Intro to Database Systems (15-445/645)

14 Query Planning & Optimization

Carnegie Mellon University

SPRING 2023

Charlie Garrod

ADMINISTRIVIA

Project 2 still ongoing
→ First checkpoint was Friday, March 3rd (15% of P2 grade)
→ Overall due Wednesday, March 22nd (85% of P2 grade)

Project 3 released next week

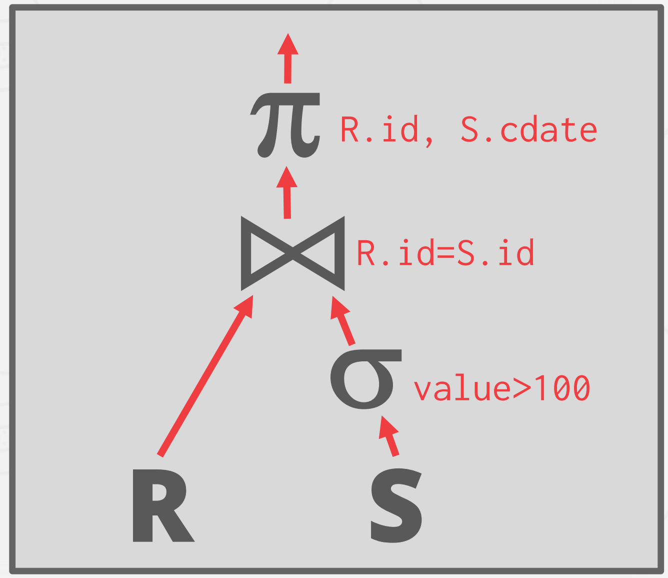
LAST TIME: PARALLEL QUERY EXECUTION

Process models

Executing queries in parallel

I/O parallelism

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



LAST TIME: ON

Process m
Executing
I/O paral


```
cdate
S
S.id
> 100
```

```
S.cdate
=S.id
alue>100
```

QUERY OPTIMIZATION

For a given query, find a **correct** execution plan that has the lowest "cost".

This is hard to implement well (is NP-Complete!).

No optimizer truly produces the "optimal" plan
→ Use estimation techniques to guess real plan cost.
→ Use heuristics to limit the search space.

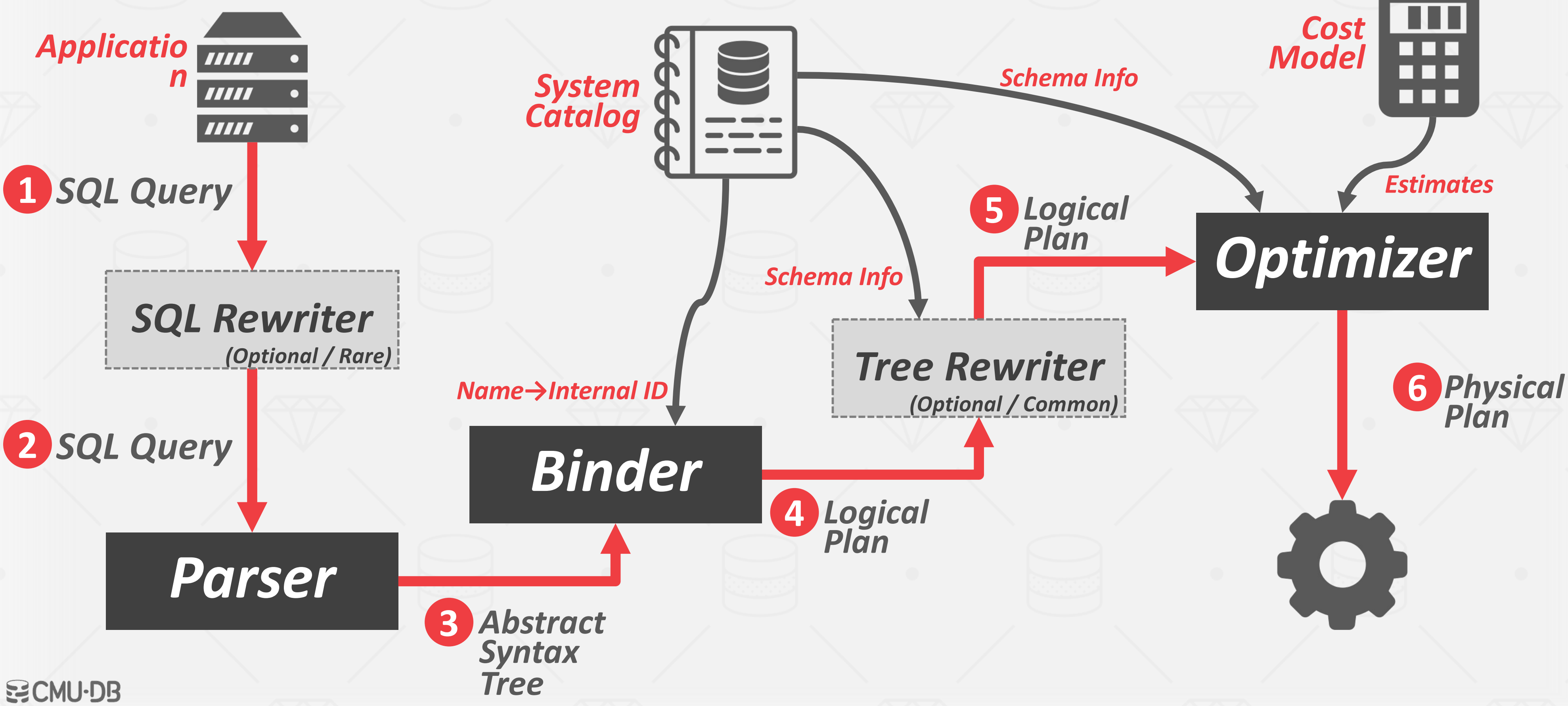
LOGICAL VS. PHYSICAL PLANS

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.

ARCHITECTURE OVERVIEW



QUERY OPTIMIZATION

Heuristics / Rules

→ Rewrite the query to remove stupid / inefficient things.
→ 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.

TODAY'S AGENDA

Heuristic/Ruled-based Optimization
Query Cost Models
Cost-based Optimization

LOGICAL PLAN OPTIMIZATION

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

The goal is to increase the likelihood of enumerating the optimal plan in the search.

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

LOGICAL QUERY OPTIMIZATION

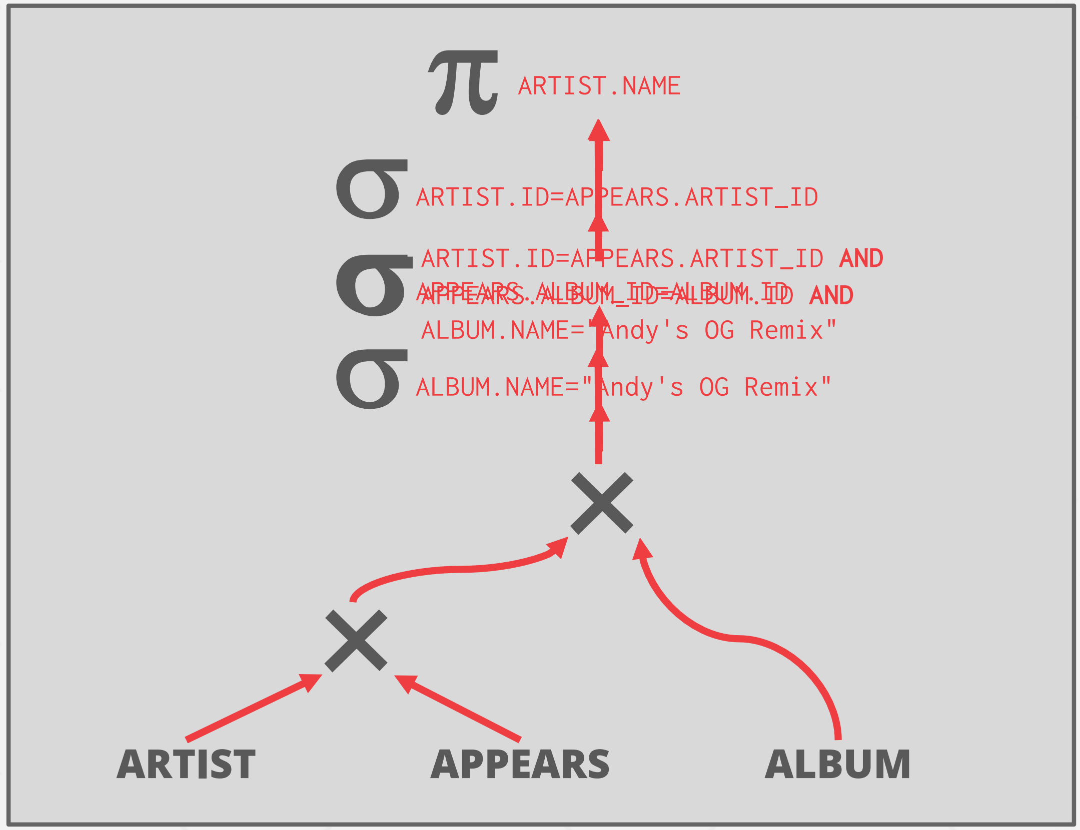
Split Conjunctive Predicates
Predicate Pushdown
Replace Cartesian Products with Joins
Projection Pushdown

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="Andy's OG Remix"
```

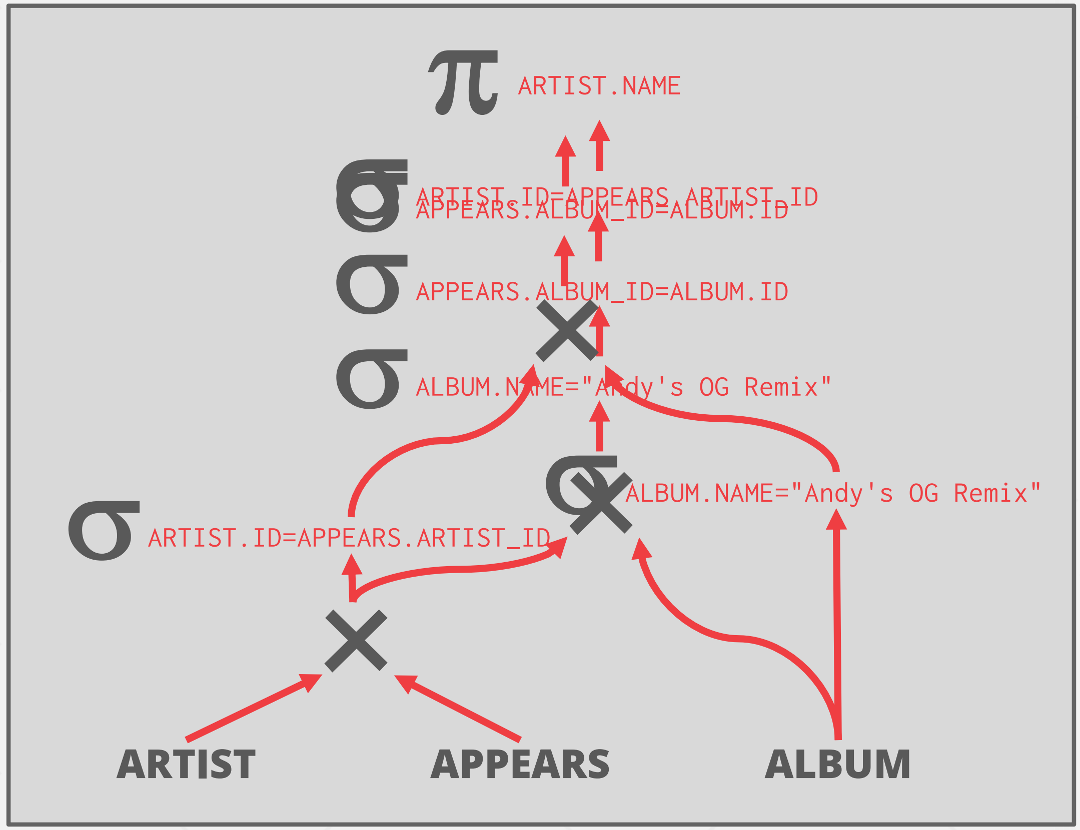
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



PREDICATE PUSHDOWN

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```

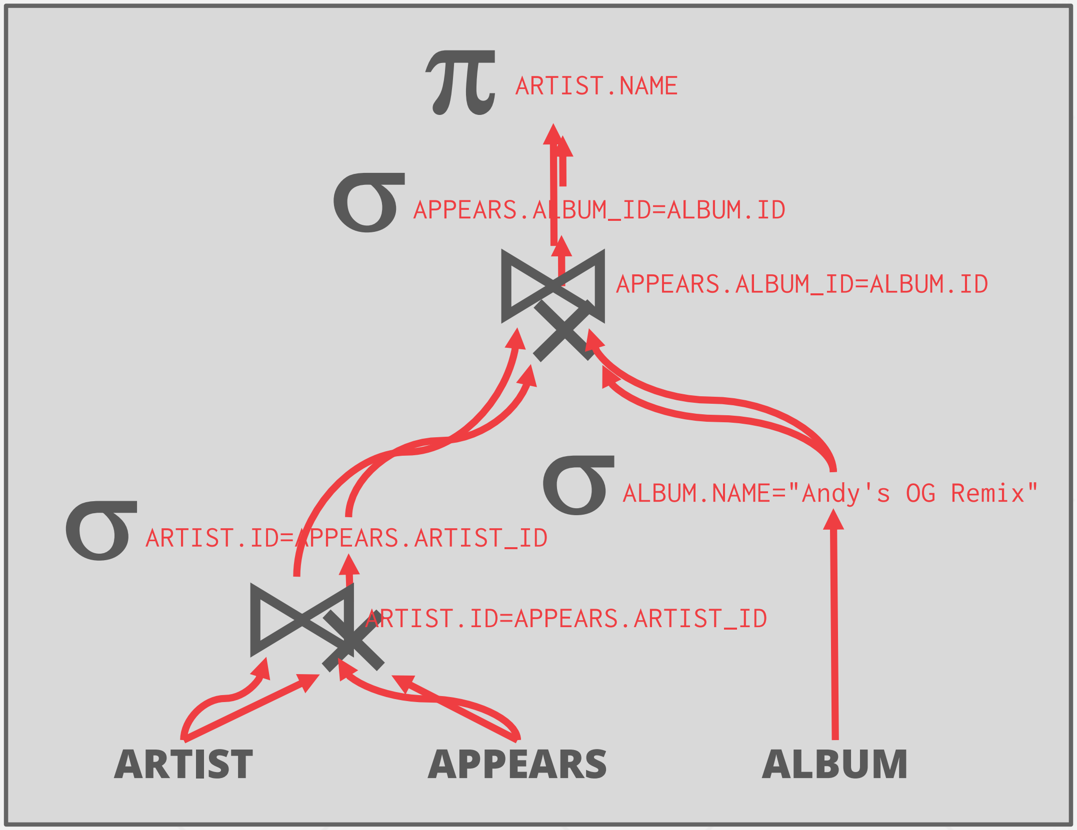
Move the predicate to the lowest applicable point in the plan.



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="Andy's OG Remix"
```

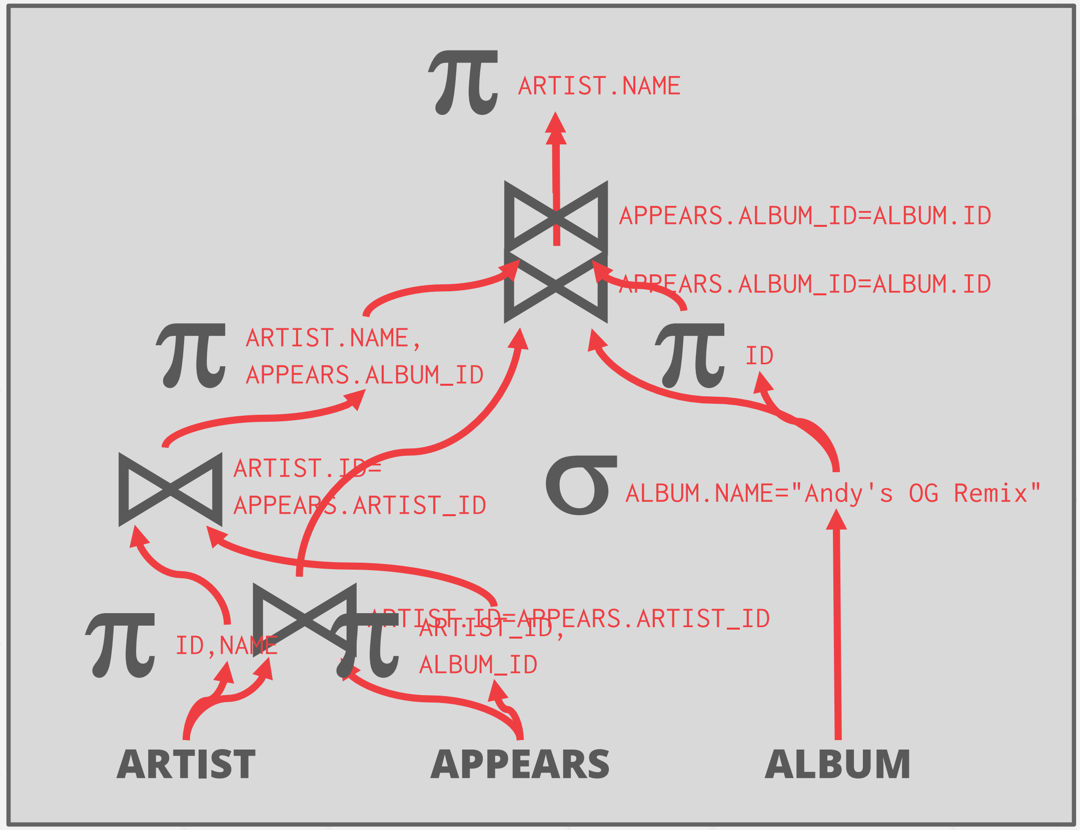
Replace all Cartesian Products with inner joins using the join predicates.



PROJECTION PUSHDOWN

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```

Eliminate redundant attributes before pipeline breakers to reduce materialization cost.



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 result to 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 a temporary table that are discarded after the query finishes.

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
```

Outer Block ***Nested 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 false;
```

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

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

Merging Predicates

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

QUERY OPTIMIZATION

Heuristics / Rules

→ Rewrite the query to remove stupid / inefficient things.
→ 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 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.

Choice #3: Algorithmic Costs

→ Complexity of the operator algorithm implementation.

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)`

IBM DB2 COST MODEL

Database characteristics in system catalogs

Hardware environment (microbenchmarks)

Storage device characteristics (microbenchmarks)

Communications bandwidth (distributed only)

Memory resources (buffer pools, sort heaps)

Concurrency Environment

→ Average number of users
→ Isolation level / blocking
→ Number of available locks

Source: Guy Lohman

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

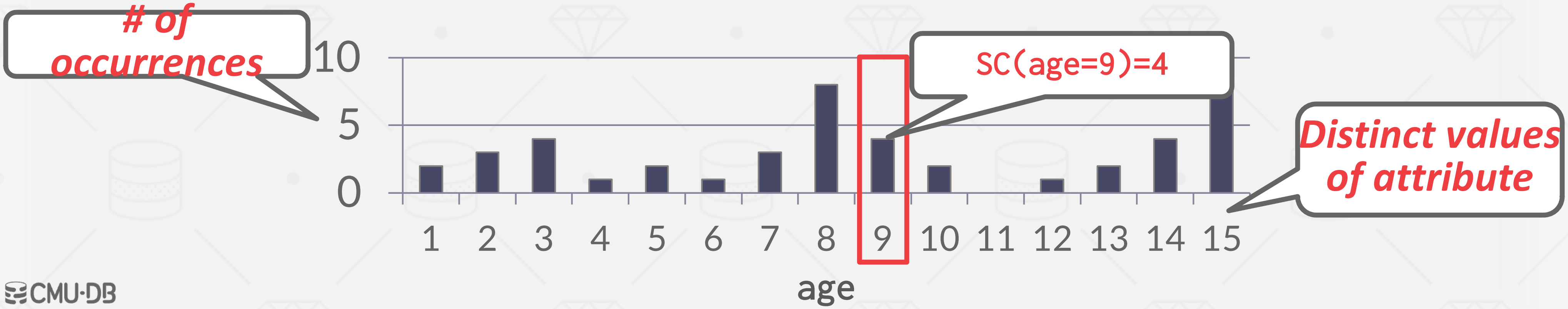
The **selectivity** (`sel`) of a predicate `P` is the fraction of tuples that qualify.

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

Equality Predicate: `A=constant`

→ `sel(A=constant) = #occurences/|R|`

→ 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

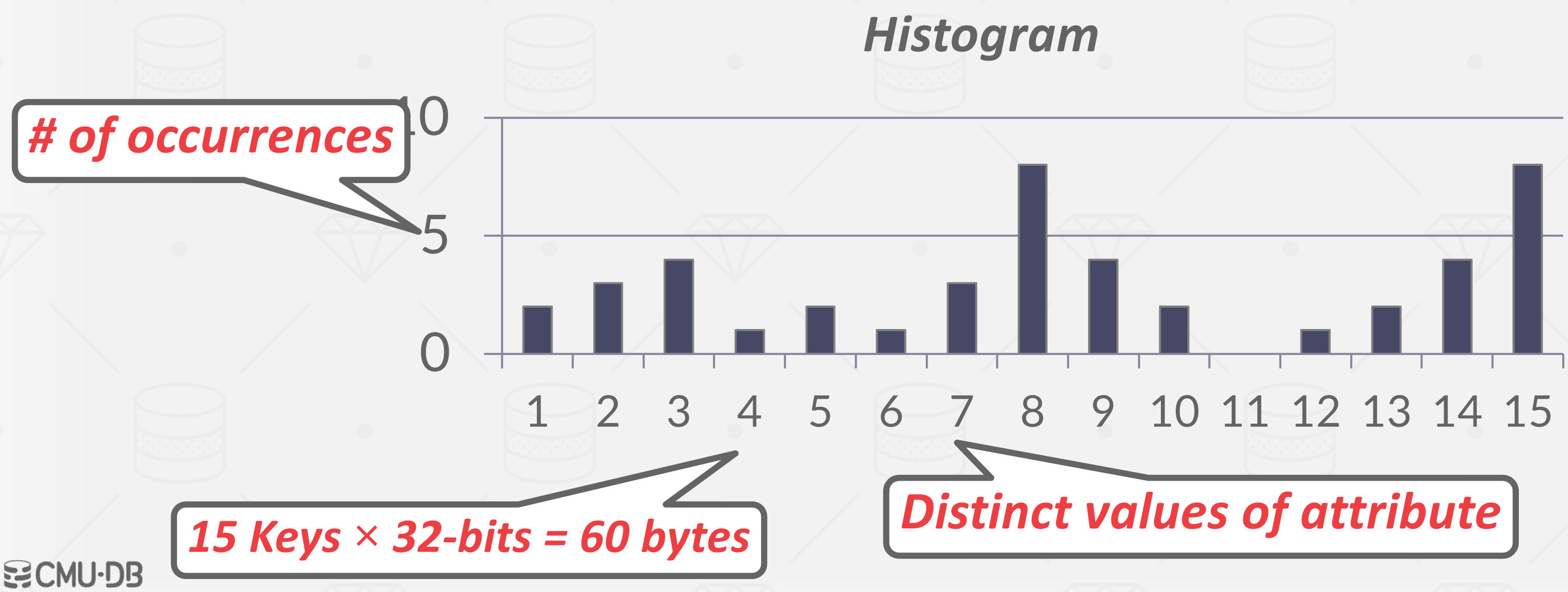
→ Probabilistic data structure that gives an approximate count for a given value.

Choice #3: Sampling

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

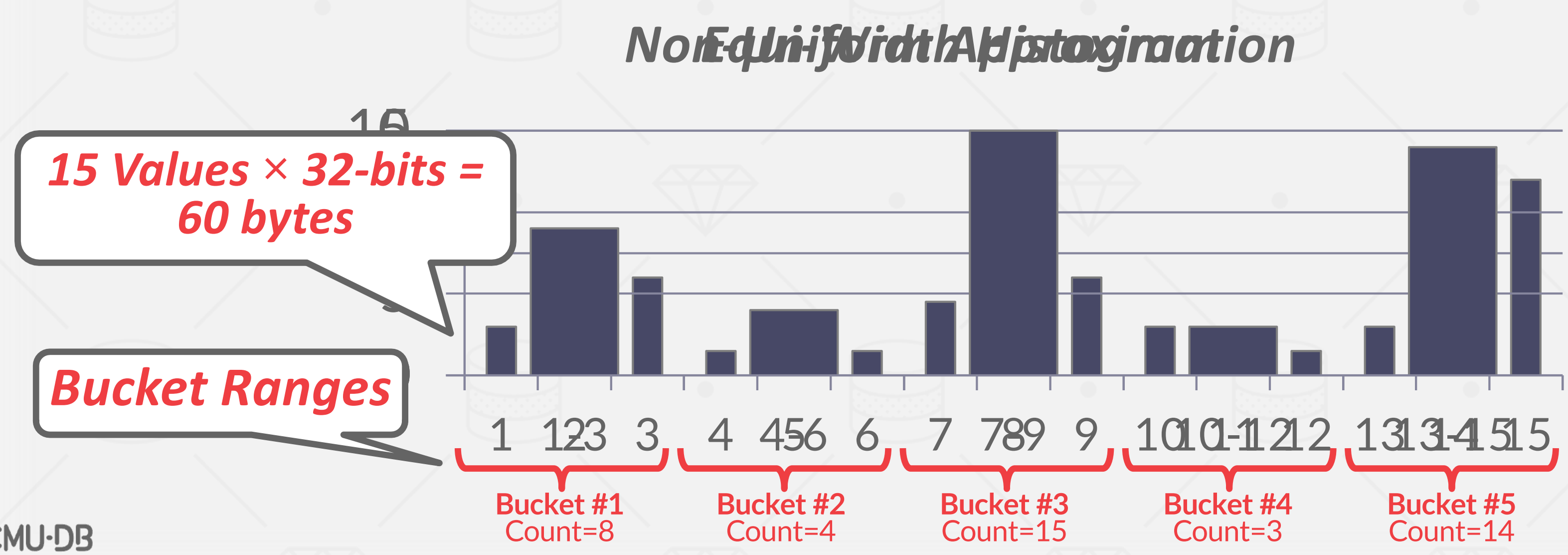
HISTOGRAMS

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



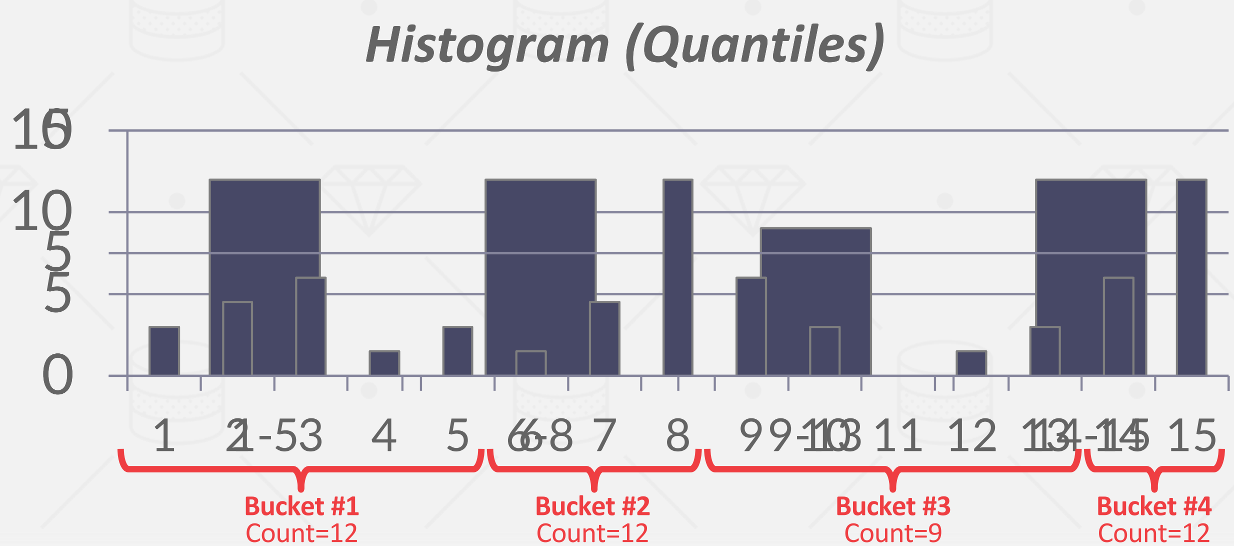
EQUI-WIDTH HISTOGRAM

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.



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.

Common examples:

→ Count-Min Sketch (2003): 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`

OBSERVATION

Now that we can (roughly) estimate the selectivity of predicates, and subsequently the cost of query plans, what can we do with them?

QUERY OPTIMIZATION

After performing rule-based rewriting, the DBMS will 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.

OLTP queries are especially easy…

OLTP QUERY PLANNING

Query planning for OLTP queries is easy because they are **sargable** (**S**earch **Arg**ument **Able**).
→ It is usually just picking the best index.
→ Joins are almost always on foreign key relationships with a small cardinality.
→ Can be implemented with simple heuristics.

```
CREATE TABLE people (
  id INT PRIMARY KEY,
  val INT NOT NULL,
  ⋮
);
```

```
SELECT * FROM people
 WHERE id = 123;
```

MULTI-RELATION QUERY PLANNING

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.

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

SYSTEM R OPTIMIZER

Break query up 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

SYSTEM R OPTIMIZER

```sql
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

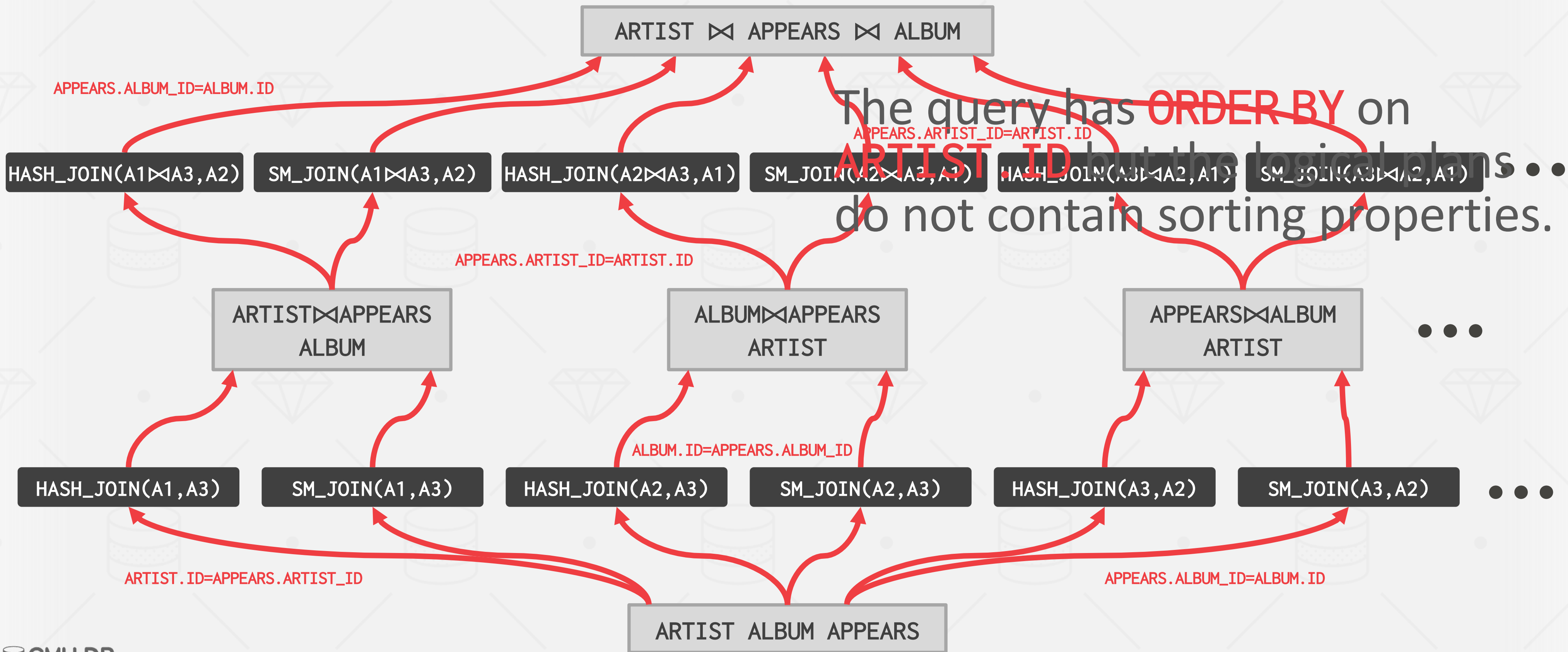
Step #2: Enumerate all possible join orderings for tables

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

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

ARTIST $\bowtie$ APPEARS $\bowtie$ ALBUM
APPEARS $\bowtie$ ALBUM $\bowtie$ ARTIST
ALBUM $\bowtie$ APPEARS $\bowtie$ ARTIST
APPEARS $\bowtie$ ARTIST $\bowtie$ ALBUM
ARTIST $\times$ ALBUM $\bowtie$ APPEARS
ALBUM $\times$ ARTIST $\bowtie$ APPEARS
⋮ ⋮ ⋮

SYSTEM R OPTIMIZER



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

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.

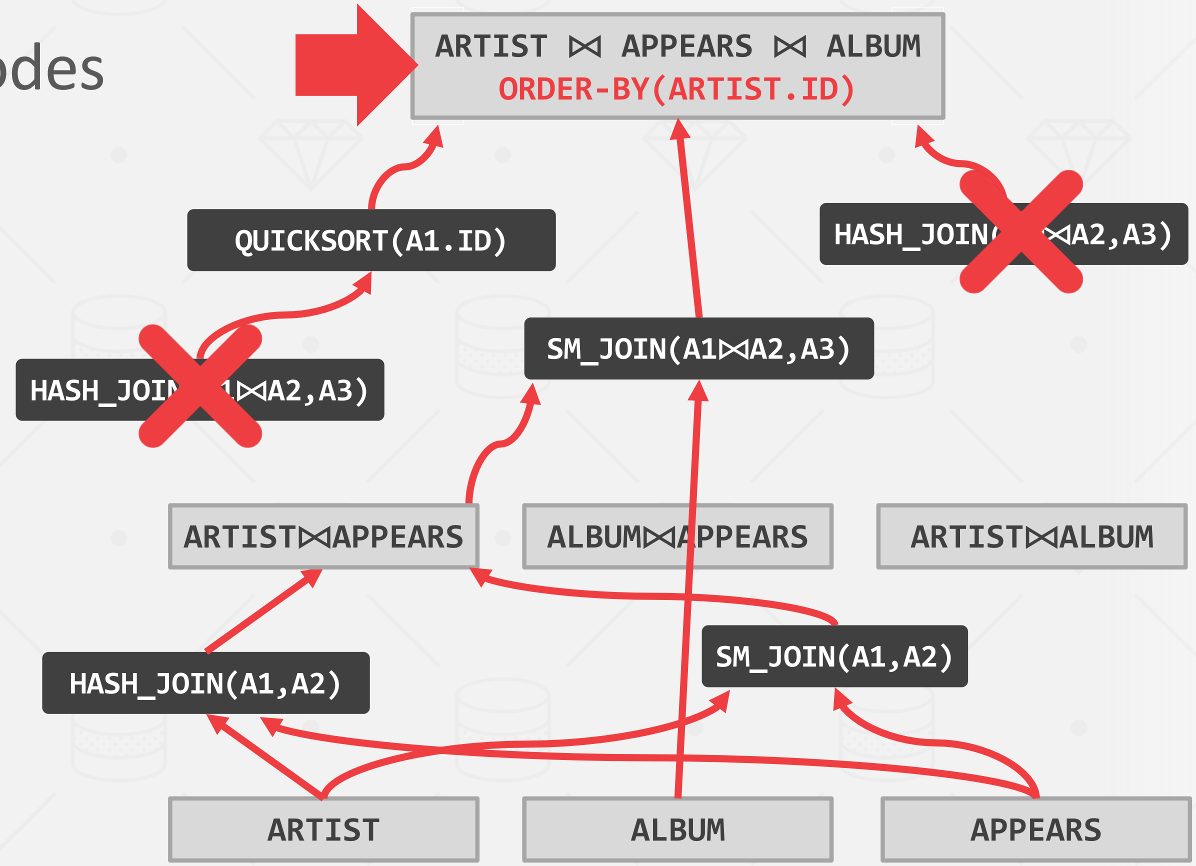
Graefe

Example: MSSQL, Greenplum, CockroachDB

TOP-DOWN OPTIMIZATION

Invoke rules to create new nodes and traverse 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.



CONCLUSION

We use static rules and heuristics to optimize a query plan without needing to understand the contents of the database.

We use cost model to help perform more advanced query optimizations

NEXT CLASS

Transactions!