft.com

now
the essence of knowledge
Boston — Delft

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)`

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

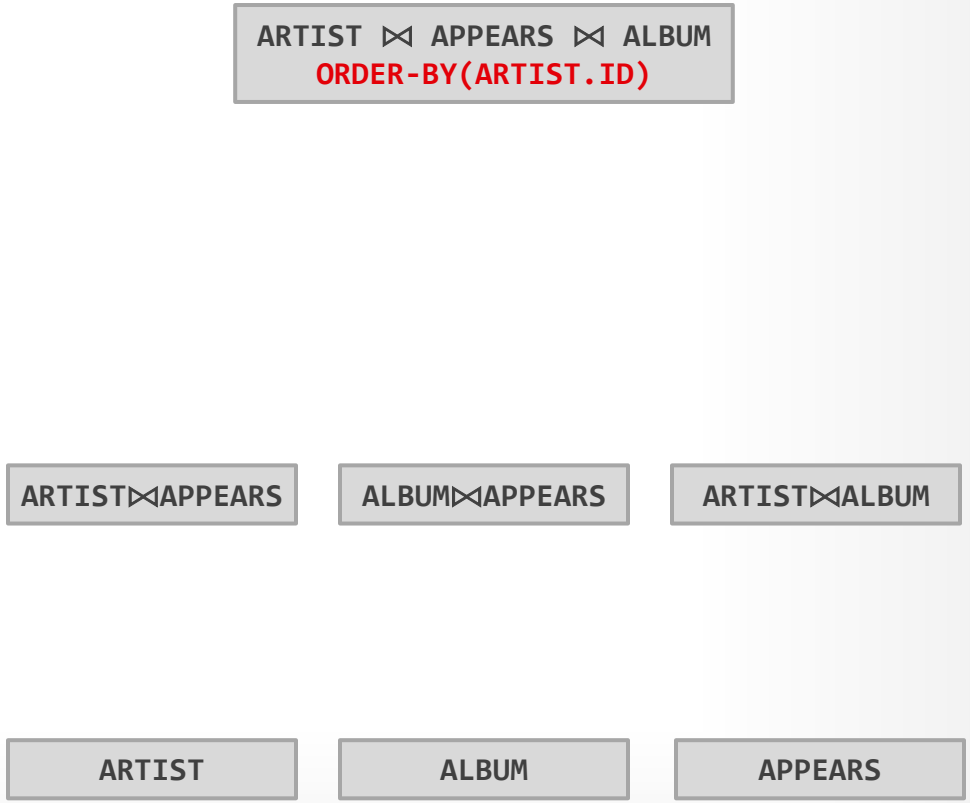
TOP-DOWN OPTIMIZATION

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)`



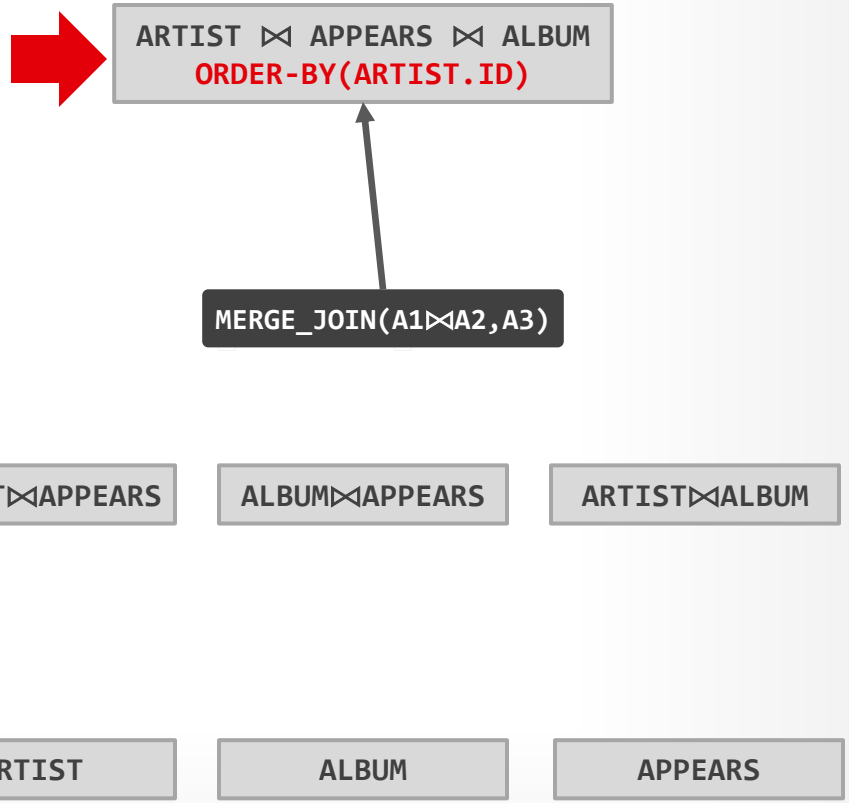
TOP-DOWN OPTIMIZATION

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)`



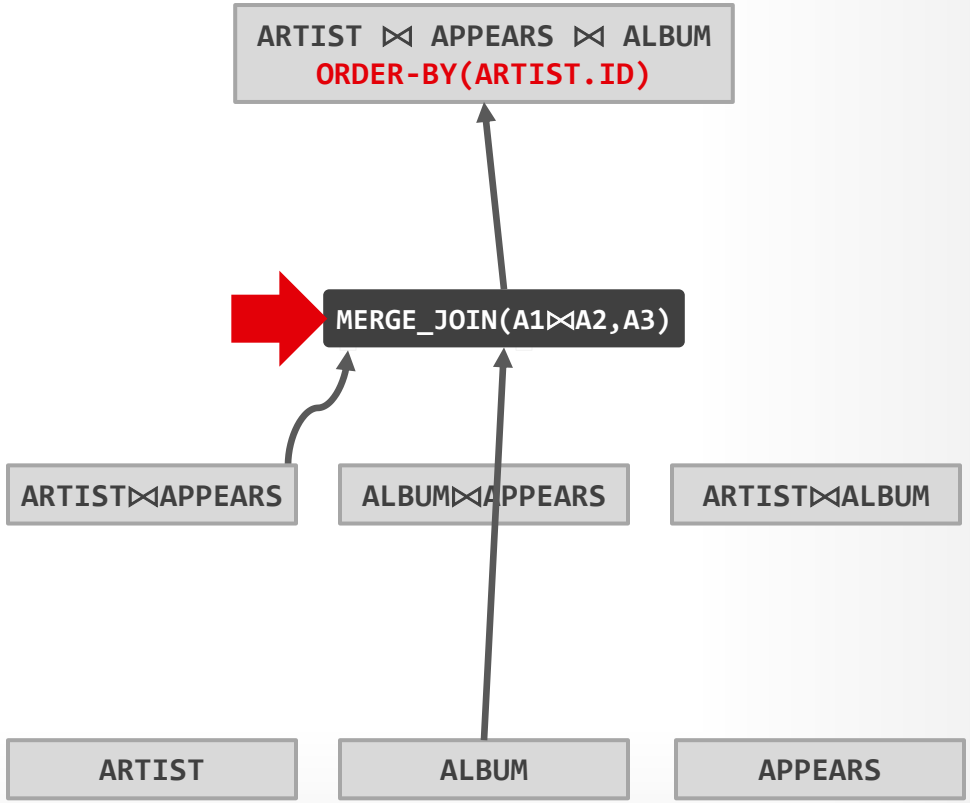
TOP-DOWN OPTIMIZATION

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)`



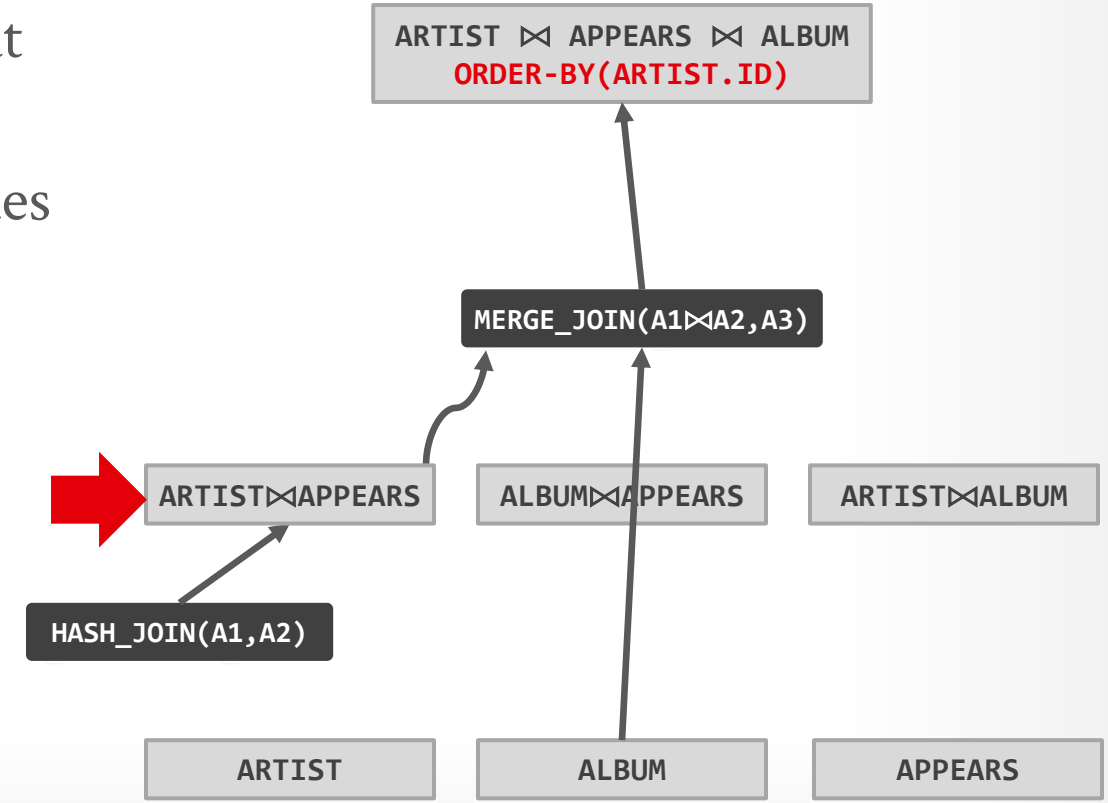
TOP-DOWN OPTIMIZATION

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)`



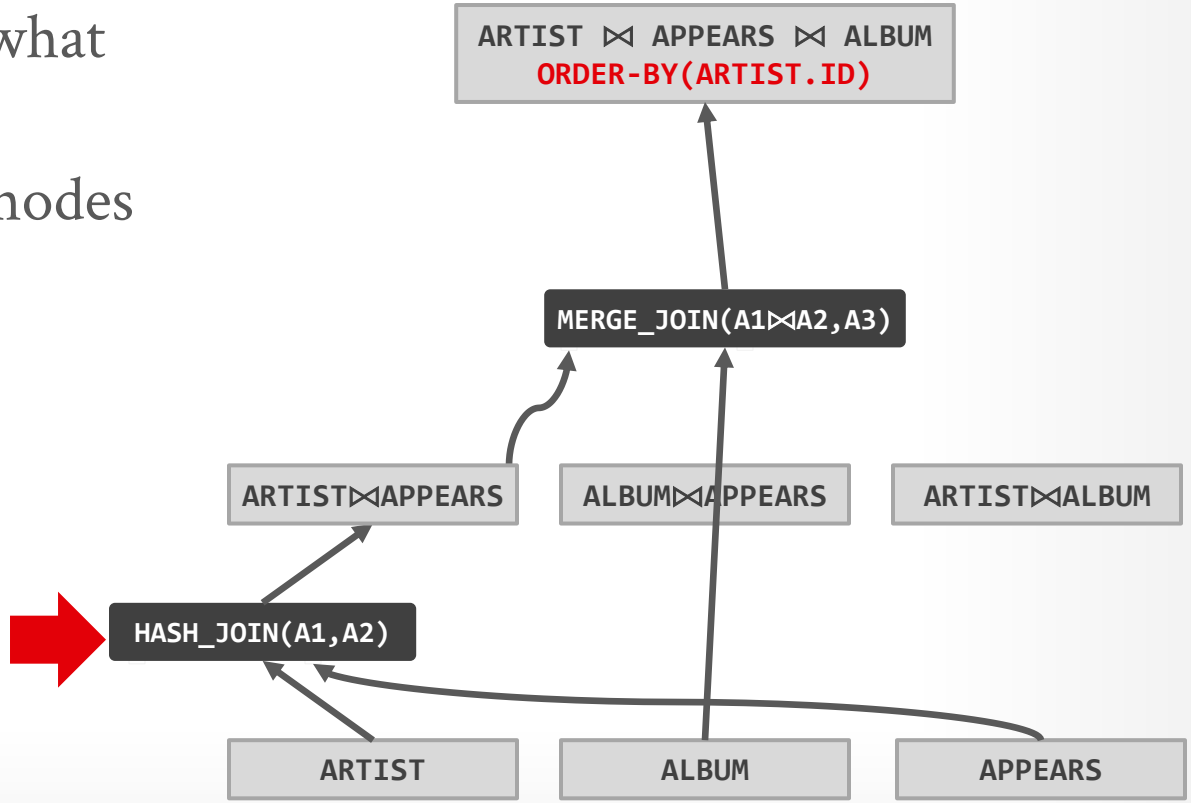
TOP-DOWN OPTIMIZATION

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)`



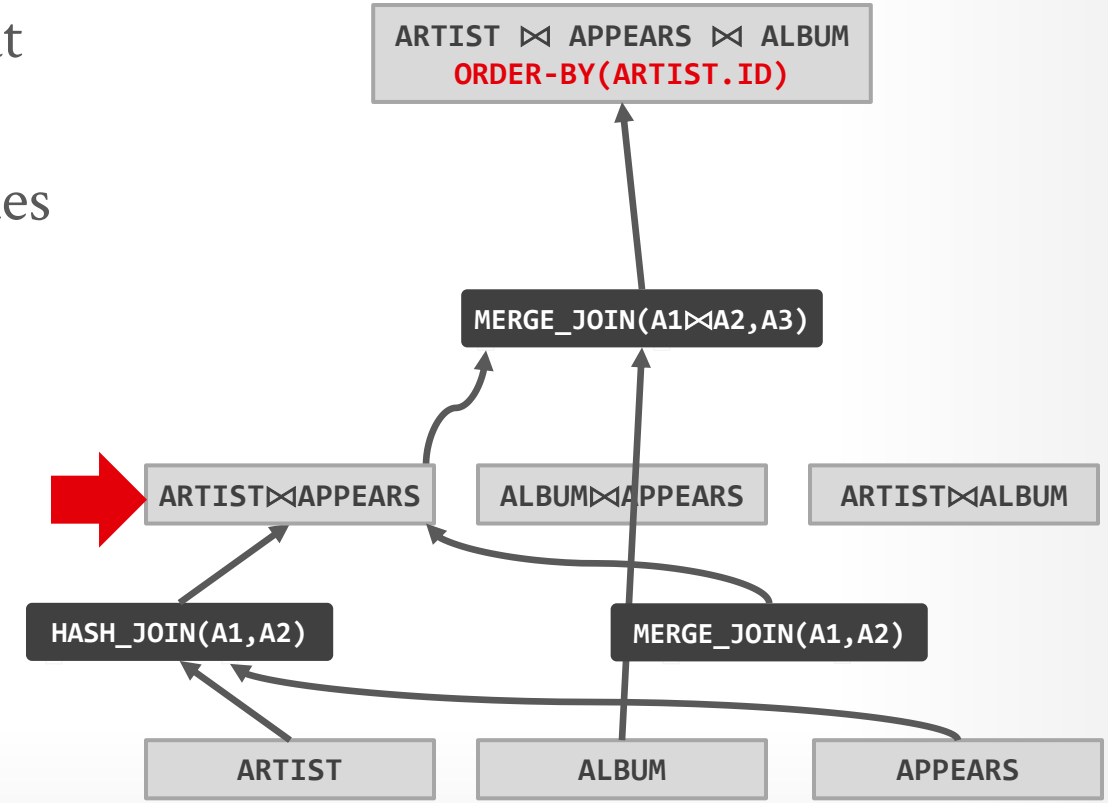
TOP-DOWN OPTIMIZATION

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)`



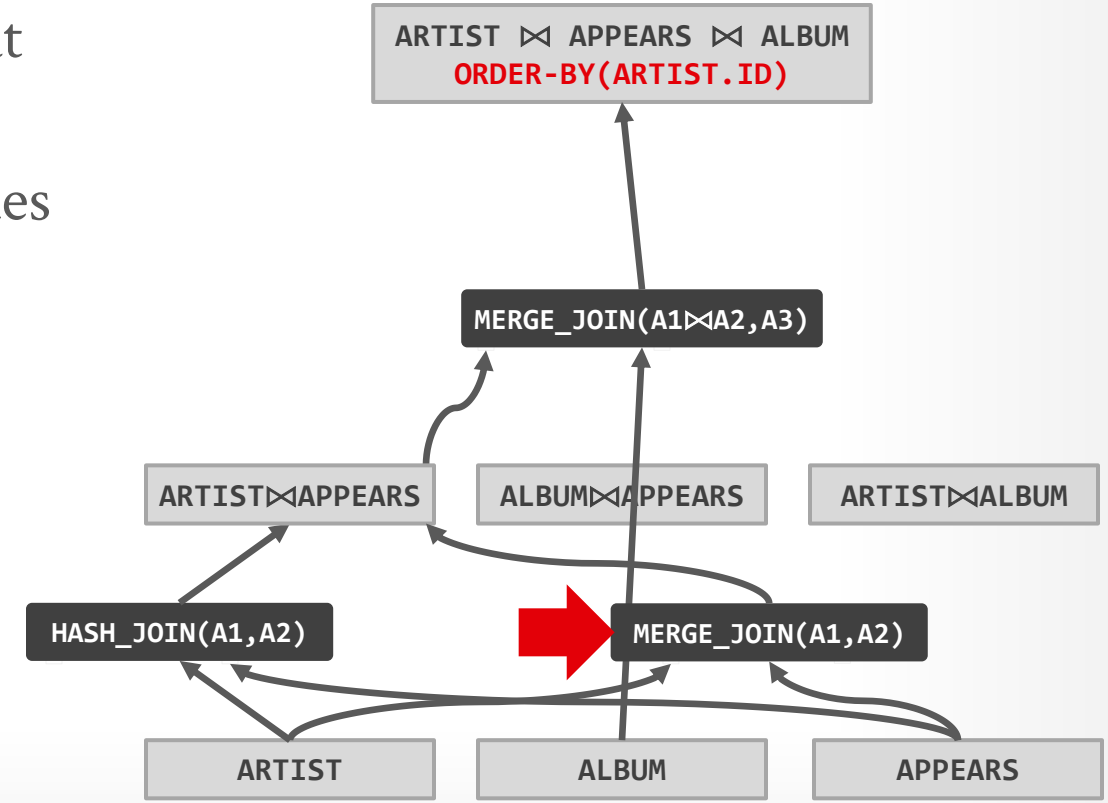
TOP-DOWN OPTIMIZATION

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)`



TOP-DOWN OPTIMIZATION

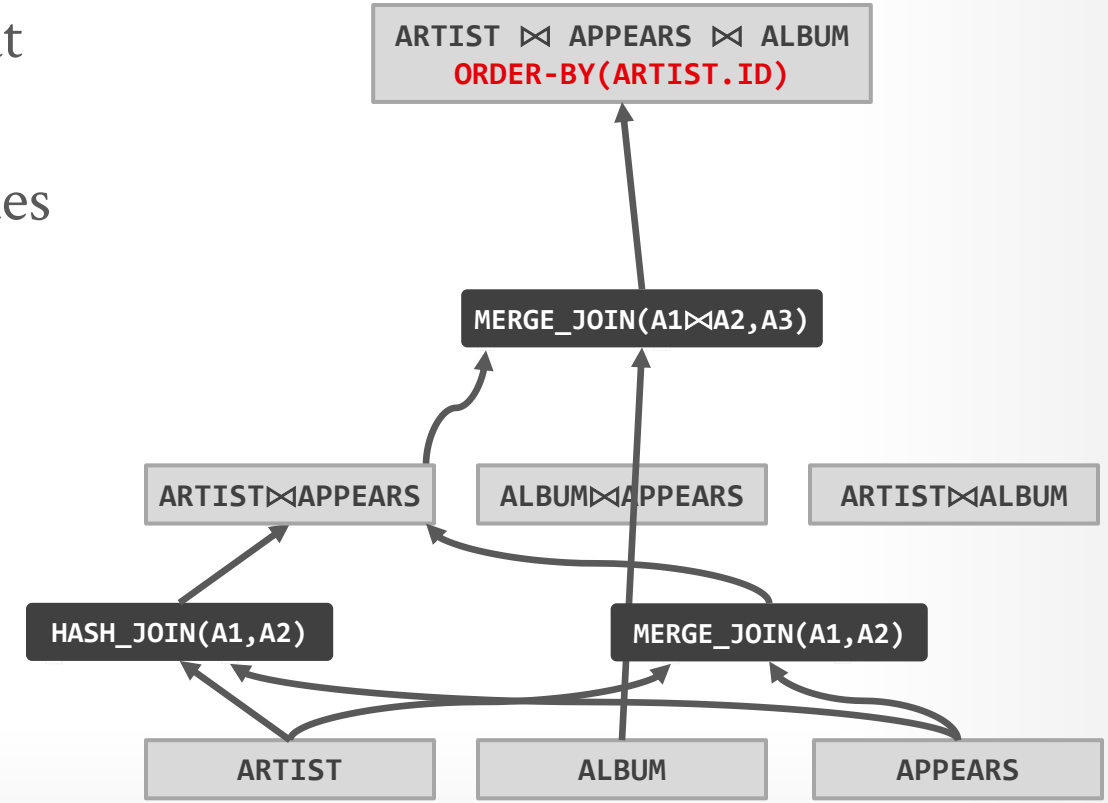
Start with a logical plan of what we want the query to be.

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.



TOP-DOWN OPTIMIZATION

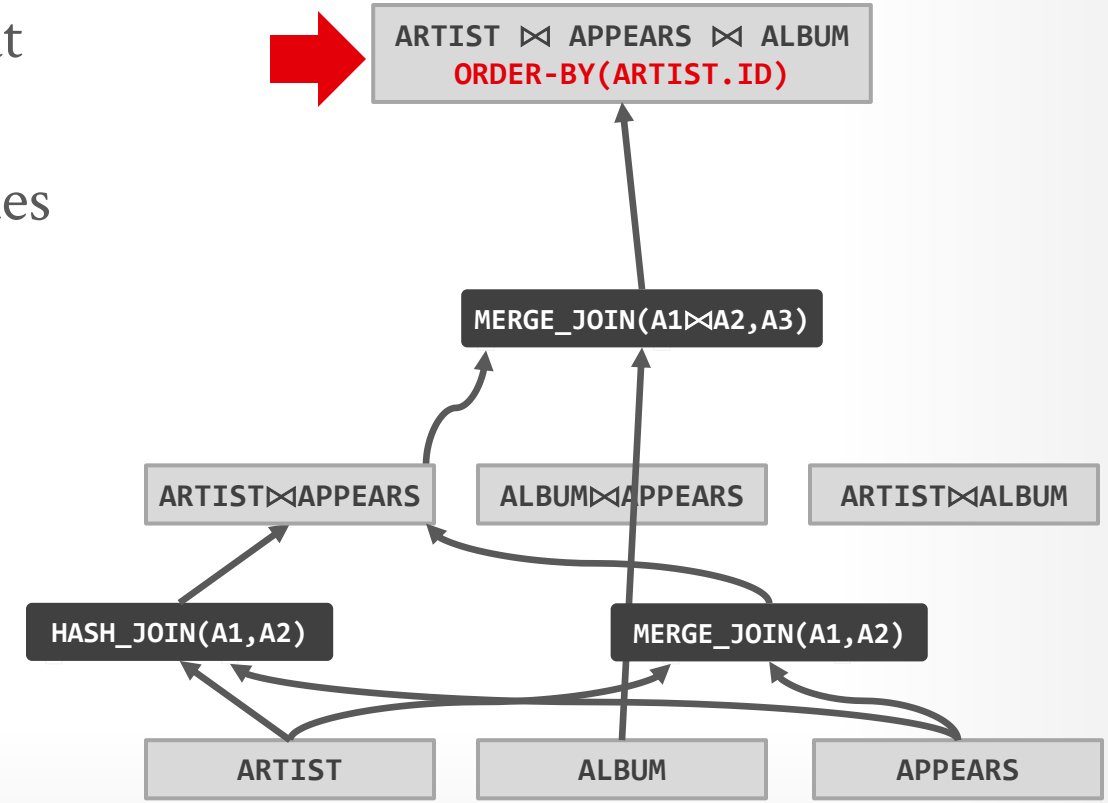
Start with a logical plan of what we want the query to be.

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.



TOP-DOWN OPTIMIZATION

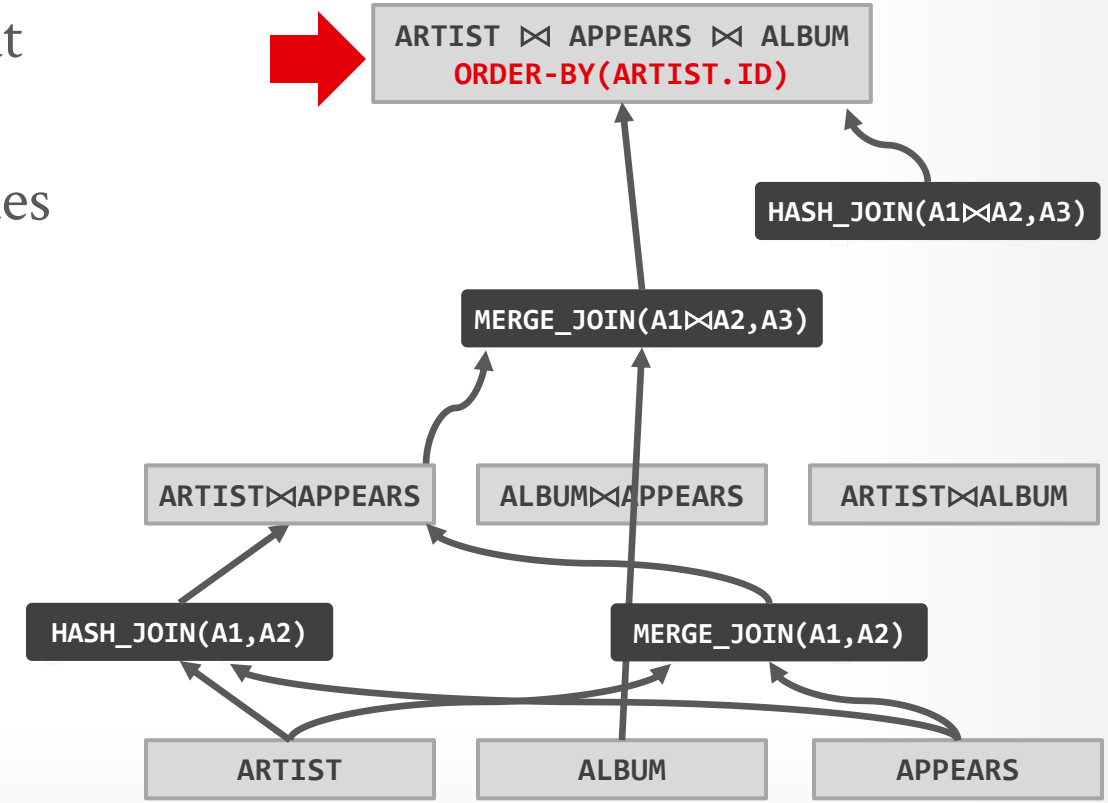
Start with a logical plan of what we want the query to be.

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.



TOP-DOWN OPTIMIZATION

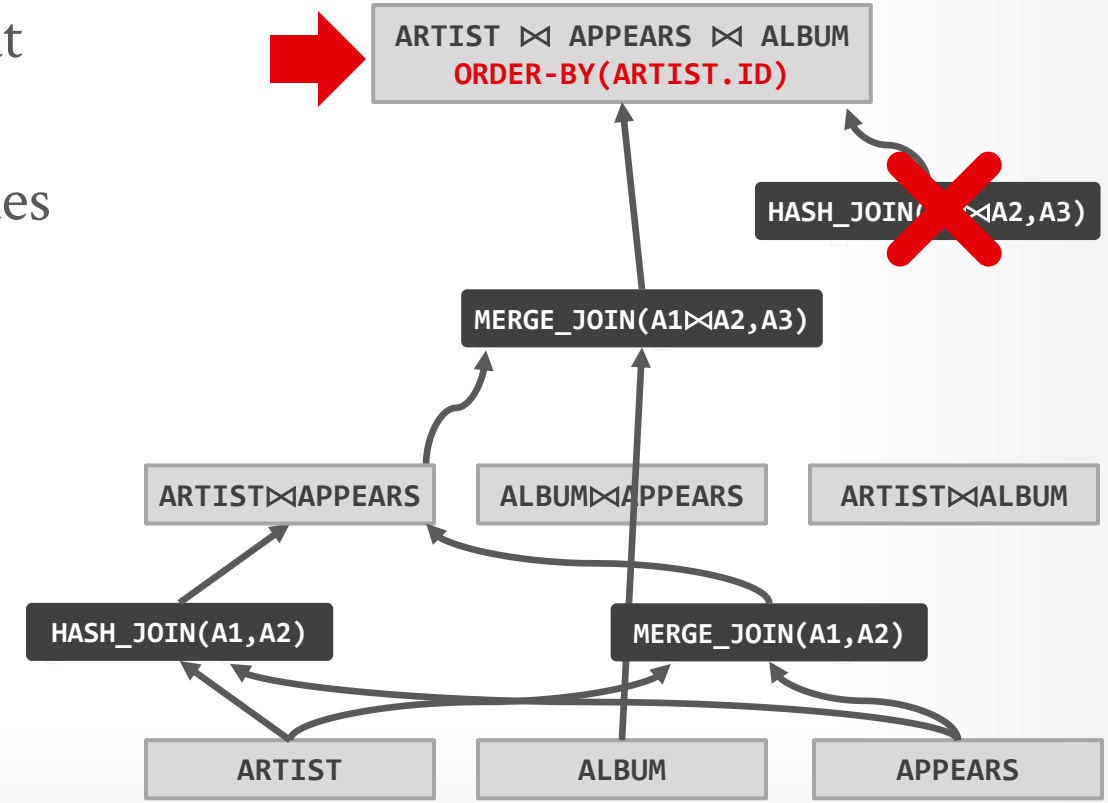
Start with a logical plan of what we want the query to be.

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.



TOP-DOWN OPTIMIZATION

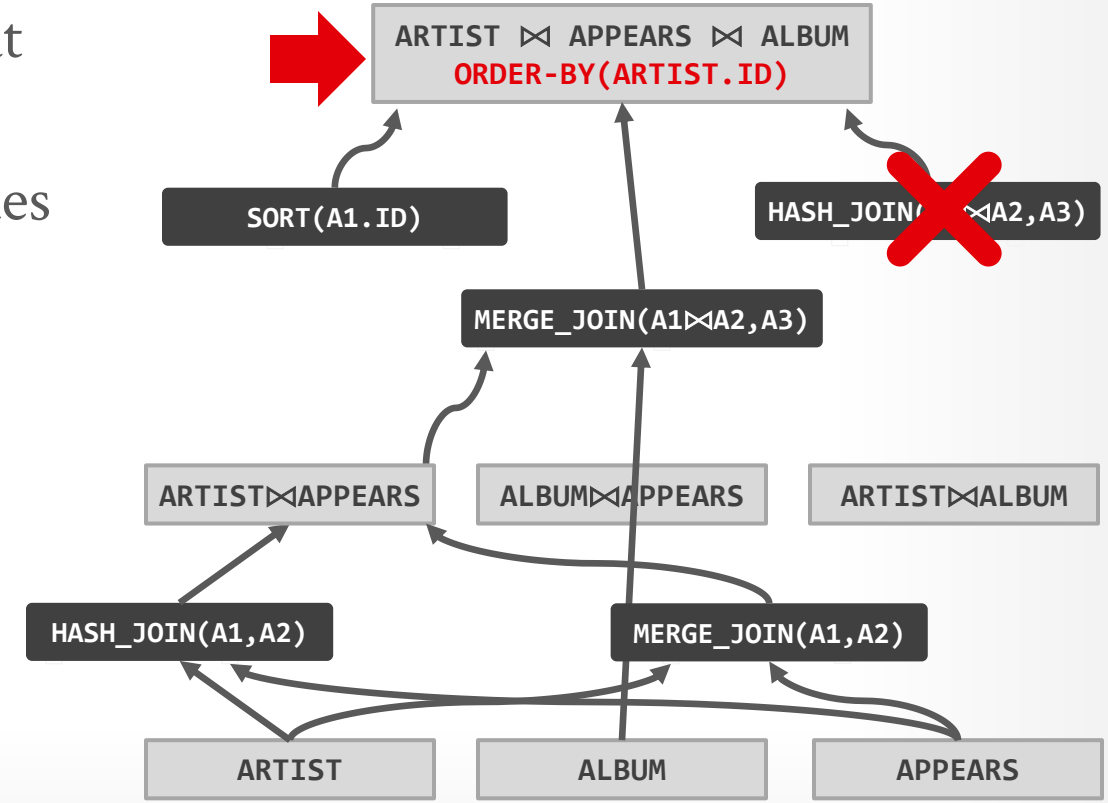
Start with a logical plan of what we want the query to be.

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.



TOP-DOWN OPTIMIZATION

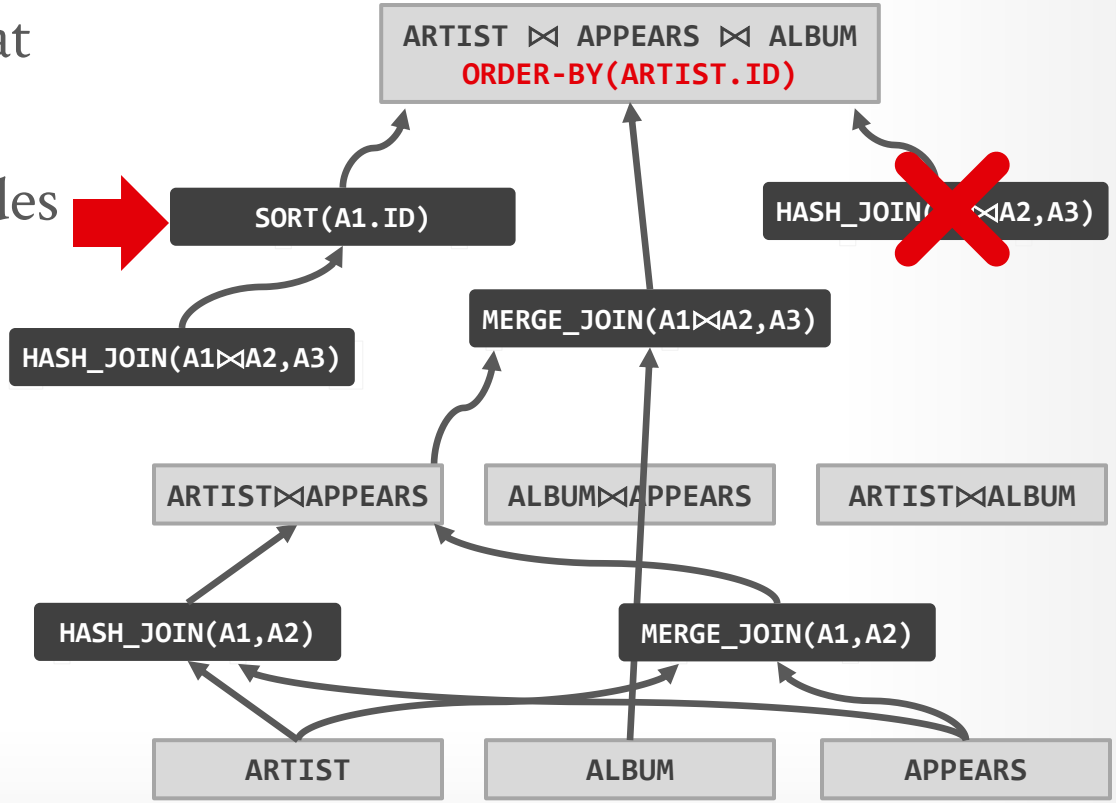
Start with a logical plan of what we want the query to be.

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

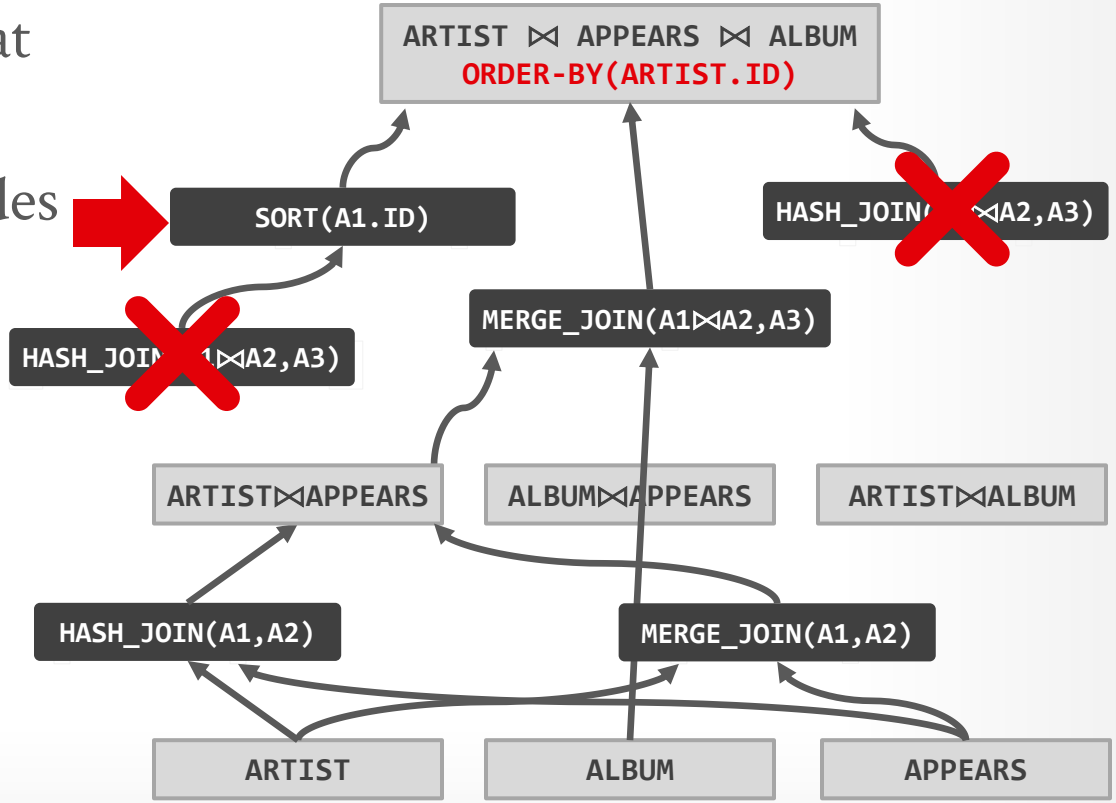
Start with a logical plan of what we want the query to be.

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.

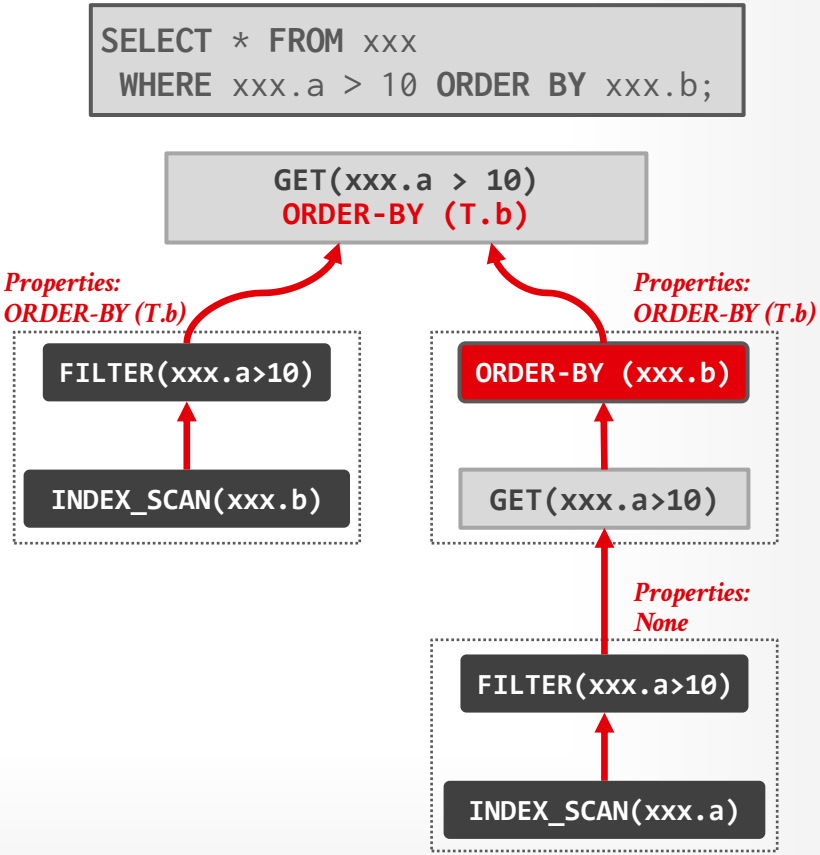


TOP-DOWN OPTIMIZATION: ENFORCERS

Enforcers are physical operators that ensure the properties of the output of a sub-plan / expression.

Volcano's rule engine has additional logical to avoid considering operators below it in the plan that satisfy its property requirements.

→ Example: **INDEX_SCAN(xxx.b)**



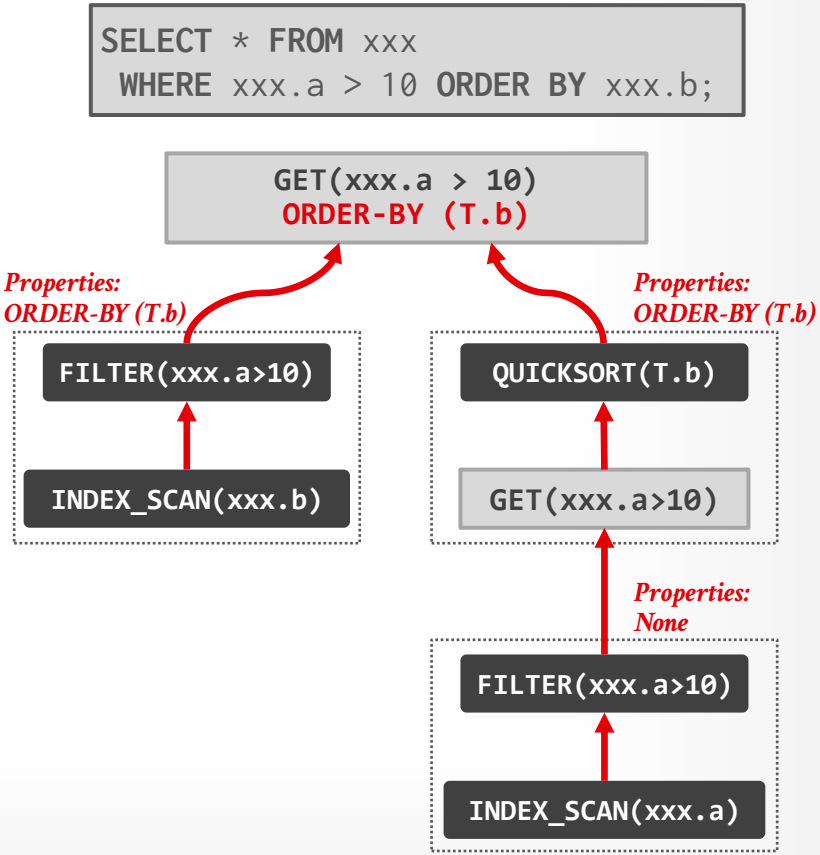
Source: EQOP Book

TOP-DOWN OPTIMIZATION: ENFORCERS

Enforcers are physical operators that ensure the properties of the output of a sub-plan / expression.

Volcano's rule engine has additional logical to avoid considering operators below it in the plan that satisfy its property requirements.

→ Example: **INDEX_SCAN(xxx.b)**



Source: EQOP Book

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 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 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;
```

CONCLUSION

Query optimization is critical for a database system.
→ SQL → Logical Plan → Physical Plan

Transformations change logical operators into either (1) new logical operators or (2) physical operators.

Two search strategies:
→ **Bottom-Up:** Start with nothing and then iteratively assemble query plan.
→ **Top-Down:** Start with the outcome and then transform it to equivalent alternatives to achieve that outcome.

NEXT CLASS

Query Optimizers Part 2: Cost Models
→ aka "Everybody has a plan until they get punched in the mouth"

Search for "$DBMS bad query plan"