Carnegie Mellon University

DATABASE SYSTEMS

Query Planning – Pt.2

LECTURE #16 » 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)

LAST CLASS

A DBMS's query optimizer takes logical query plan as input and generates a physical execution plan that has the lowest "cost".

The quality of the plans that an optimizer generates is mostly based on three factors:
→ Transformations / Enumeration
→ Search Algorithm
→ Cost Model

TODAY'S AGENDA

Search Algorithms
Data Statistics
Cost Models

SEARCH ALGORITHM

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

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: 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 $\bowtie$ 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: 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 $\bowtie$ 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
⋮ ⋮ ⋮

Logical Op

Physical Op

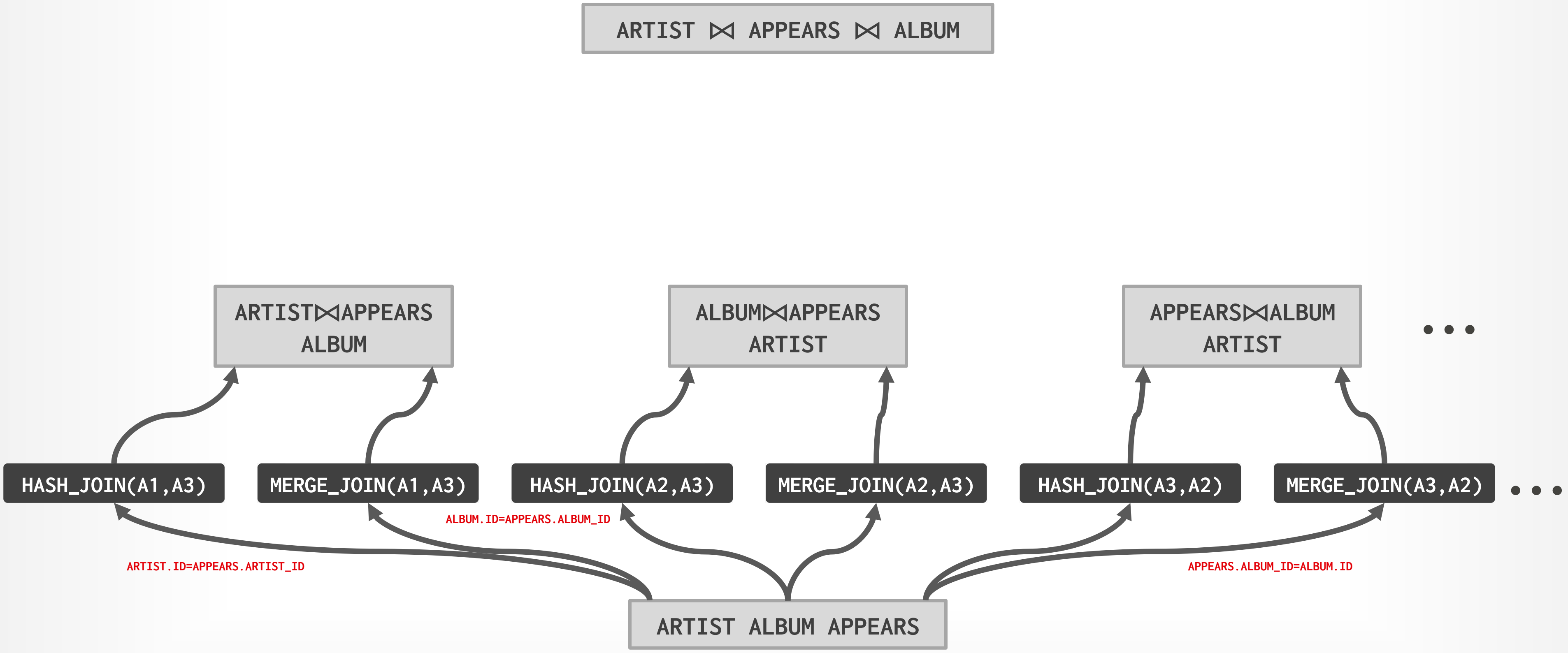
SYSTEM R OPTIMIZER

ARTIST ⋈ APPEARS ⋈ ALBUM

ARTIST ALBUM APPEARS

Logical Op
Physical Op

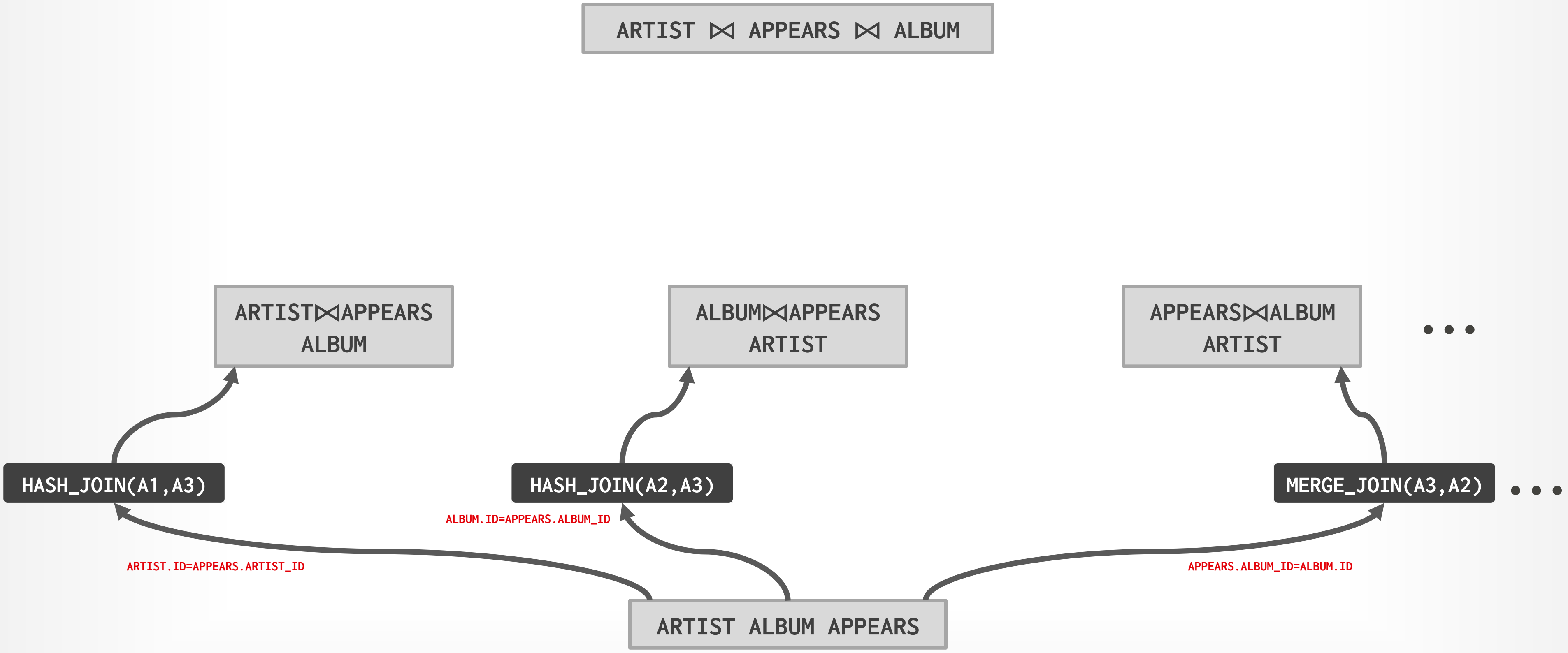
SYSTEM R OPTIMIZER



Logical Op

Physical Op

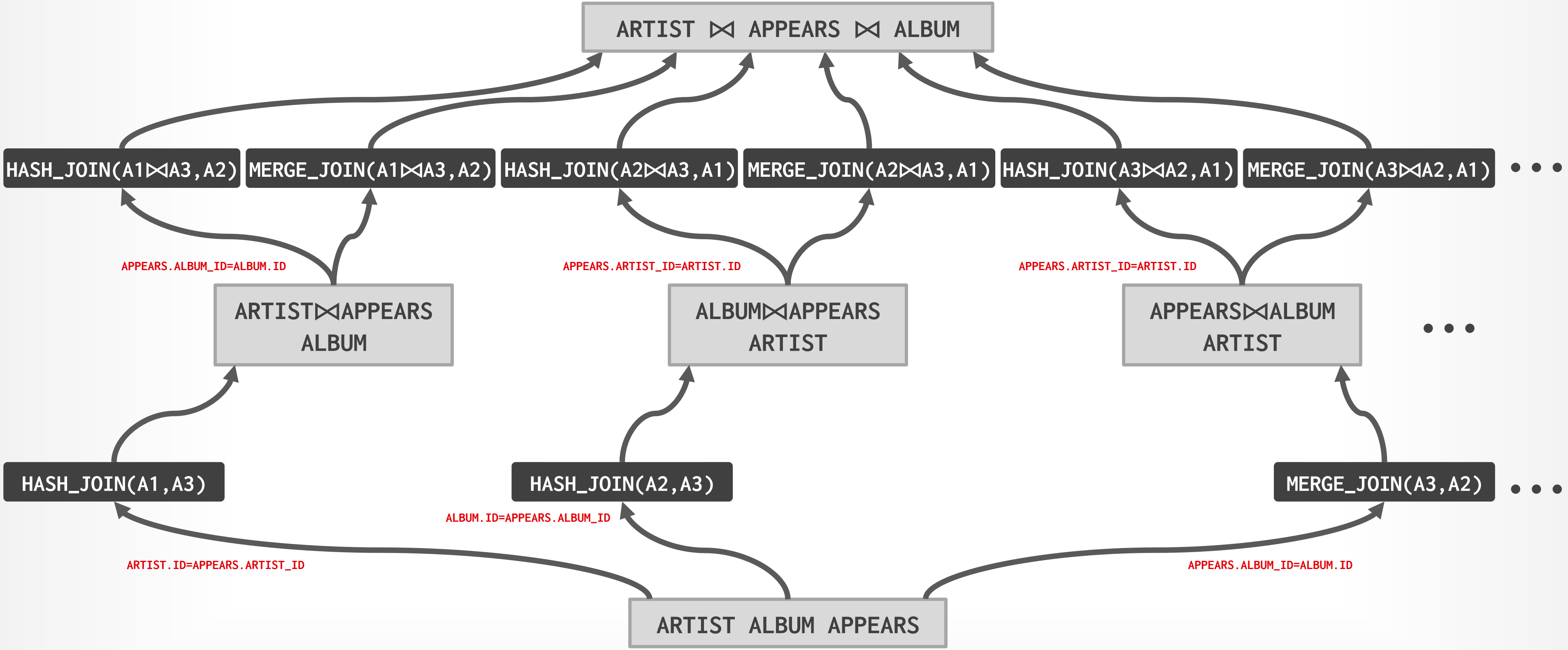
SYSTEM R OPTIMIZER



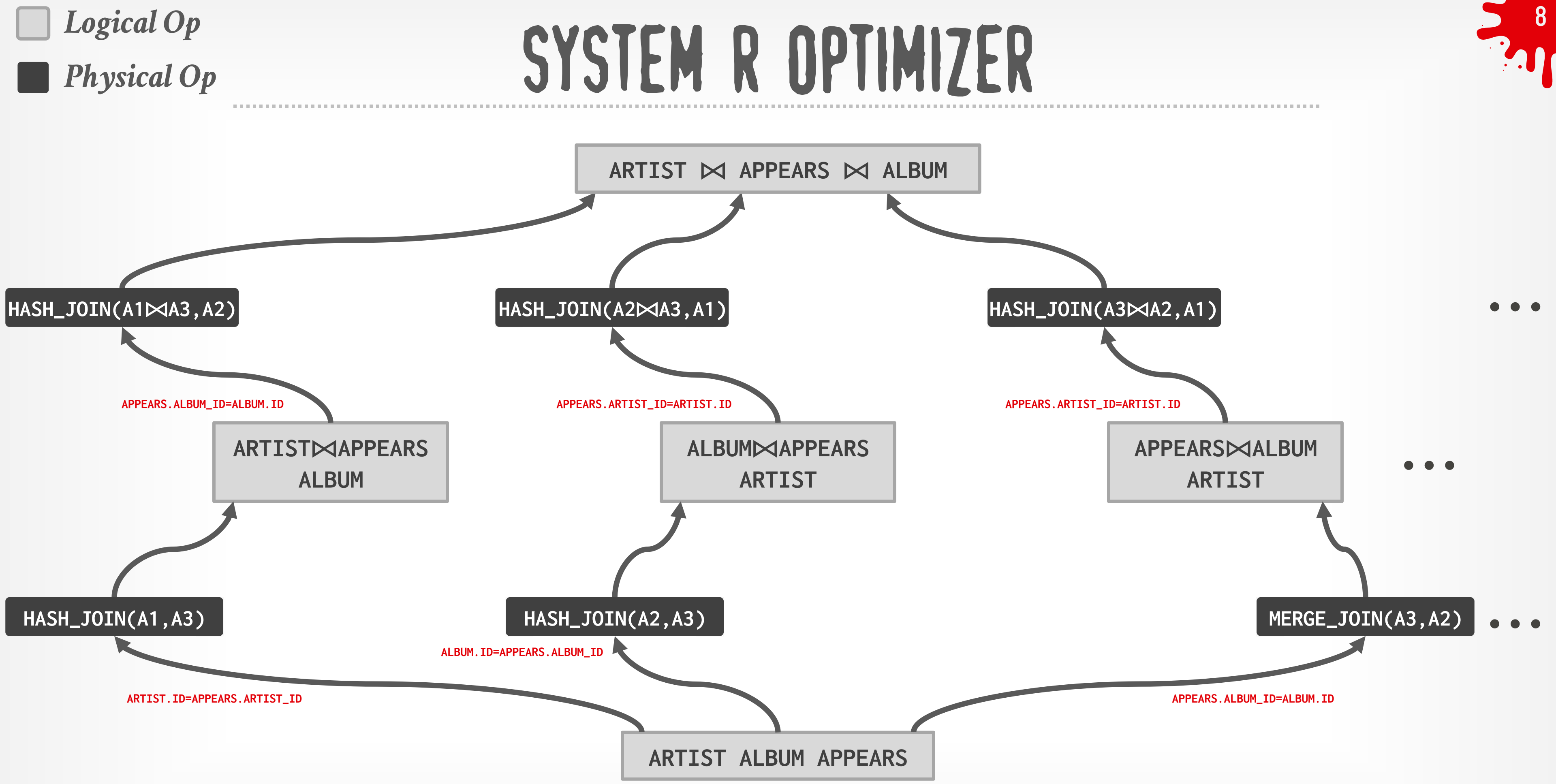
Logical Op

Physical Op

SYSTEM R OPTIMIZER



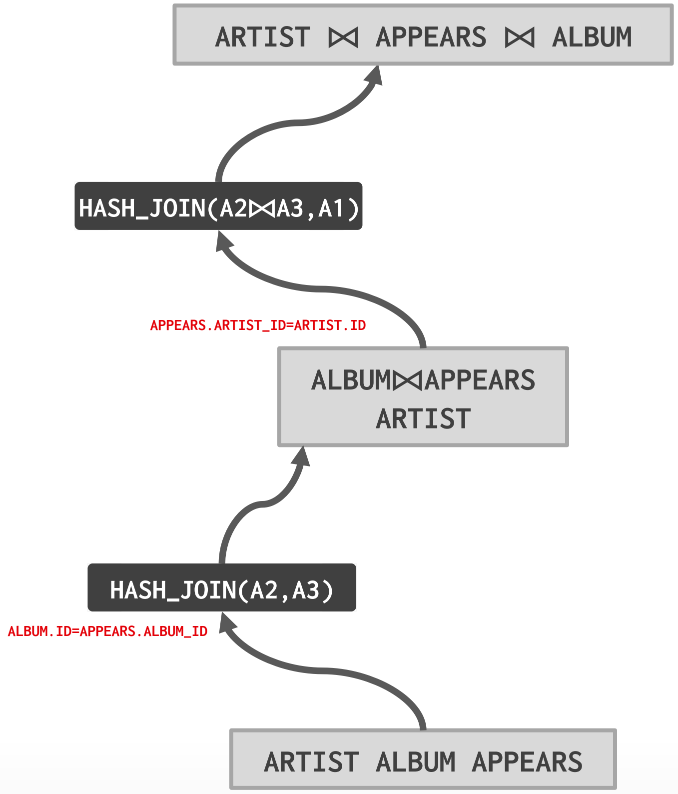
SYSTEM R OPTIMIZER



SYSTEM R OPTIMIZER

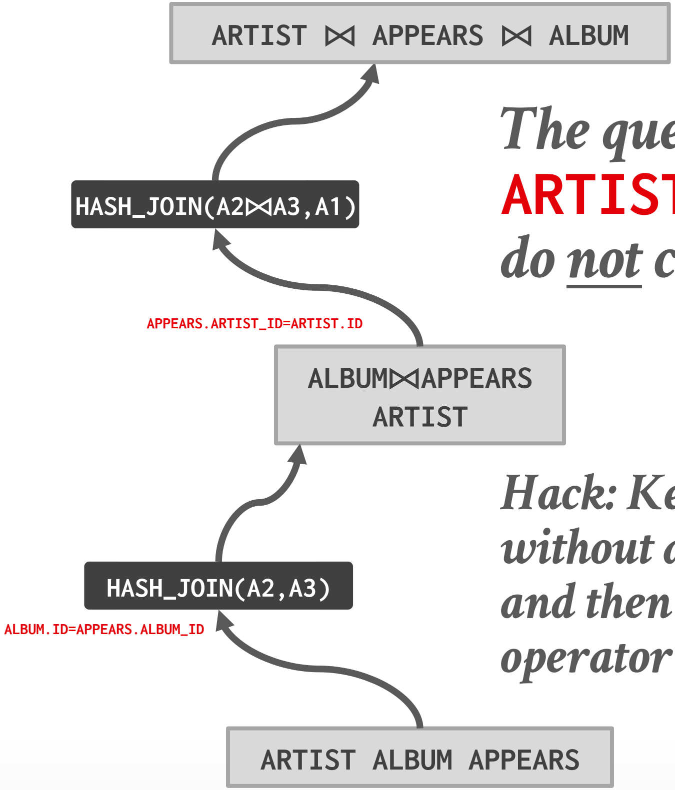
Logical Op

Physical Op



SYSTEM R OPTIMIZER

Logical Op
Physical Op



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 se
→ 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

Logical Op

Physical Op

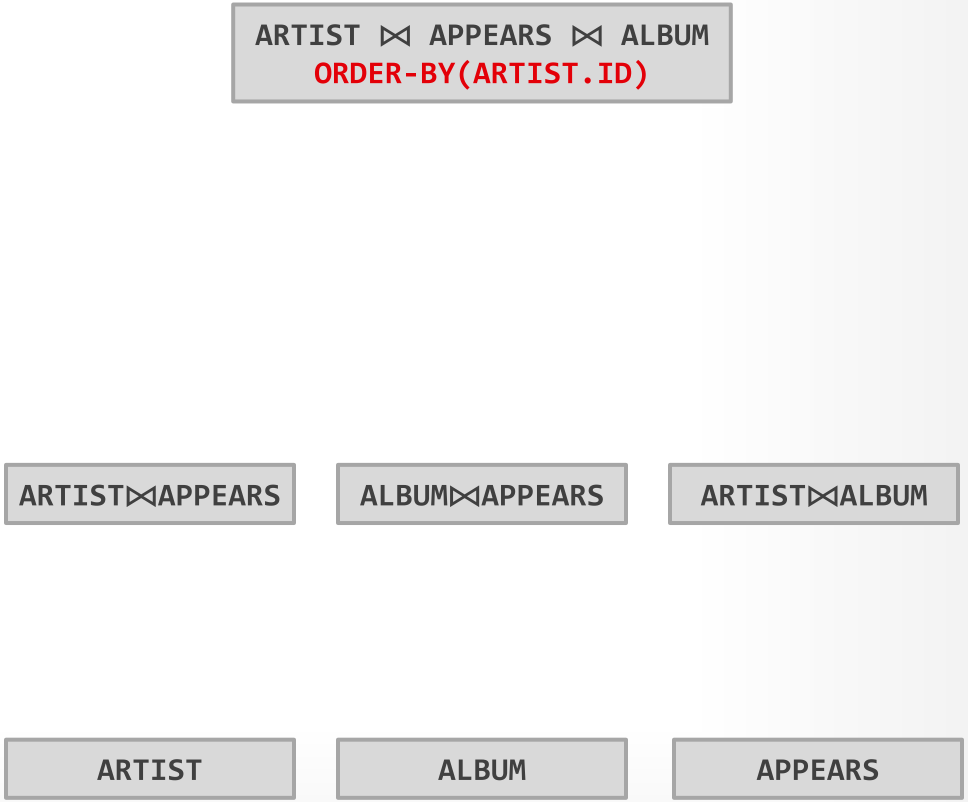
TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.

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

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



Logical Op

Physical Op

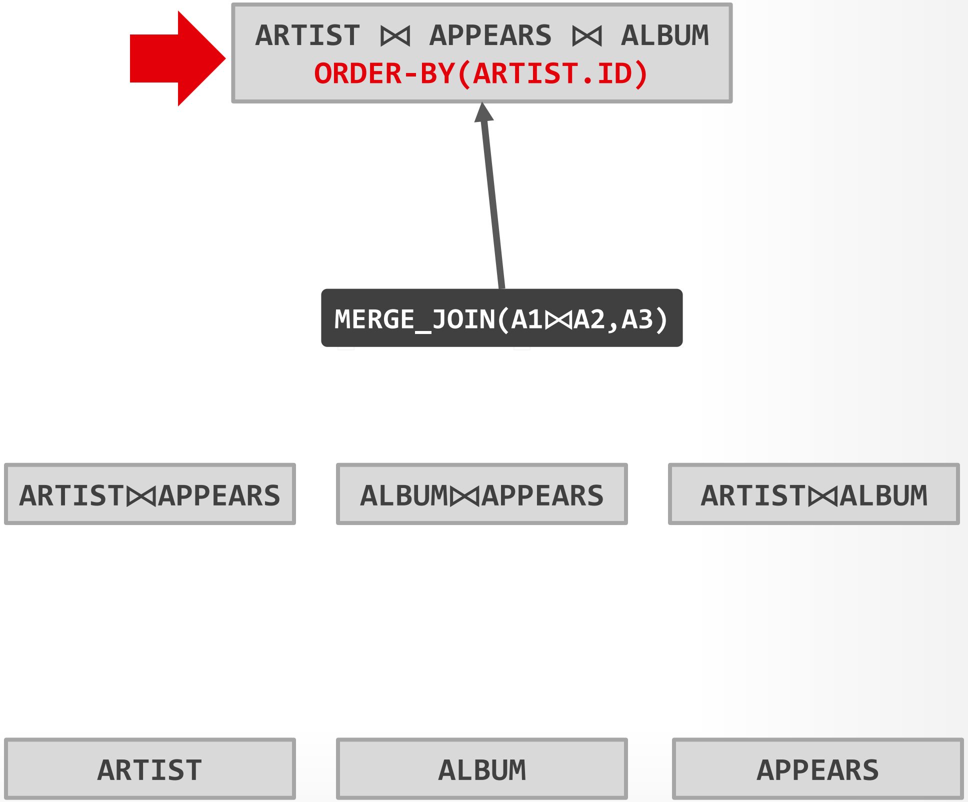
TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.

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

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



Logical Op

Physical Op

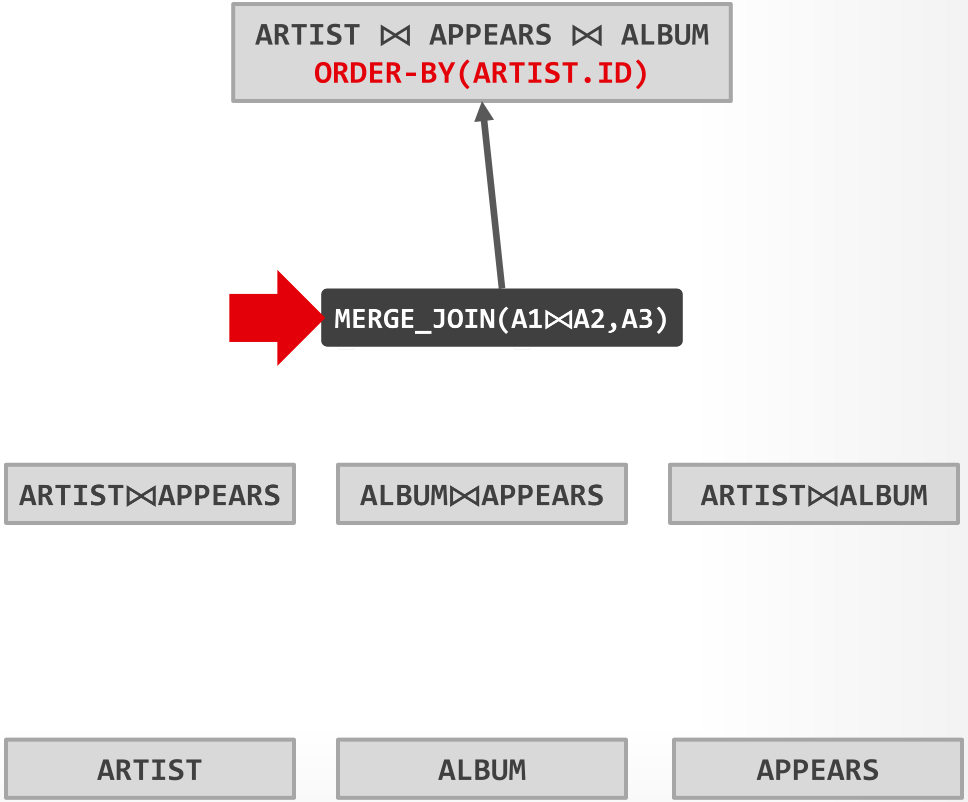
TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.

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

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



Logical Op

Physical Op

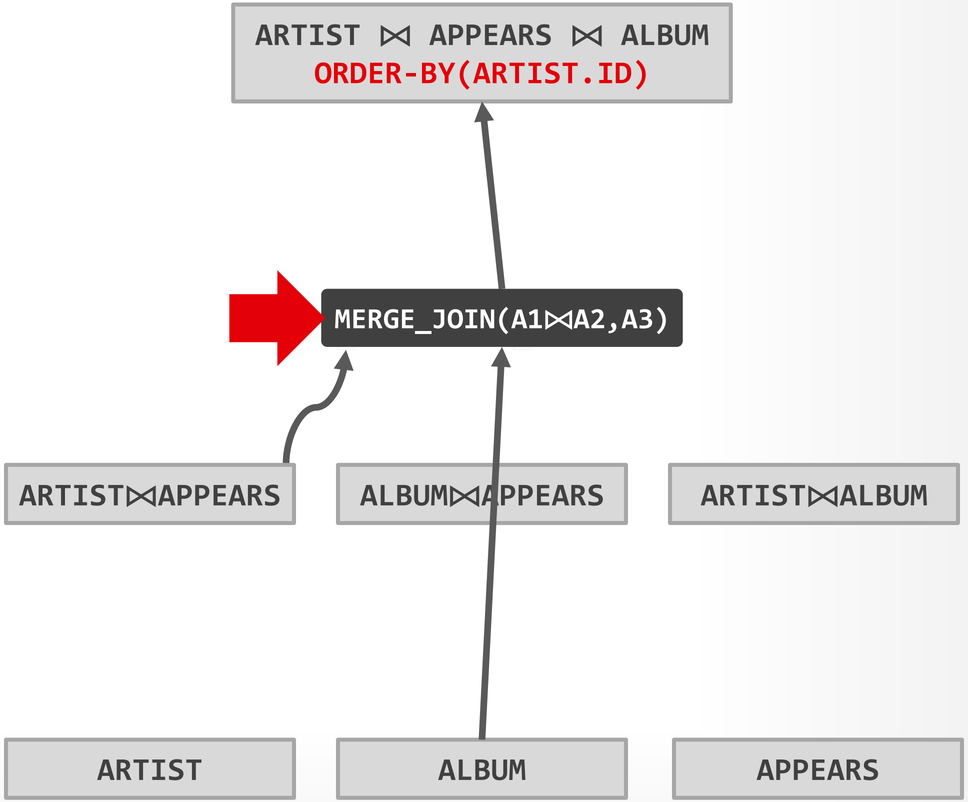
TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.

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

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



Logical Op

Physical Op

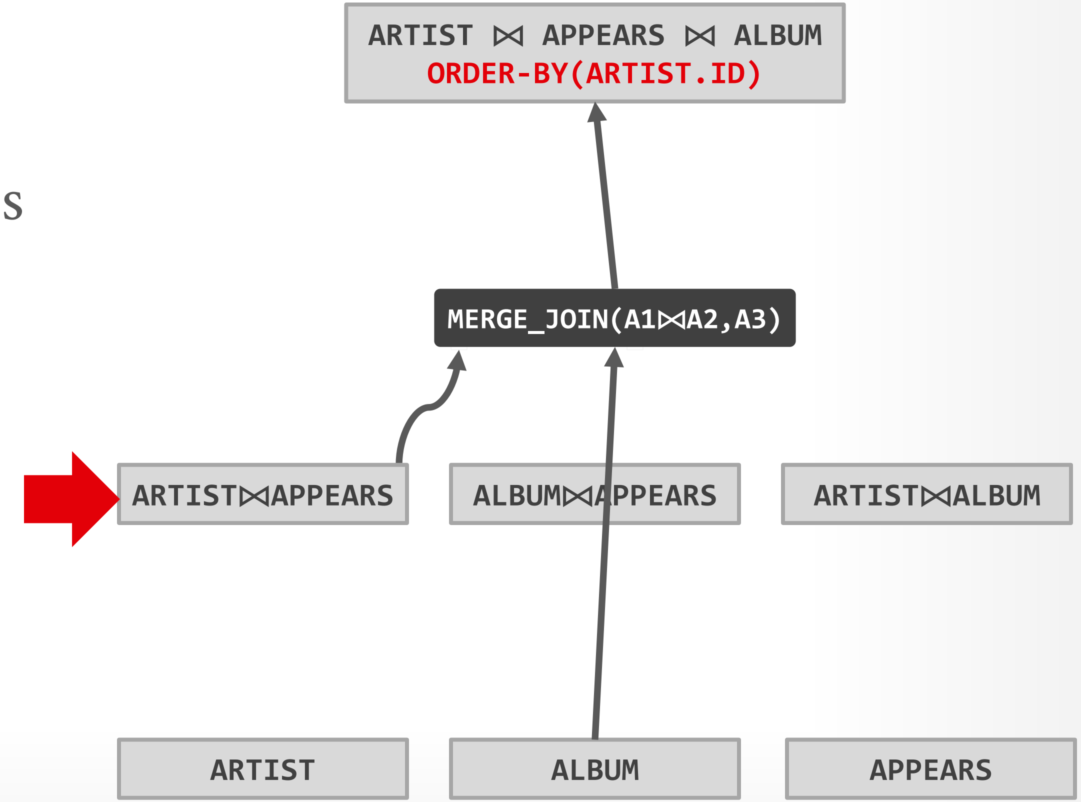
TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.

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

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



Logical Op

Physical Op

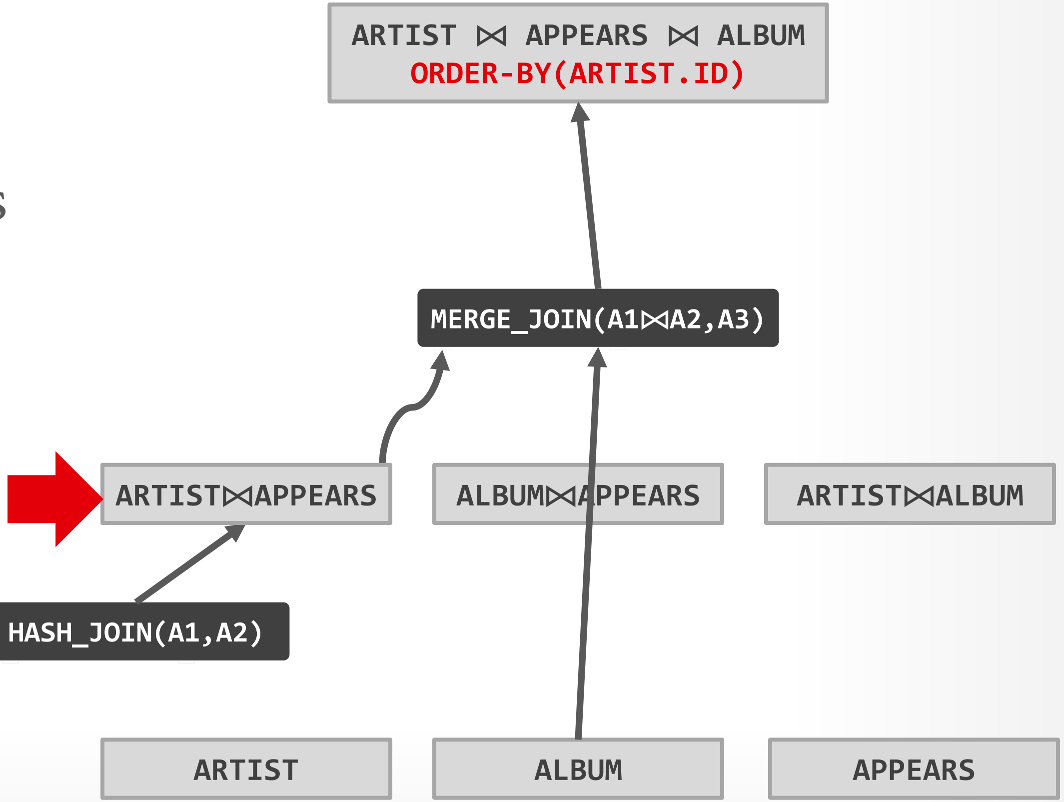
TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.

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

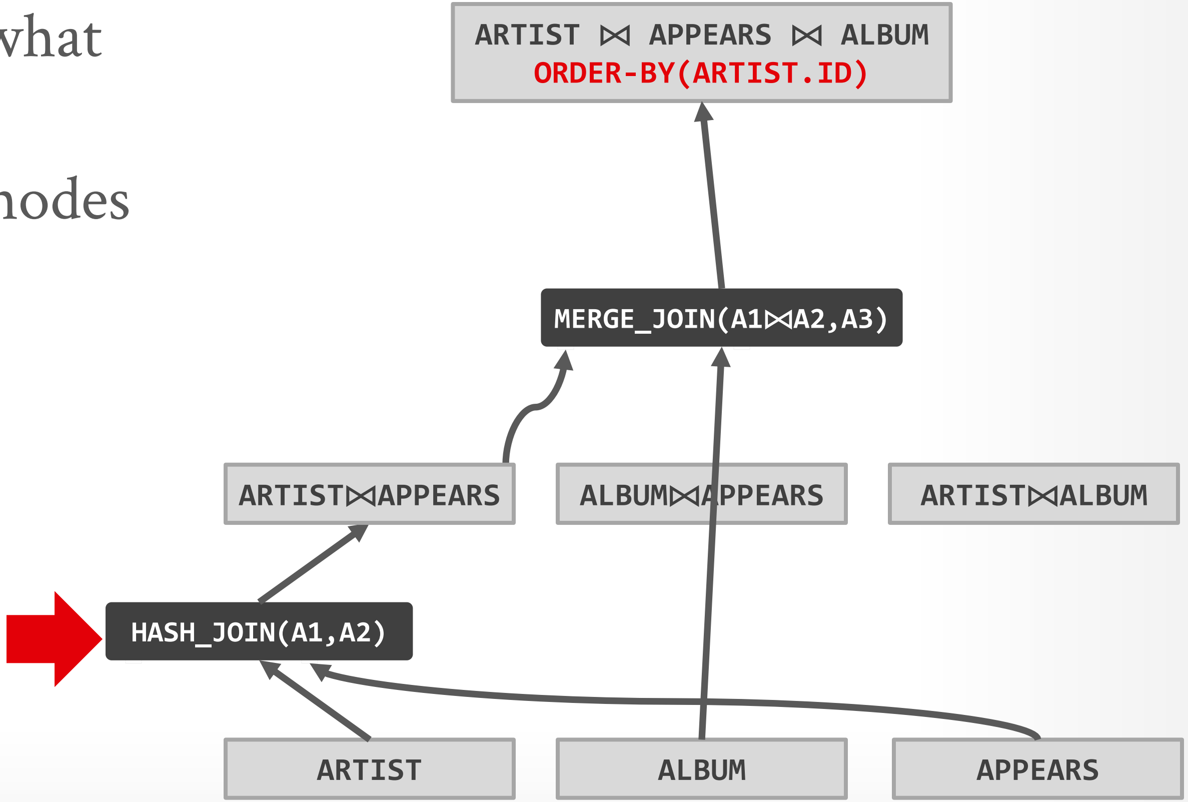
→ **Logical→Physical:** JOIN(A,B) to HASH_JOIN(A,B)



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)



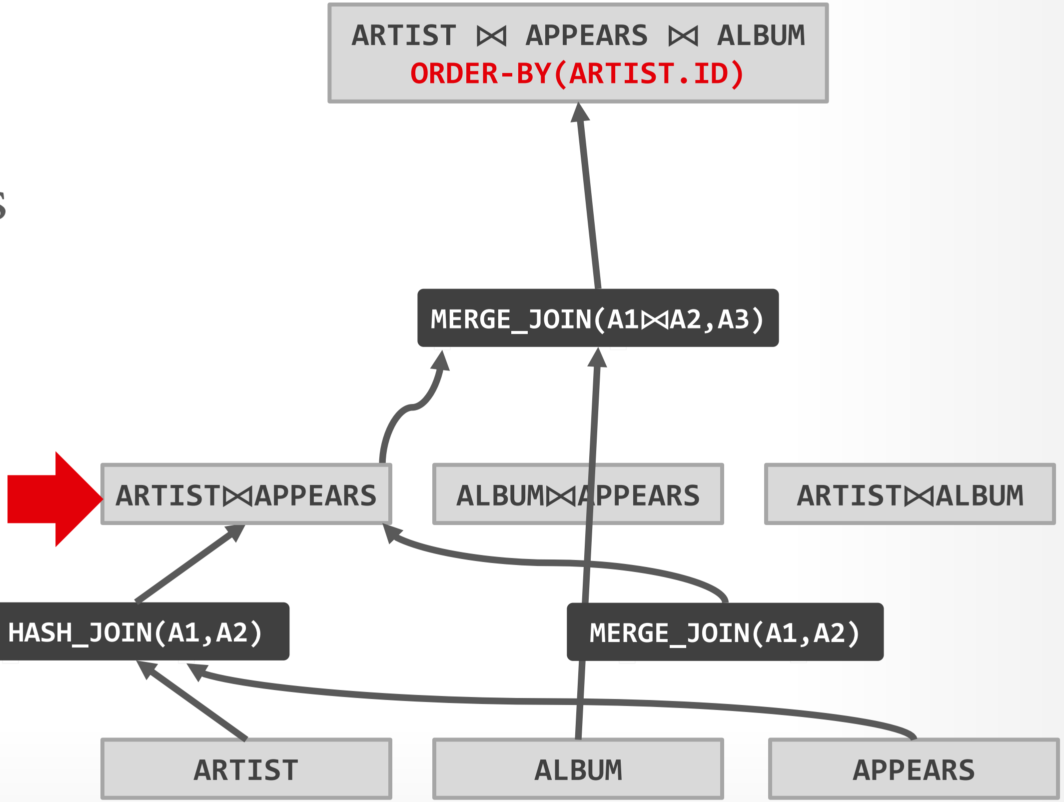
TOP-DOWN OPTIMIZATION

Logical Op
Physical Op

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

Invoke rules to create new nodes and traverse tree.

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



Logical Op

Physical Op

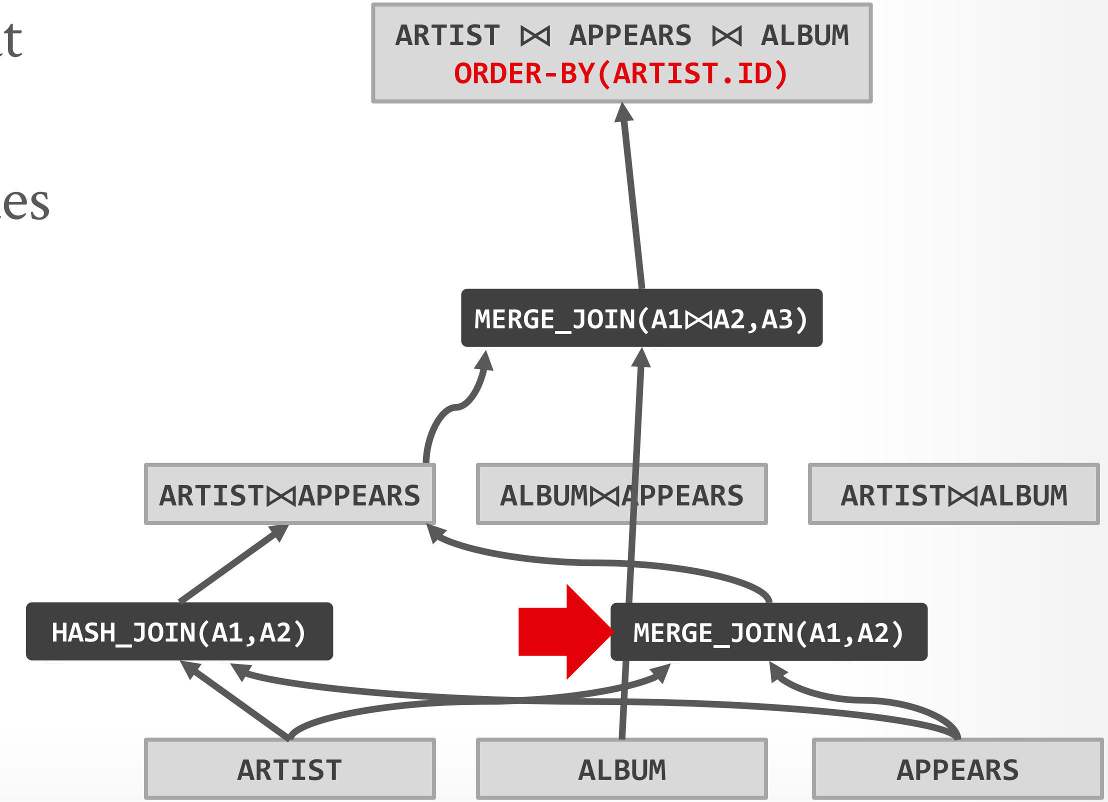
TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.

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

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



Logical Op

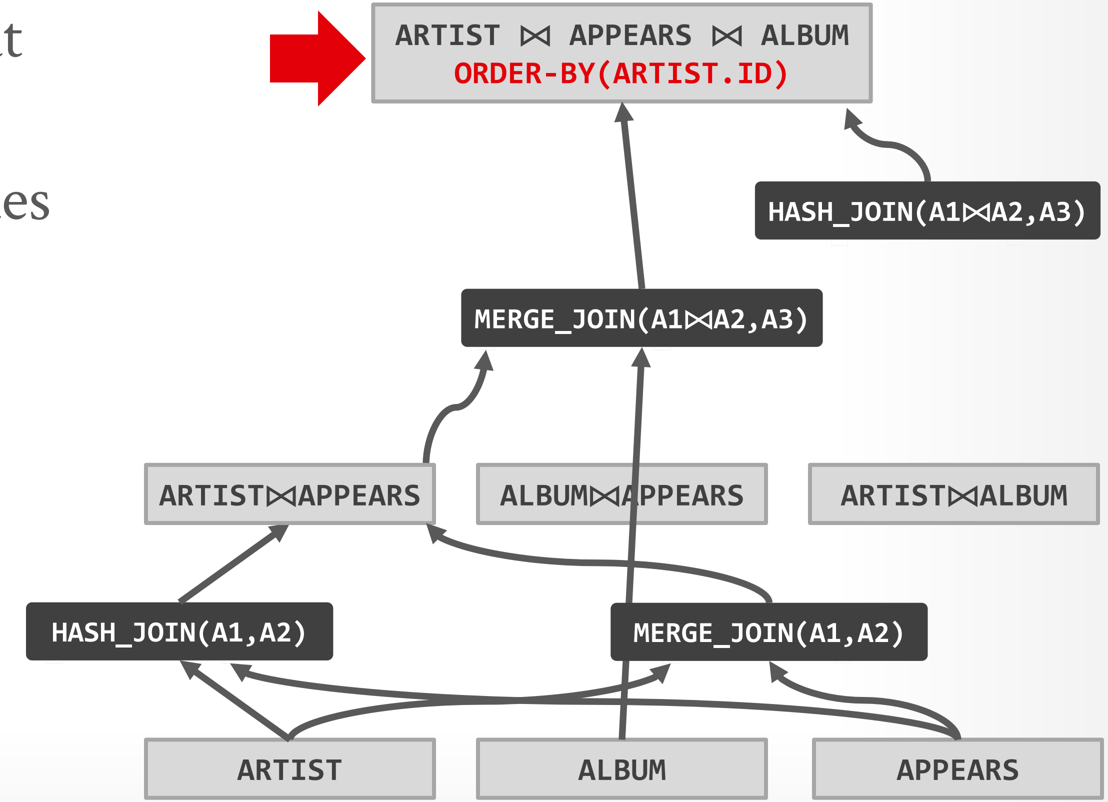
Physical Op

TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.
→ **Logical→Logical:** JOIN(A,B) to JOIN(B,A)
→ **Logical→Physical:** JOIN(A,B) to HASH_JOIN(A,B)

Can create "enforcer" rules that require input to have certain properties.



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

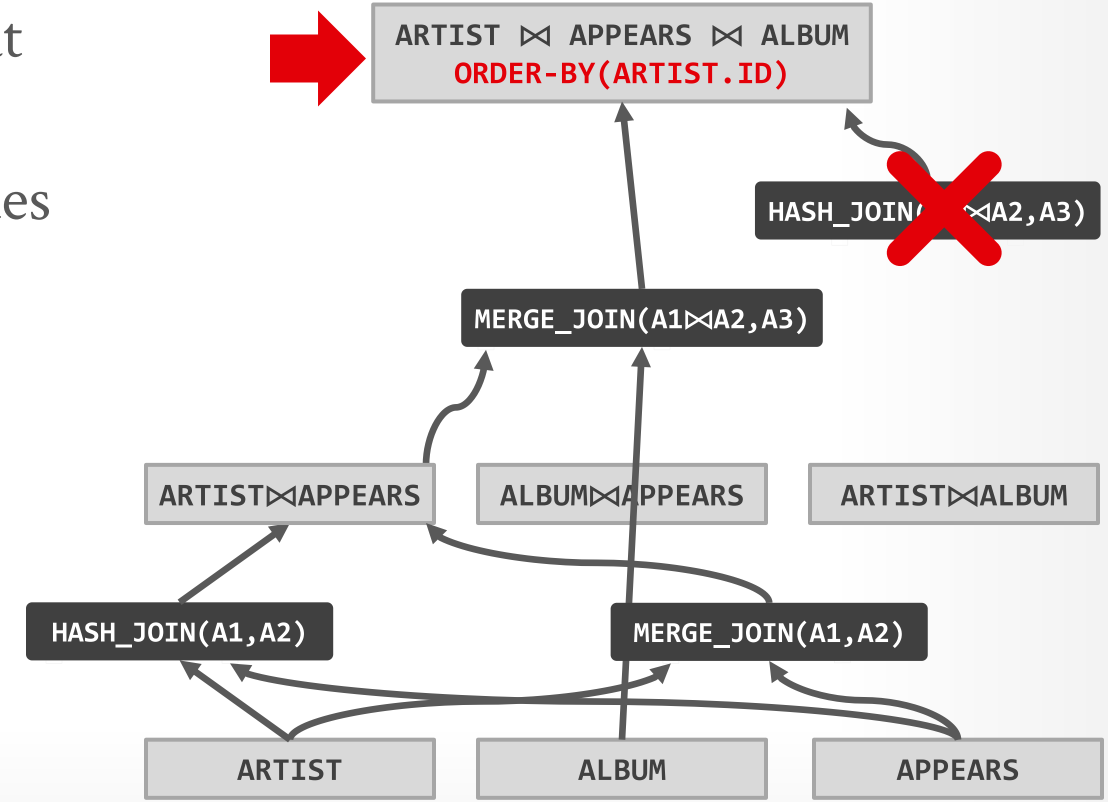
Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

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

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

Can create "enforcer" rules that require input to have certain properties.



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

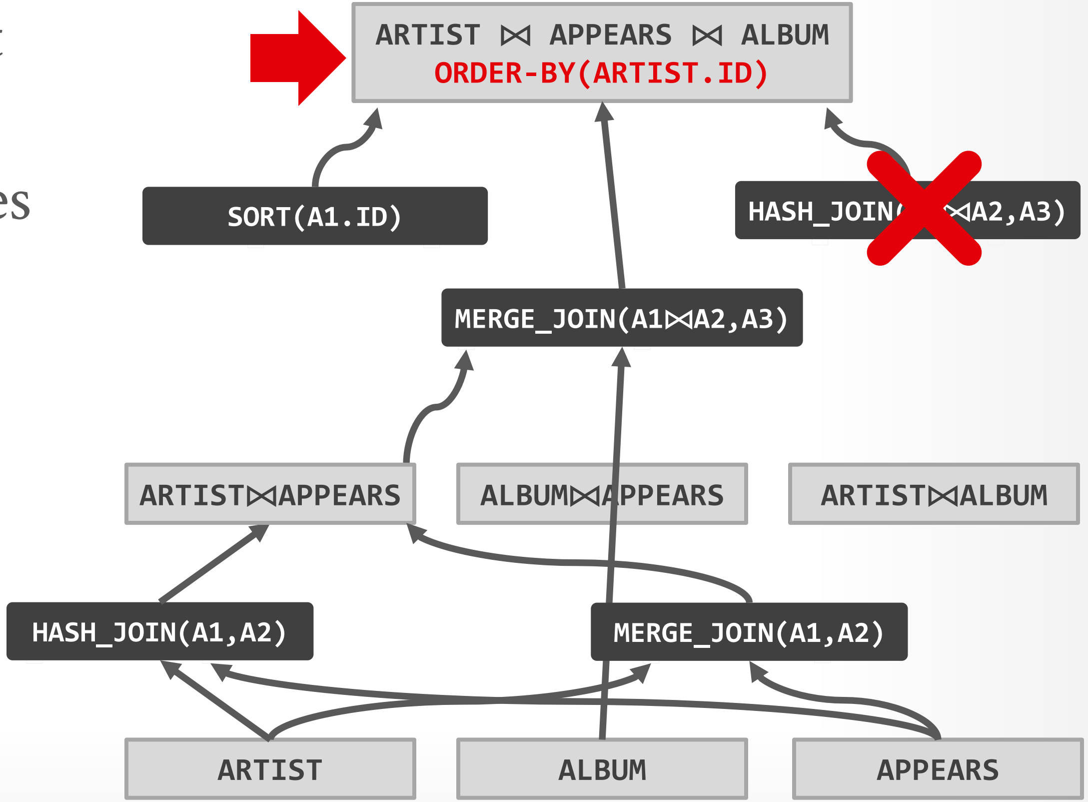
Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

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

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

Can create "enforcer" rules that require input to have certain properties.



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

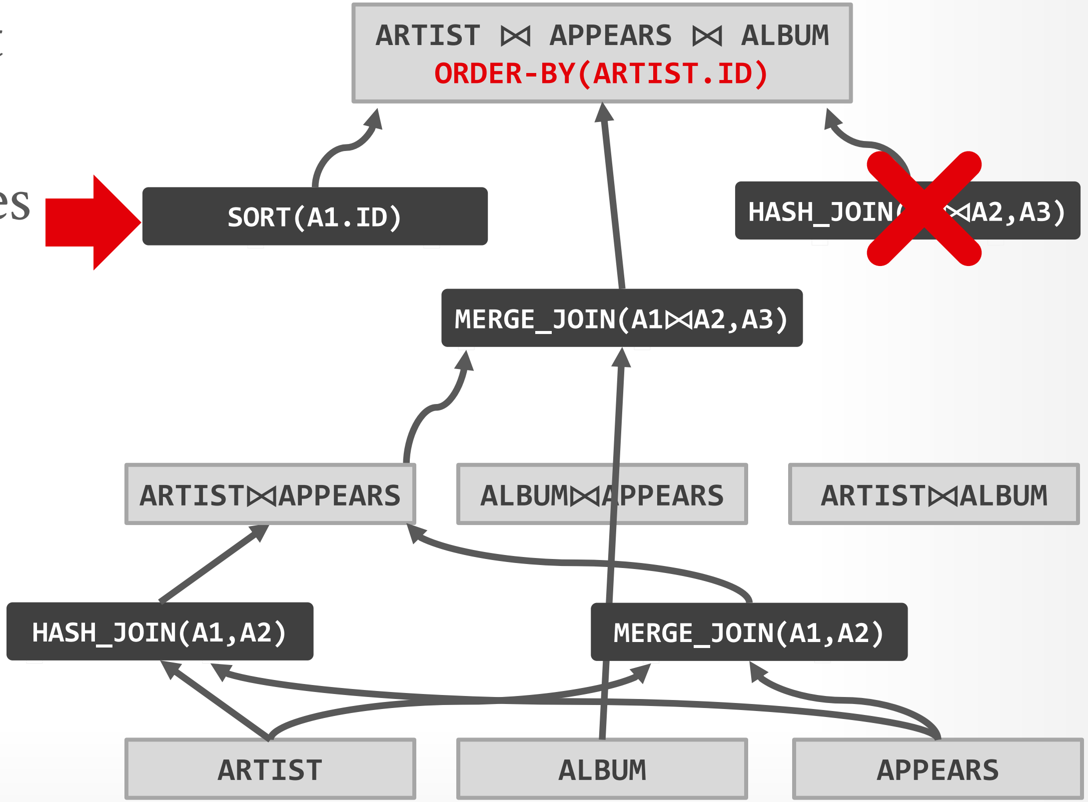
Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

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

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

Can create "enforcer" rules that require input to have certain properties.



Logical Op

Physical Op

TOP-DOWN OPTIMIZATION

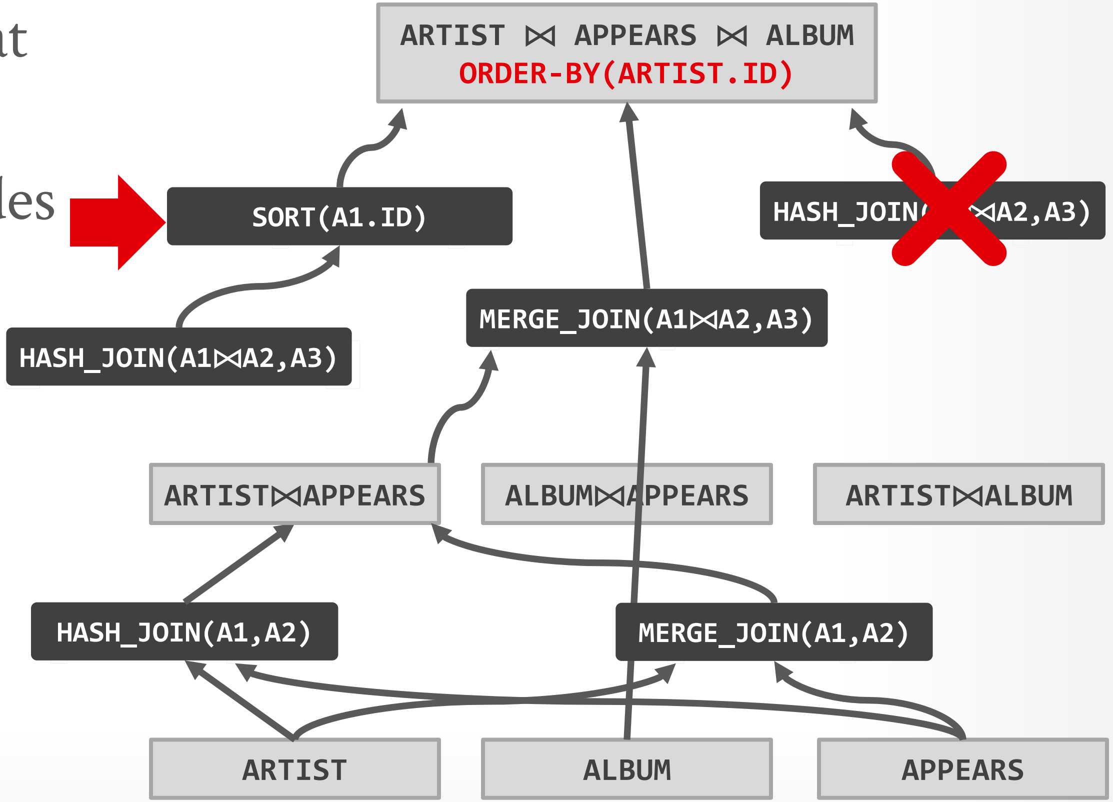
Start with a logical plan of what we want the query to be.

Invoke rules to create new nodes and traverse tree.

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

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

Can create "enforcer" rules that require input to have certain properties.



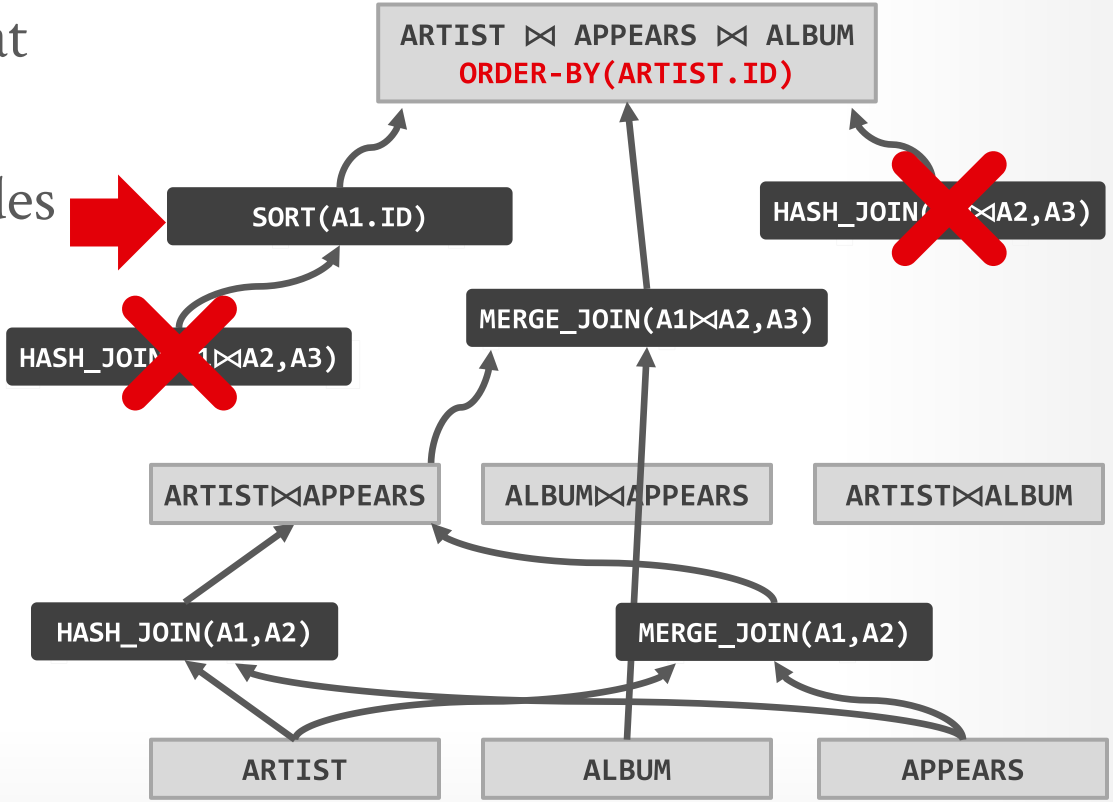
Logical Op
Physical Op

TOP-DOWN OPTIMIZATION

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

Invoke rules to create new nodes and traverse tree.
→ **Logical→Logical:**
JOIN(A,B) to JOIN(B,A)
→ **Logical→Physical:**
JOIN(A,B) to HASH_JOIN(A,B)

Can create "enforcer" rules that require input to have certain properties.



Logical Op
Physical Op
Enforcer

TOP-DOWN OPTIMIZATION: ENFORCERS


```
SELECT * FROM xxx
 WHERE xxx.a > 10 ORDER BY xxx.b;
```

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

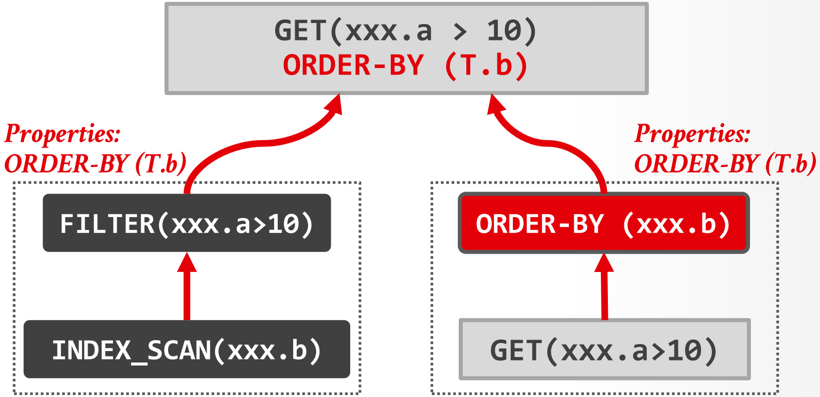
Logical Op
Physical Op
Enforcer

TOP-DOWN OPTIMIZATION: ENFORCERS

```
SELECT * FROM xxx
 WHERE xxx.a > 10 ORDER BY xxx.b;
```

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

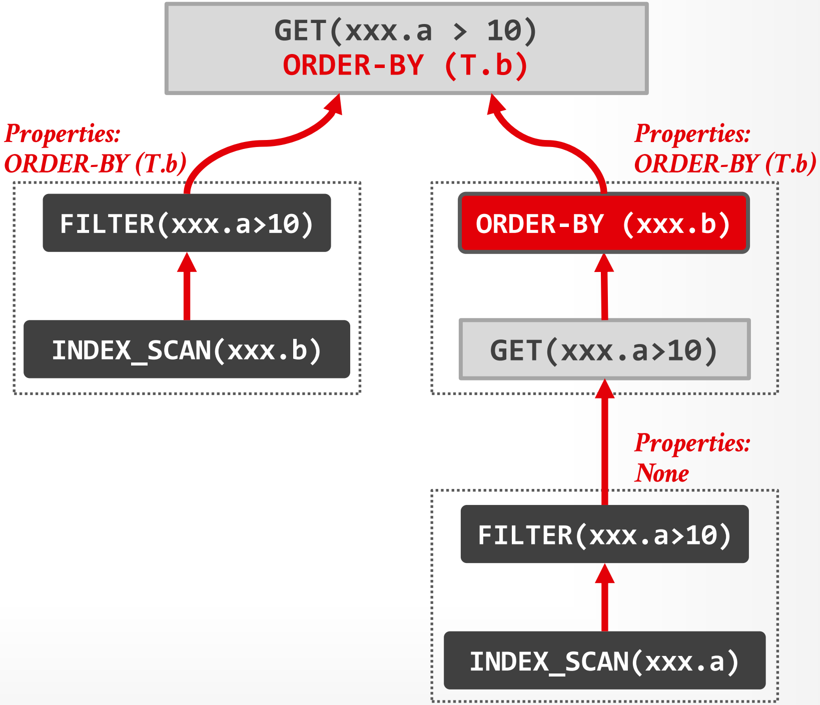
Logical Op
Physical Op
Enforcer

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)**

```
SELECT * FROM xxx
 WHERE xxx.a > 10 ORDER BY xxx.b;
```



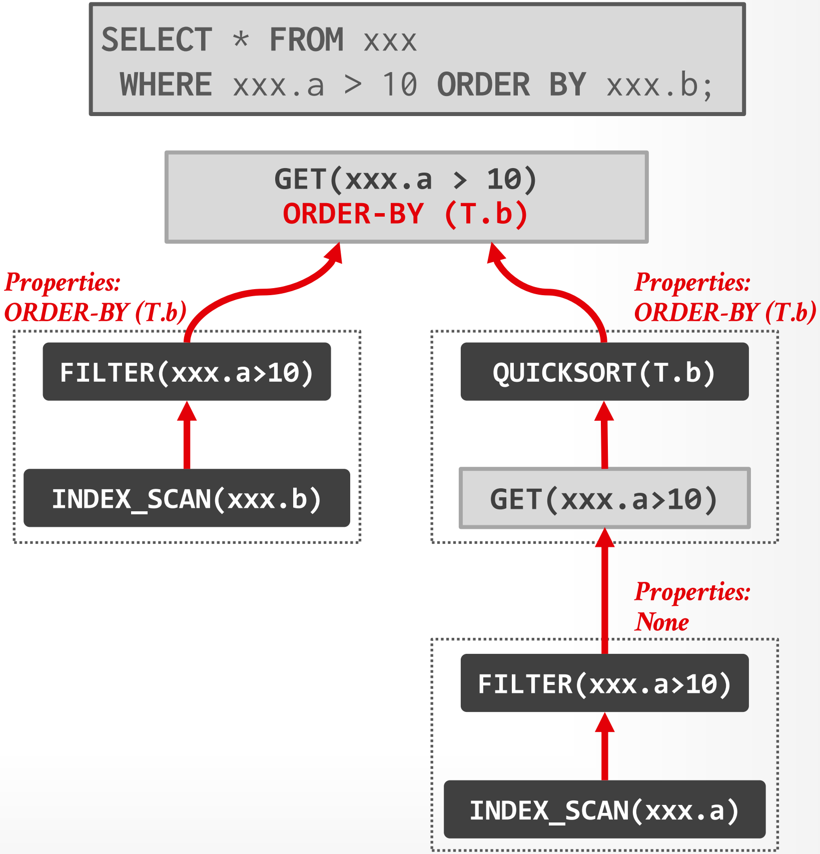
Source: EQOP Book

Logical Op
Physical Op
Enforcer

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)**

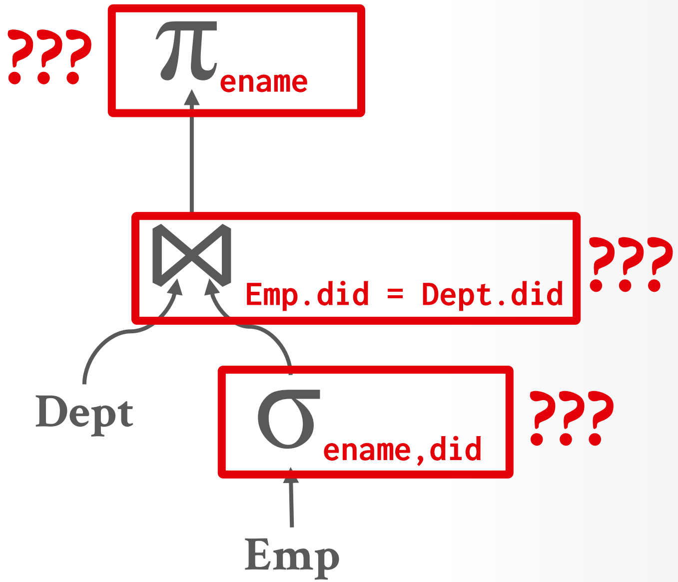


Source: EQOP Book

OBSERVATION

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

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



COST ESTIMATION

The DBMS uses a cost model to predict the behavior of a query plan given a database state.

→ This is an internal cost that allows the DBMS to compare one plan with another.

It is too expensive to run every possible plan to determine this information, so the DBMS need a way to derive this information.

COST MODEL COMPONENTS

Choice #1: Physical Costs

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

→ Depends heavily on hardware.

Choice #2: Logical Costs

→ Estimate output size per operator.

→ Independent of the operator algorithm.

→ Need estimations for operator result sizes.

POSTGRES COST MODEL

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

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

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

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

19.7.2. Planner Cost Constants

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

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

`seq_page_cost` (`floating point`)

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

`random_page_cost` (`floating point`)

STATISTICAL SUMMARIES

Auxiliary data structures that the DBMS populates from scanning the database to allow the optimizer to approximate data contents for different scenarios.

Trade-offs to consider:

→ Accuracy
→ Efficiency
→ Memory Consumption
→ Coverage / Applicability
→ Creation + Maintenance Costs

Source: EQOP Book

STATISTICS STORAGE

Most DBMSs store a database's statistics in its internal catalog.

The DBMS will periodically update statistics according to one or more triggering mechanisms:

→ Periodic Background Tasks (e.g., Postgres Autovacuum)
→ Maintenance Schedules (e.g., Oracle)
→ Modification Thresholds
→ Manual Invocation (e.g., `ANALYZE`, `UPDATE STATISTICS`)

COLUMN STATISTICS

Most DBMSs create single-column statistics for each column in a table.

The DBMS can also track statistics for groups of attributes together rather than just treating them all as independent variables.

→ Some systems automatically build multi-column statistics if they are already used in an index together (MSSQL).
→ Otherwise, a human manually specifies target columns.
→ Also called Column Group Statistics (Db2) or Extended Statistics (Oracle).

SUMMARIZATION APPROACHES

Choice #1: Histograms ⬅ *Most Common*

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

Choice #2: Sketches ⬅ *Increasing Usage*

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

Choice #3: Sampling ⬅ *Rare*

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

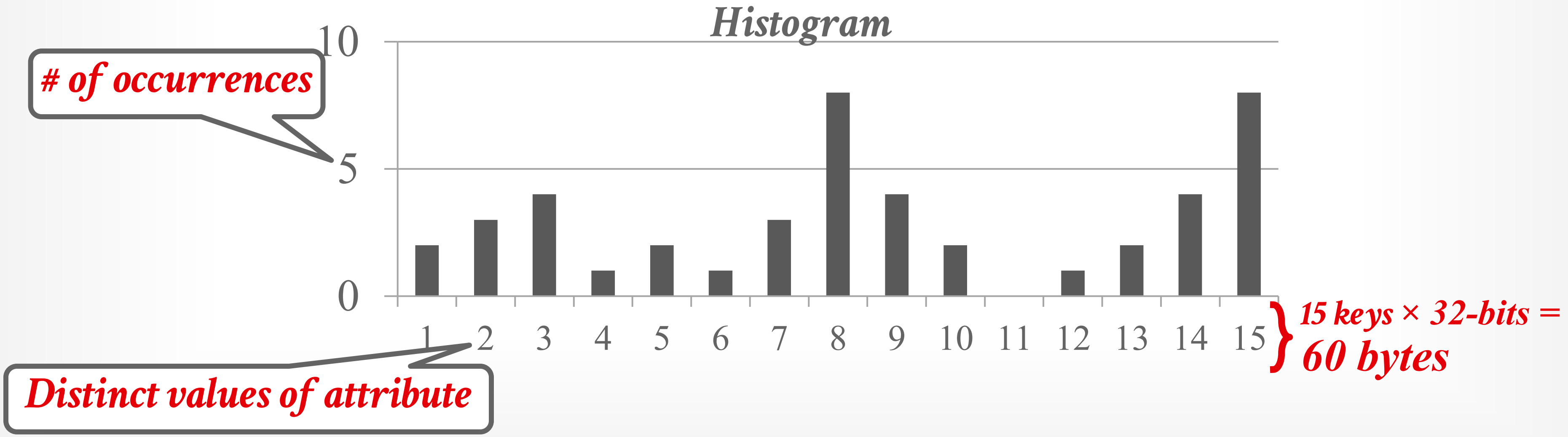
Choice #4: ML Model ⬅ *Experimental / Very Rare*

→ Train an ML model that learns the selectivity of predicates and correlations between multiple tables.

HISTOGRAMS

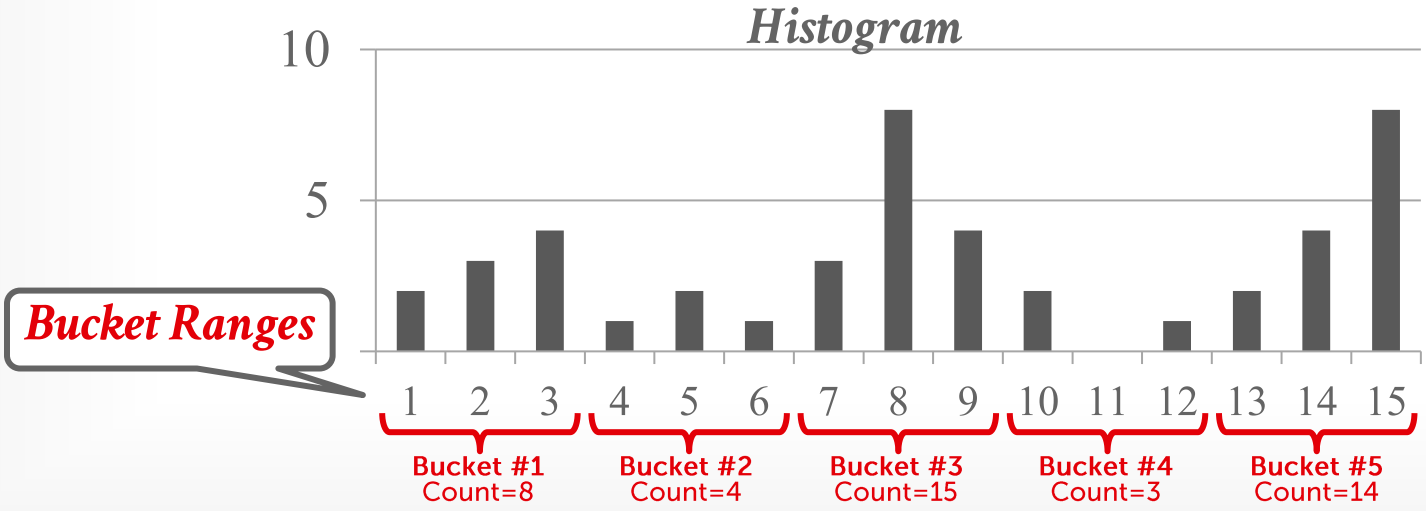
Approximate the distribution of values in a column for cardinality estimation.

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



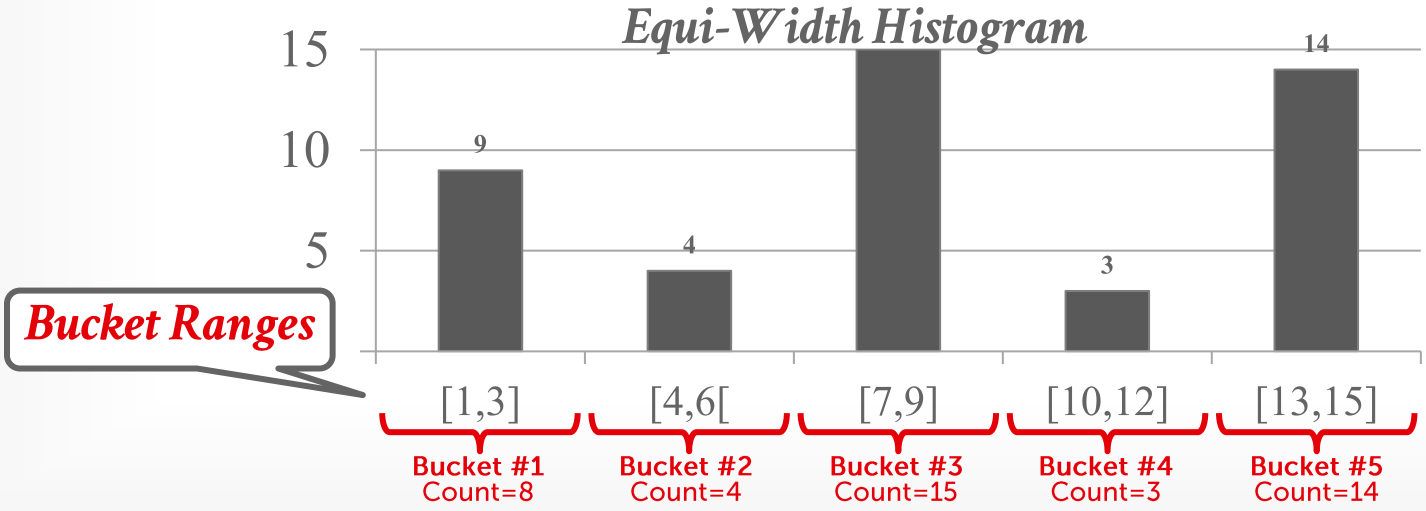
EQUI-WIDTH HISTOGRAM

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



EQUI-WIDTH HISTOGRAM

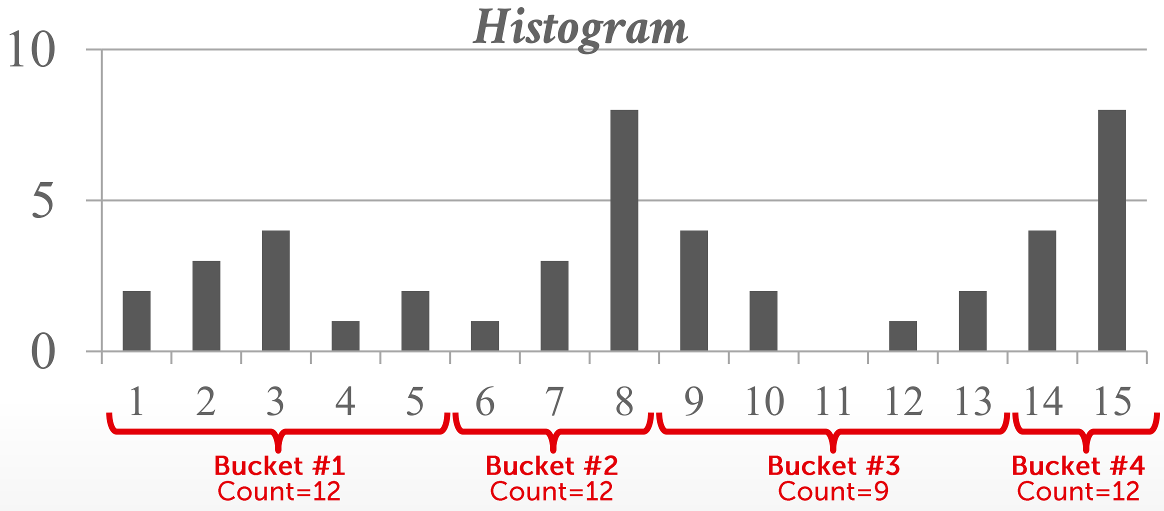
Maintain counts for a group of values instead of each unique key. All buckets have the same width (i.e., same # of value).



EQUI-DEPTH HISTOGRAMS

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

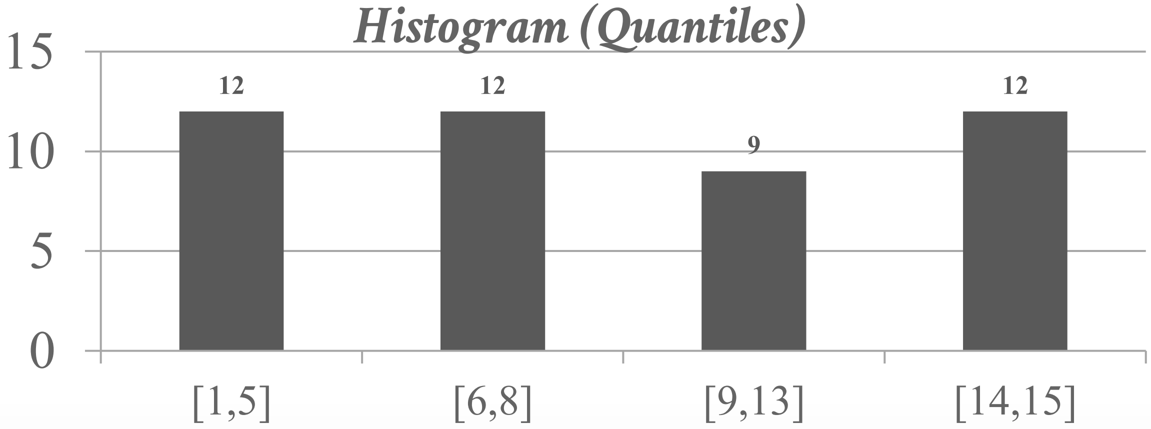
→ Equi-depth histograms are shown to have better worst-case and average error than equi-width histograms.



EQUI-DEPTH HISTOGRAMS

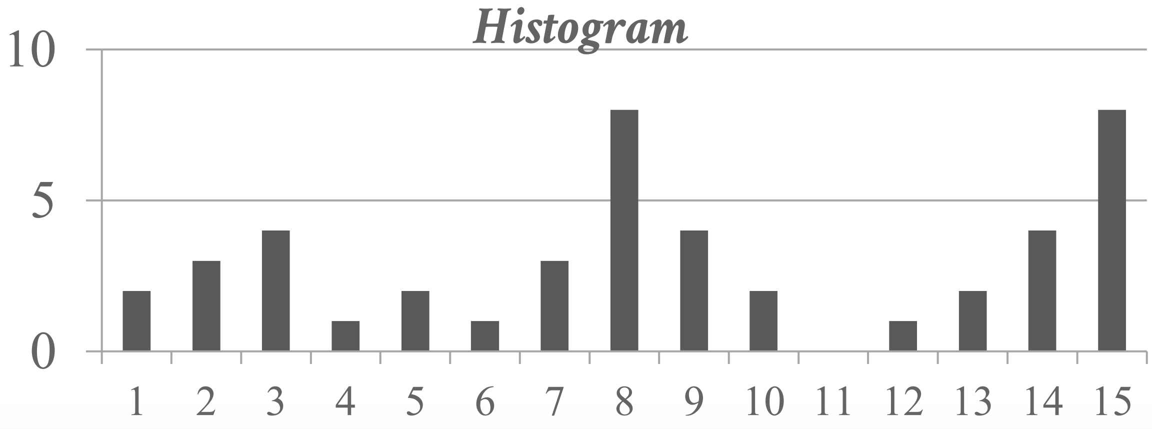
Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.

→ Equi-depth histograms are shown to have better worst-case and average error than equi-width histograms.



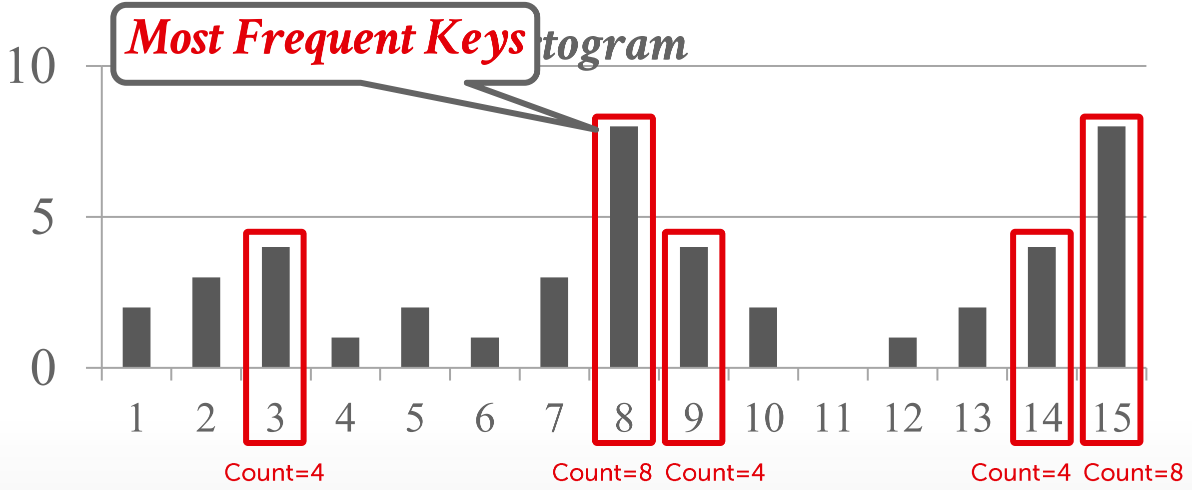
END-BIASED HISTOGRAMS

Use ***N-1*** buckets to store the exact count for the most frequent keys. The last bucket (**R**) stores the average frequency of all remaining values.



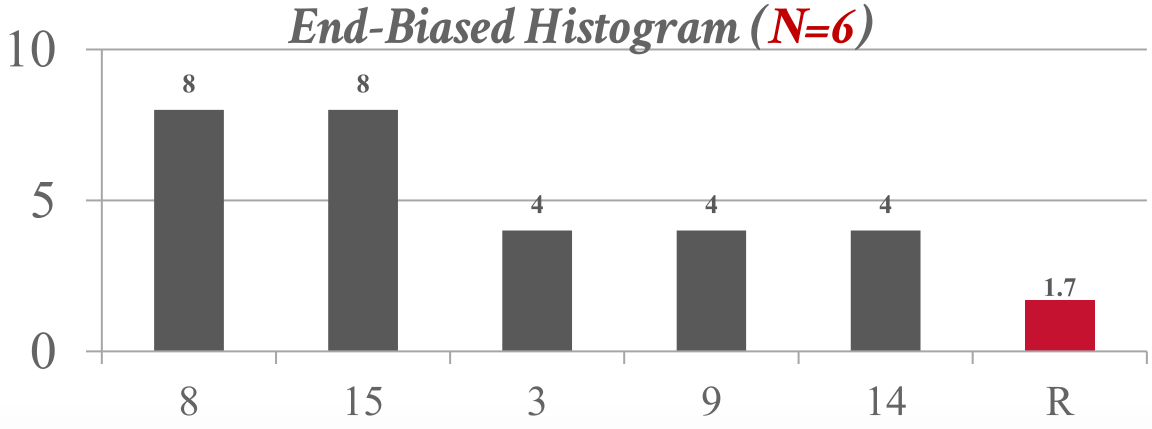
END-BIASED HISTOGRAMS

Use ***N-1*** buckets to store the exact count for the most frequent keys. The last bucket (**R**) stores the average frequency of all remaining values.



END-BIASED HISTOGRAMS

Use ***N-1*** buckets to store the exact count for the most frequent keys. The last bucket (**R**) stores the average frequency of all remaining values.



SKETCHES

Maintaining exact statistics about the database is expensive and slow.

Use probabilistic data structures called **sketches** to generate error-bounded estimates.

→ Frequent Items (Count-min Sketch)
→ Count Distinct (HyperLogLog)
→ Quantiles (t-digest)

Open-source implementations are available (Apache DataSketches, Google ZetaSketch)

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch

	0	1	2	3	4	5	6	7
$hash_1$	0	0	0	0	0	0	0	0
$hash_2$	0	0	0	0	0	0	0	0
$hash_3$	0	0	0	0	0	0	0	0
$hash_4$	0	0	0	0	0	0	0	0

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch

	0	1	2	3	4	5	6	7
$hash_1$	0	0	0	0	0	0	0	0
$hash_2$	0	0	0	0	0	0	0	0
$hash_3$	0	0	0	0	0	0	0	0
$hash_4$	0	0	0	0	0	0	0	0

```
INSERT 'ODB'
```

$$\left\{\begin{array}{l} hash_1(\text{'ODB'}) = 9022 \,\%\, 8 = 6 \\ hash_2(\text{'ODB'}) = 1412 \,\%\, 8 = 4 \\ hash_3(\text{'ODB'}) = 4211 \,\%\, 8 = 3 \\ hash_4(\text{'ODB'}) = 5000 \,\%\, 8 = 0 \end{array}\right.$$

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch

	0	1	2	3	4	5	6	7
$hash_1$	0	0	0	0	0	0	+1	0
$hash_2$	0	0	0	0	0	0	0	0
$hash_3$	0	0	0	0	0	0	0	0
$hash_4$	0	0	0	0	0	0	0	0

```
INSERT 'ODB'
```

$hash_1(\text{'ODB'}) = 9022 \% 8 = 6$

$hash_2(\text{'ODB'}) = 1412 \% 8 = 4$

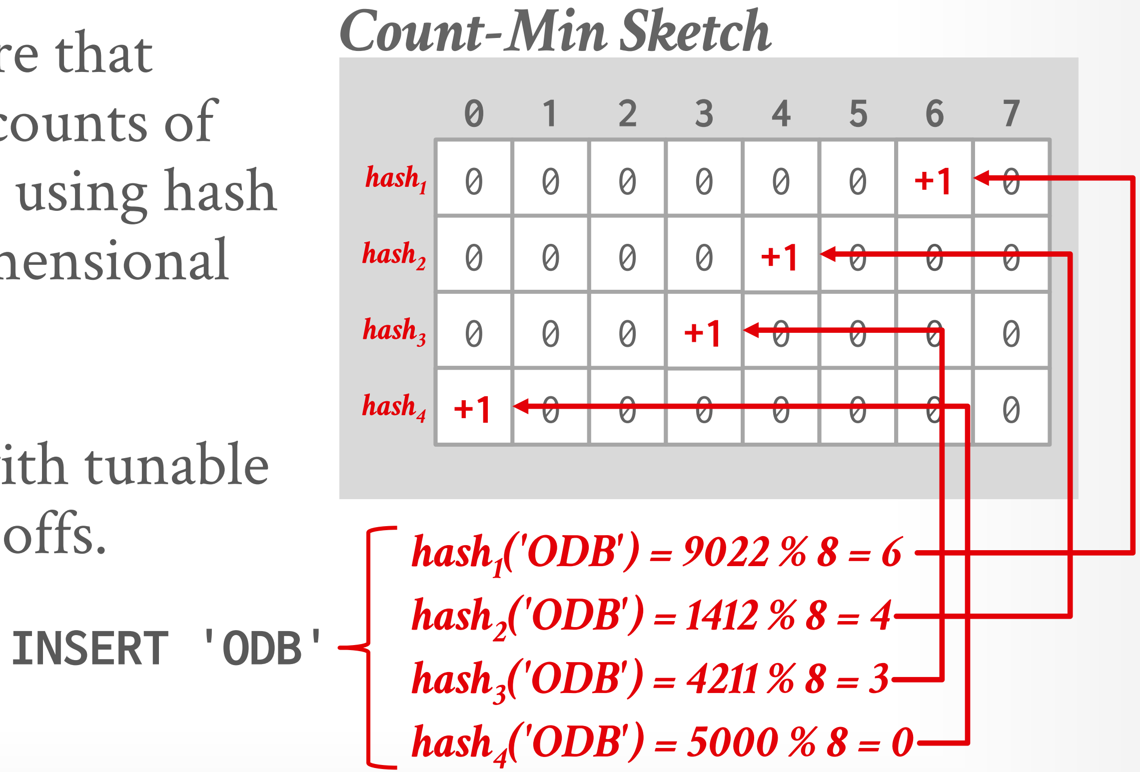
$hash_3(\text{'ODB'}) = 4211 \% 8 = 3$

$hash_4(\text{'ODB'}) = 5000 \% 8 = 0$

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.



COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch

	0	1	2	3	4	5	6	7
$hash_1$	0	+10	0	0	0	+2	+2	0
$hash_2$	0	0	+2	0	+3	+8	0	+1
$hash_3$	+1	0	+6	+6	0	0	+1	0
$hash_4$	+3	0	0	+4	0	+5	0	+2

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch

	0	1	2	3	4	5	6	7
$hash_1$	0	+10	0	0	0	+2	+2	0
$hash_2$	0	0	+2	0	+3	+8	0	+1
$hash_3$	+1	0	+6	+6	0	0	+1	0
$hash_4$	+3	0	0	+4	0	+5	0	+2

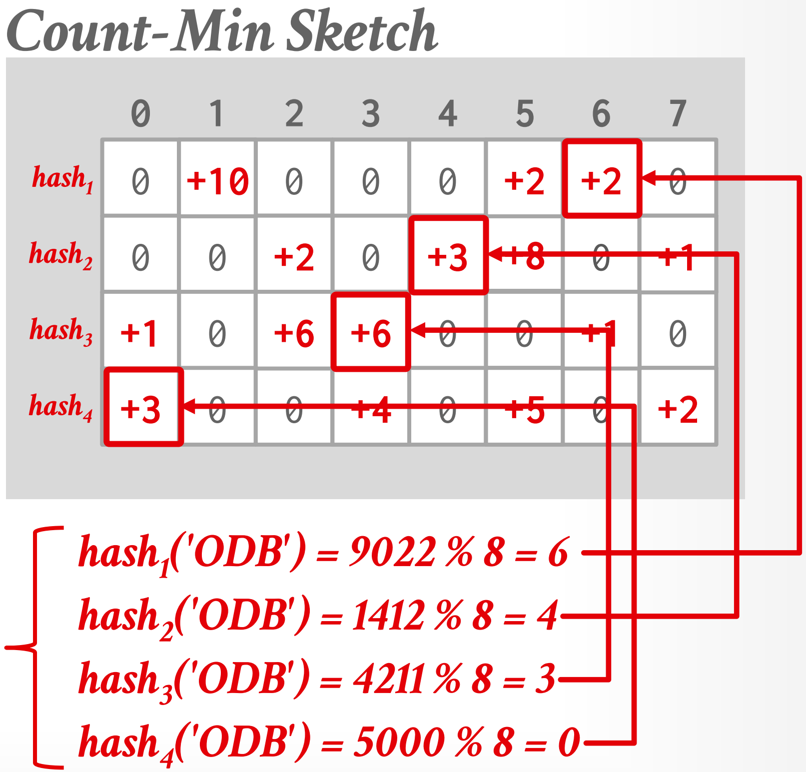
GET 'ODB'

$$\left\{\begin{array}{l} hash_1(\text{'ODB'}) = 9022 \% 8 = 6 \\ hash_2(\text{'ODB'}) = 1412 \% 8 = 4 \\ hash_3(\text{'ODB'}) = 4211 \% 8 = 3 \\ hash_4(\text{'ODB'}) = 5000 \% 8 = 0 \end{array}\right.$$

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.




```
GET 'ODB'
```

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch

	0	1	2	3	4	5	6	7
$hash_1$	0	+10	0	0	0	+2	+2	0
$hash_2$	0	0	+2	0	+3	+8	0	+1
$hash_3$	+1	0	+6	+6	0	0	+1	0
$hash_4$	+3	0	0	+4	0	+5	0	+2

`GET 'ODB'` $\left\{ Min(2,3,6,3) = 2 \right.$

HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

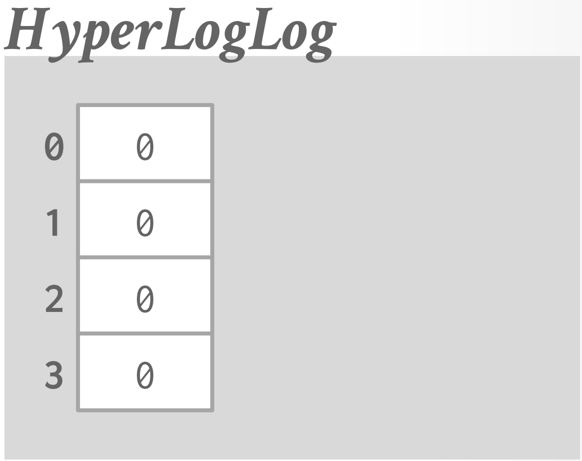
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.

HyperLogLog

0	0
1	0
2	0
3	0

```
INSERT 'ODB'
```

hash('ODB') = 9022

0010001100111110

HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

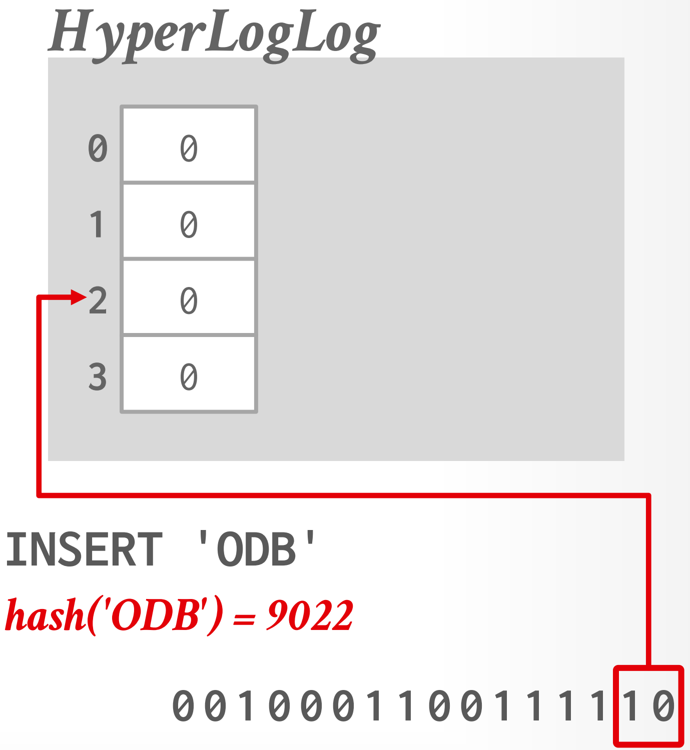
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

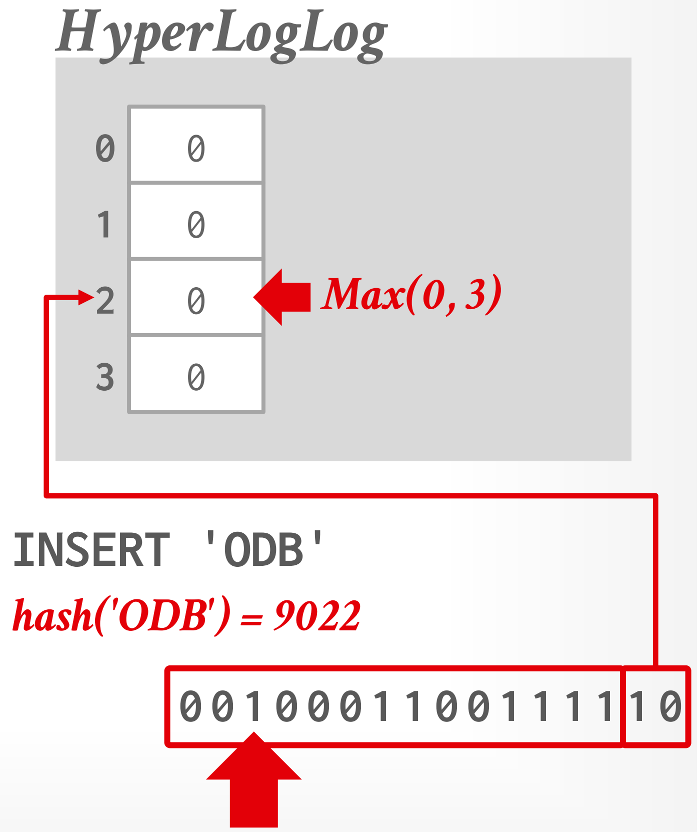
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.

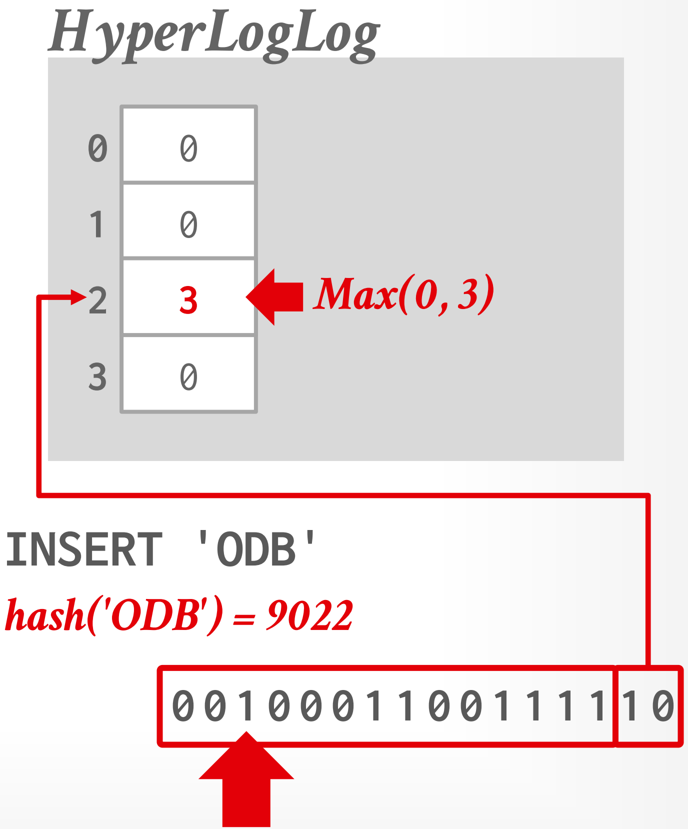


HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.
- → Store ***m*** fixed-size array of counters.

Update:
- → The first ***b*** bits of the hash determine which counter to update.
- → Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:
- → Compute the Harmonic mean across counters and correct with a corrective fudge factor.

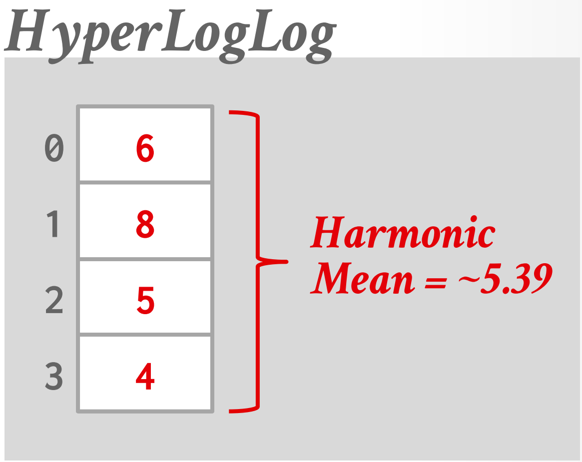


HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.
→ Store ***m*** fixed-size array of counters.

Update:
→ The first ***b*** bits of the hash determine which counter to update.
→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:
→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



SAMPLING

Execute a predicate on a random sample of the target data set. The number of tuples to examine depends on the size of the original table.

Approach #1: Maintain Read-Only Copy

→ Periodically refresh to maintain accuracy.

Approach #2: Sample Real Tables

→ May read multiple versions of same logical tuple.

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	64	Rested
1002	Swift	35	Engaged
1003	Tupac	25	Dead
1004	Bieber	31	Crunk
1005	DJ Cache	21	Paid
1006	TigerKing	62	Jailed

⋮

1 billion tuples

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	64	Rested
1002	Swift	35	Engaged
1003	Tupac	25	Dead
1004	Bieber	31	Crunk
1005	DJ Cache	21	Paid
1006	TigerKing	62	Jailed

⋮

1 billion tuples

Table Sample

1001	Obama	64	Rested
1003	Tupac	25	Dead
1005	DJ Cache	21	Paid

sel(age>50) = 1/3

CARDINALITY ESTIMATION

Estimate the number of rows that a query operator will produce, such as a filter or join, to help the optimizer choose the most efficient execution plan.

There are three cardinality estimations an optimizer must support as the core of its cost model:
→ Selection Conditions (filters)
→ Join Size Estimation
→ Distinct Value Estimation

Source: EQOP Book

DERIVABLE STATISTICS

For each relation ***R***, the DBMS maintains statistics to approximate the following information:

→ N_R: Number of tuples in ***R***.
→ ***V(A,R)***: Number of distinct values for attribute ***A***.

The **selection cardinality** ***SC(A,R)*** is the average number of tuples with a value for an attribute ***A*** given $N_R / V(A,R)$

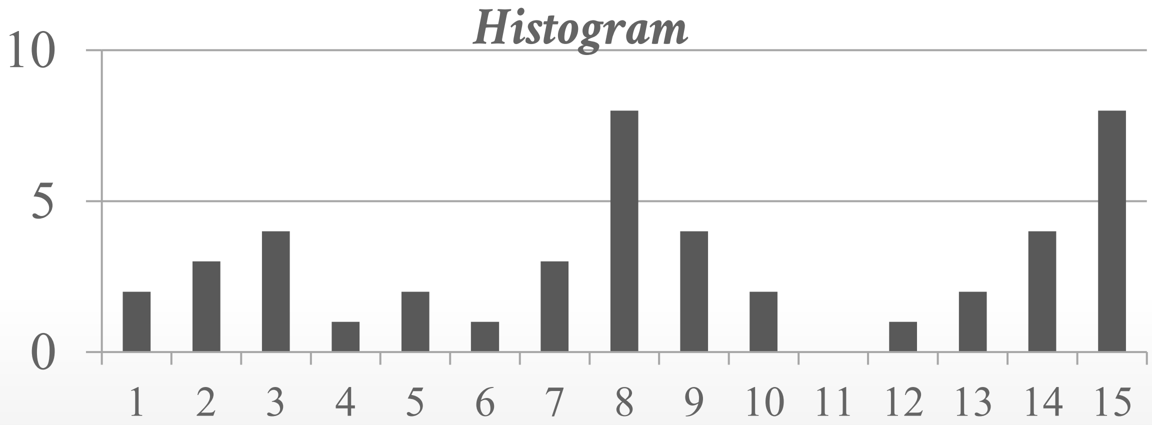
SINGLE SELECTION CONDITION

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) = #occurrences / |R|*



SINGLE SELECTION CONDITION

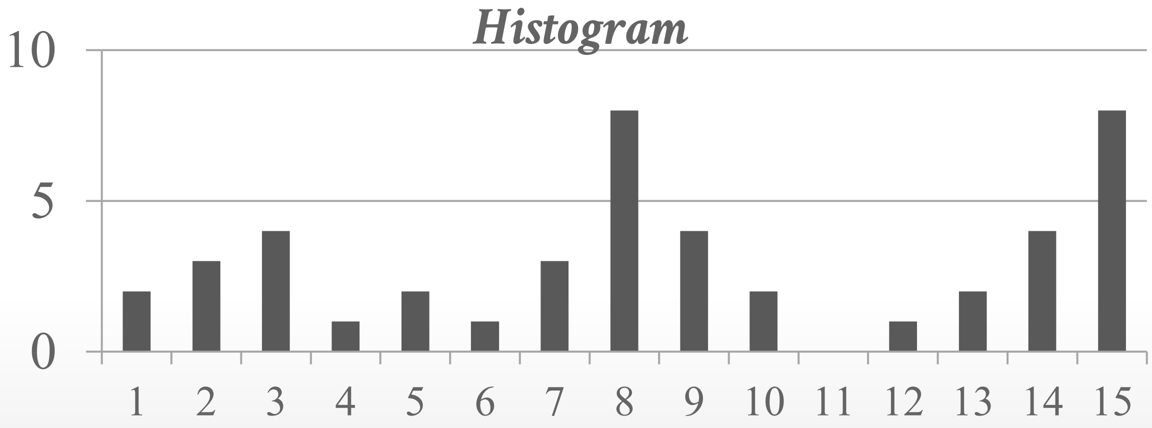
The **selectivity** (*sel*) of a predicate ***P*** is the fraction of tuples that qualify.

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

Equality Predicate: *A=constant*

→ *sel(A=constant) = #occurrences / |R|*

→ Example: *sel(age=9)*



SINGLE SELECTION CONDITION

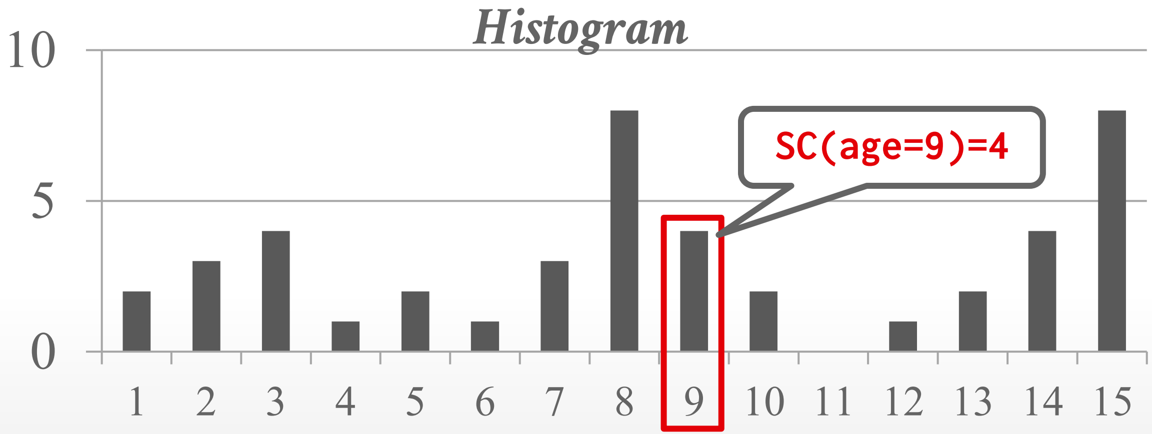
The **selectivity** (*sel*) of a predicate ***P*** is the fraction of tuples that qualify.

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

Equality Predicate: ***A=constant***

→ ***sel(A=constant) = #occurrences / |R|***

→ Example: ***sel(age=9) = 4 / 45 = 0.088***



SINGLE SELECTION CONDITION

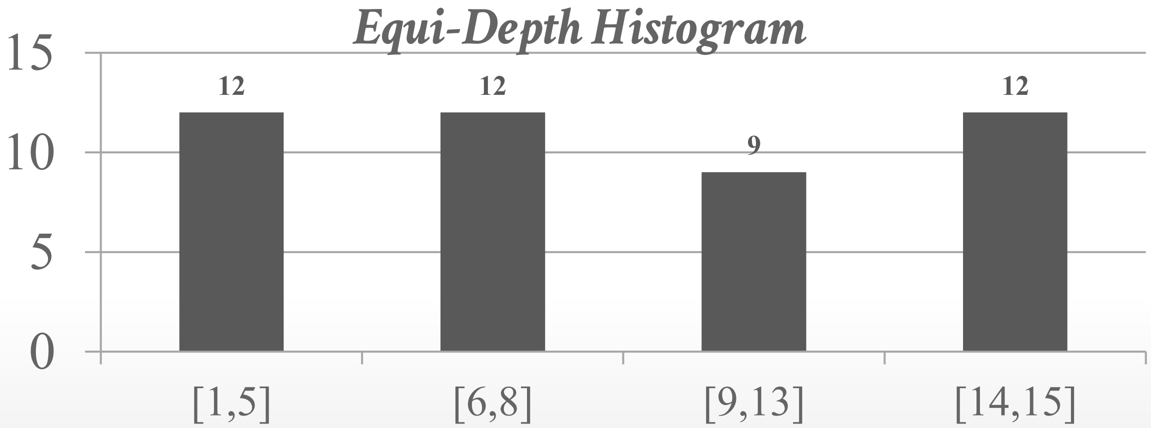
```
SELECT * FROM people
 WHERE age = 9
```

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

Equality Predicate: *A=constant*

→ ***sel(A=constant) = #occurrences / |R|***

→ Example: ***sel(age=9) = 4/45 = 0.088***



SINGLE SELECTION CONDITION

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

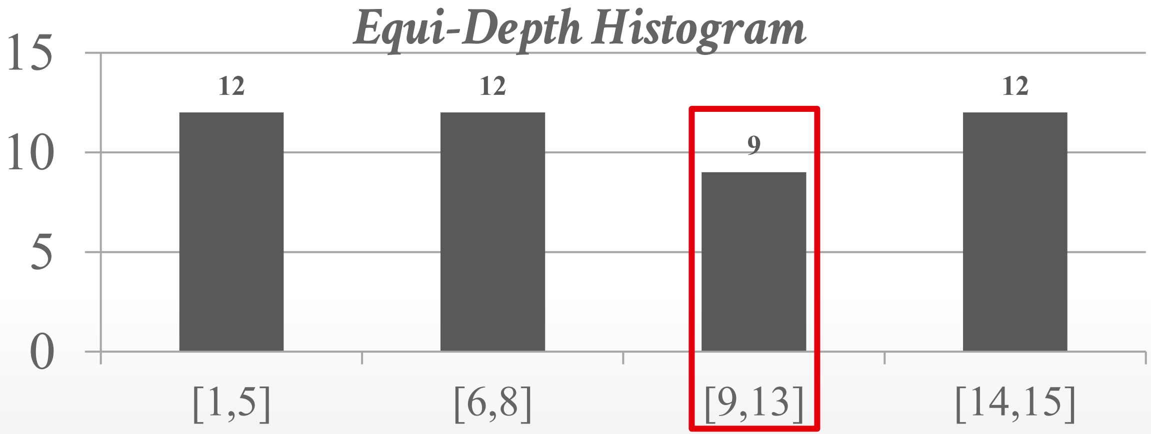
The **selectivity** (*sel*) of a predicate ***P*** is the fraction of tuples that qualify.

Equality Predicate: ***A=constant***

→ ***sel(A=constant) = #occurrences / |R|***

→ Example: ***sel(age=9) = 4/45 = 0.088***

$$\approx (9/5) / 45 \approx 1.8 / 45 \approx 0.04$$



ASSUMPTIONS

Assumption #1: Uniform Data

→ The distribution of values (except for the heavy hitters) is the same within a histogram bucket.

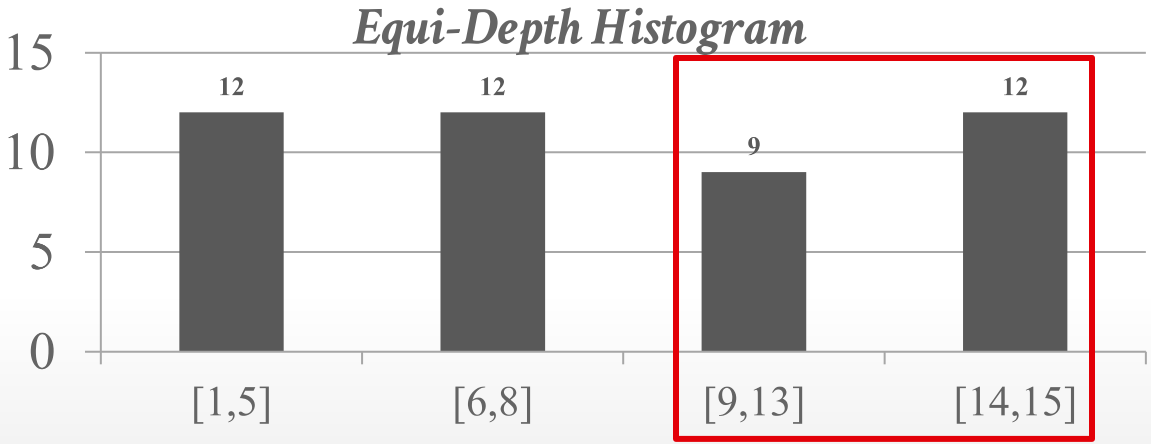
SINGLE SELECTION CONDITION

Range Predicate:

→ *sel(A >= a) = (#RANGE-ROWS + #EQ-ROWS) / |R|*

→ Example: *sel(age >= 7) ≈ ((9+12)*

```
SELECT * FROM people
 WHERE age >= 7
```



SINGLE SELECTION CONDITION

```
SELECT * FROM people
 WHERE age >= 7
```

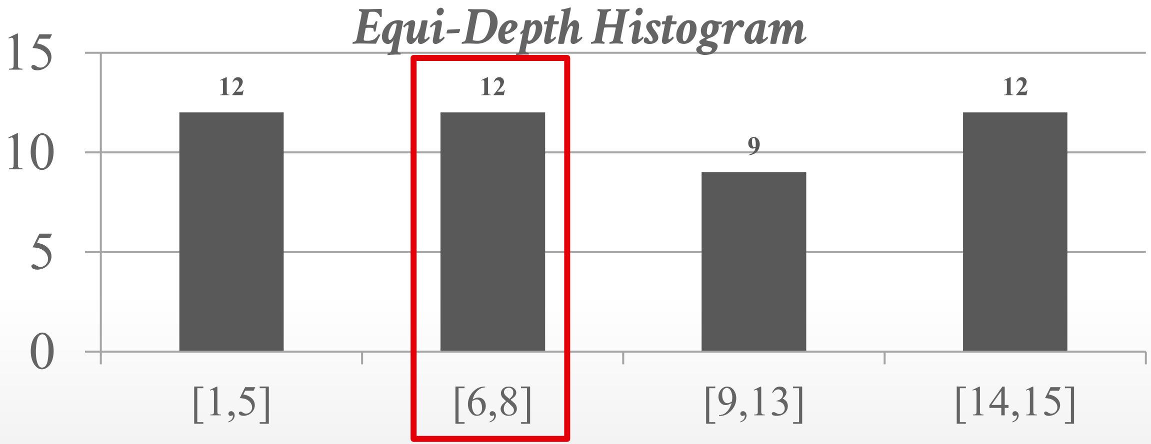
Range Predicate:

→ $sel(A >= a) = (\#RANGE\text{-}ROWS + \#EQ\text{-}ROWS) / |R|$

→ Example: $sel(age >= 7) \approx ((9+12) + (2 \times (12/3))) / 45$

$$\approx 29 / 45 \approx 0.6444$$

This assumes continuous distribution of values.



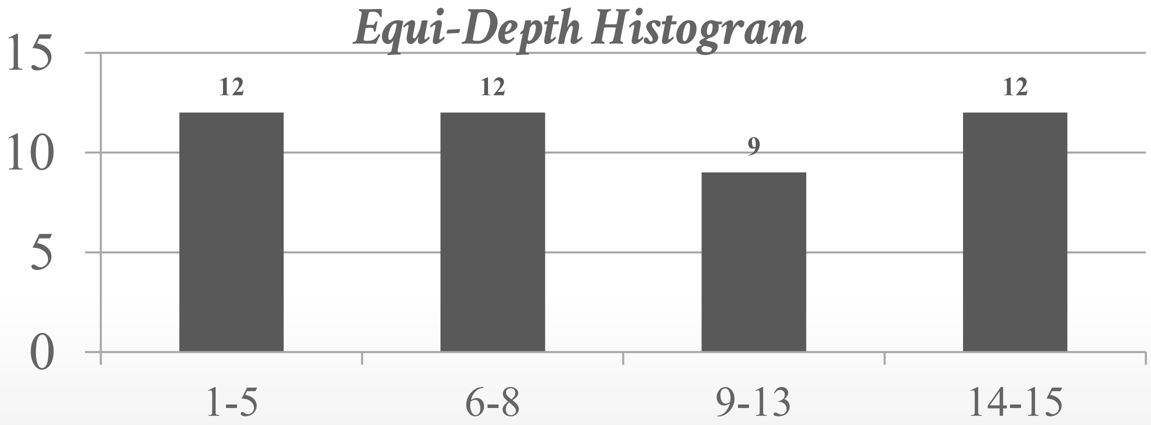
SINGLE SELECTION CONDITION

```
SELECT * FROM people
 WHERE age != 2
```

Negation Query:

→ ***sel(not P) = 1 – sel(P)***

→ Example: ***sel(age != 2)***



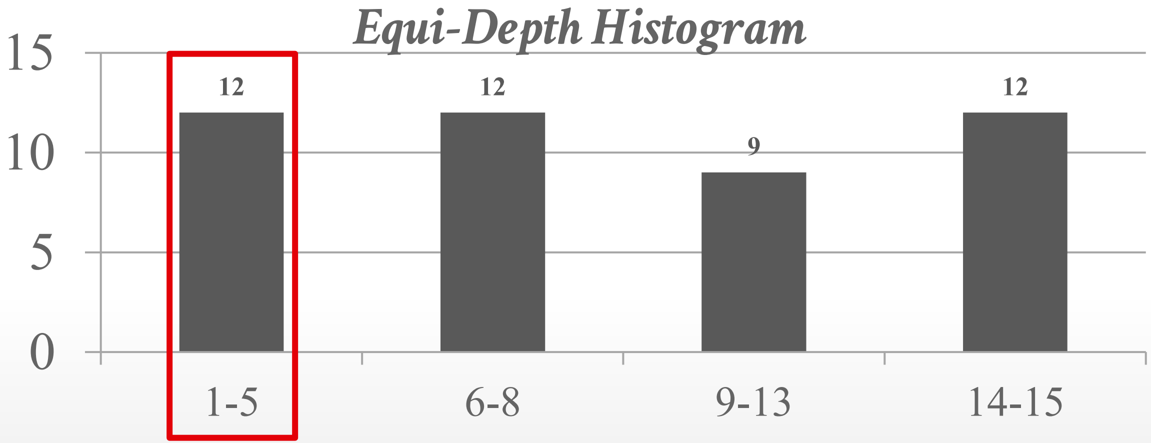
SINGLE SELECTION CONDITION

Negation Query:

→ ***sel(not P) = 1 – sel(P)***

→ Example: ***sel(age != 2)***

```
SELECT * FROM people
 WHERE age != 2
```



SINGLE SELECTION CONDITION

```
SELECT * FROM people
 WHERE age != 2
```

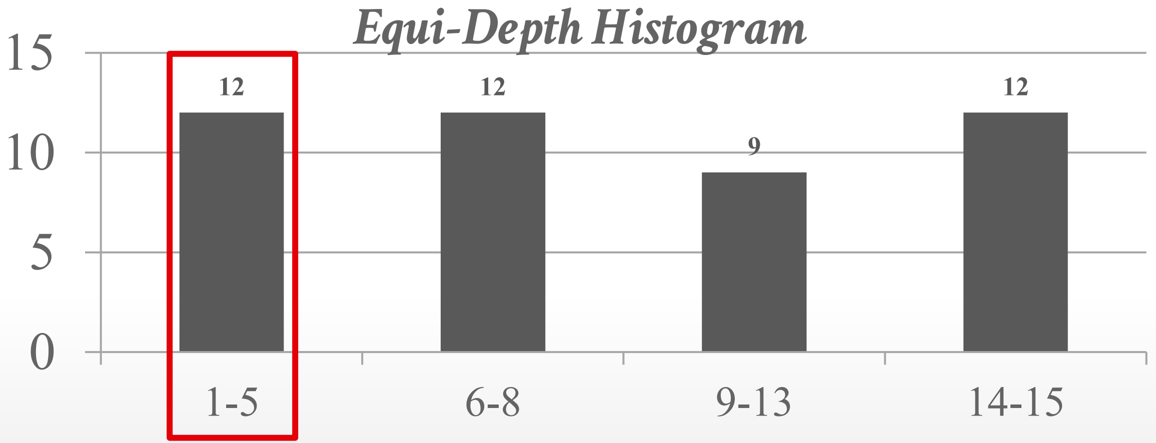
Negation Query:

→ $sel(\text{not } P) = 1 - sel(P)$

→ Example: $sel(age \mathrel{!=} 2) \approx 1 - ((12/5)/45)$

$\approx 1 - (2.4/45) \approx 1 - 0.05 \approx 0.95$

Observation: Selectivity ≈ Probability



OBSERVATION

We can compute selectivities for individual predicates, but what happens if there are multiple predicates in a query?

→ Even though the predicates are on the same table, the attributes may have different distributions.

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```

Example:

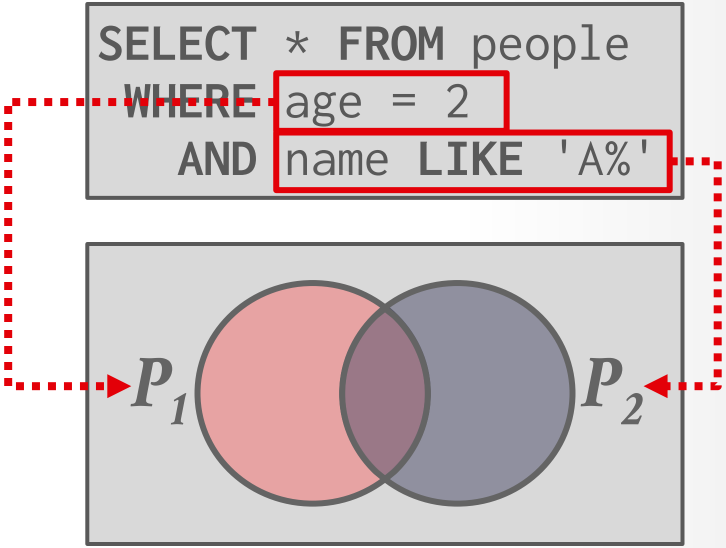
→ $sel(age = 2) \approx 0.053$

→ $sel(name\ LIKE\ 'A\%') \approx 0.1$

OBSERVATION

We can compute selectivities for individual predicates, but what happens if there are multiple predicates in a query?

→ Even though the predicates are on the same table, the attributes may have different distributions.



Example:

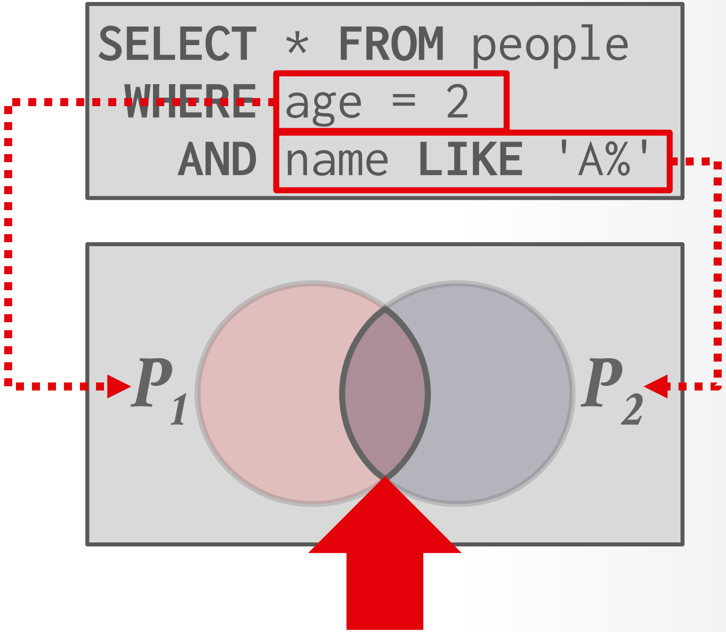
→ $sel(age = 2) \approx 0.053$

→ $sel(name\ LIKE\ 'A\%') \approx 0.1$

OBSERVATION

We can compute selectivities for individual predicates, but what happens if there are multiple predicates in a query?

→ Even though the predicates are on the same table, the attributes may have different distributions.



Example:

→ $sel(age = 2) \approx 0.053$

→ $sel(name\ LIKE\ 'A\%') \approx 0.1$

ASSUMPTIONS

Assumption #1: Uniform Data

→ The distribution of values (except for the heavy hitters) is the same within a histogram bucket.

Assumption #2: Independent Predicates

→ The selectivity of the conjunction of two or more predicates is estimated as the product of their individual selectivities.

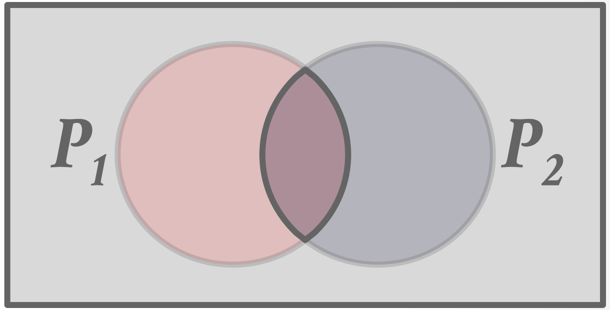
MULTIPLE SELECTION CONDITION

Conjunction:

→ $sel(P1 \land P2) = sel(P1) \times sel(P2)$

→ Example: $sel(age{=}2 \land name\ LIKE\ 'A\%')$
$\approx sel(age{=}2) \times sel(name\ LIKE\ 'A\%')$
$\approx 0.053 \times 0.1 \approx 0.0053$

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```



This assumes that the predicates are **independent**.

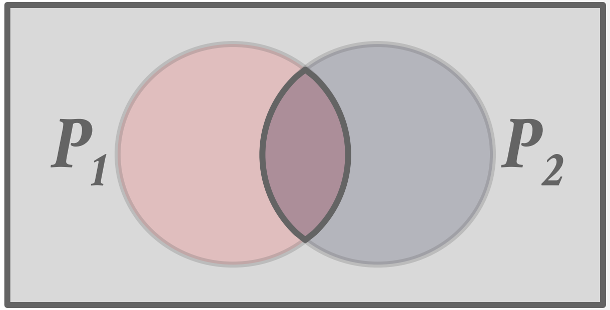
MULTIPLE SELECTION CONDITION

```sql
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```

Conjunction:

→ $sel(P1 \land P2) = sel(P1) \times sel(P2)$

→ Example: $sel(age{=}2 \land name\ LIKE\ 'A\%')$
$\approx sel(age{=}2) \times sel(name\ LIKE\ 'A\%')$
$\approx 0.053 \times 0.1 \approx 0.0053$



This assumes that the predicates are <u>**independent**</u>.

Optimization: When there are multiple predicates, diminish their weights to reduce underestimations.

MULTIPLE SELECTION CONDITION

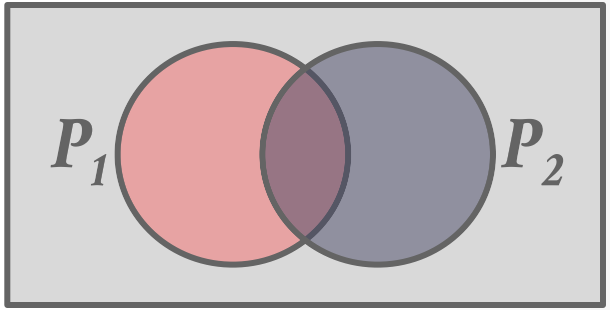
Disjunction:

→ $sel(P1 \vee P2)$

$\approx sel(P1) + sel(P2) - sel(P1 \wedge P2)$

$\approx sel(P1) + sel(P2) - sel(P1) \times sel(P2)$

```
SELECT * FROM people
 WHERE age = 2
    OR name LIKE 'A%'
```



MULTIPLE SELECTION CONDITION

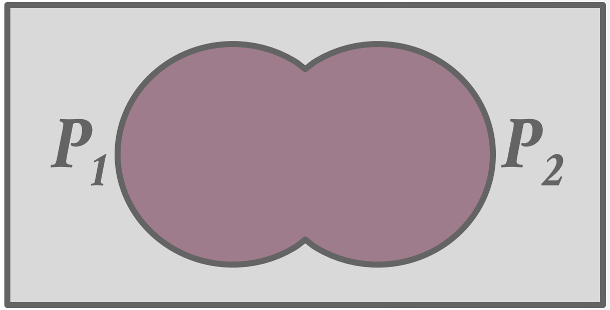
Disjunction:

→ $sel(P1 \vee P2)$

$\approx sel(P1) + sel(P2) - sel(P1 \wedge P2)$

$\approx sel(P1) + sel(P2) - sel(P1) \times sel(P2)$

```
SELECT * FROM people
 WHERE age = 2
    OR name LIKE 'A%'
```



MULTIPLE SELECTION CONDITION

Disjunction:

→ $sel(P1 \lor P2)$

$\approx sel(P1) + sel(P2) - sel(P1 \land P2)$

$\approx sel(P1) + sel(P2) - sel(P1) \times sel(P2)$

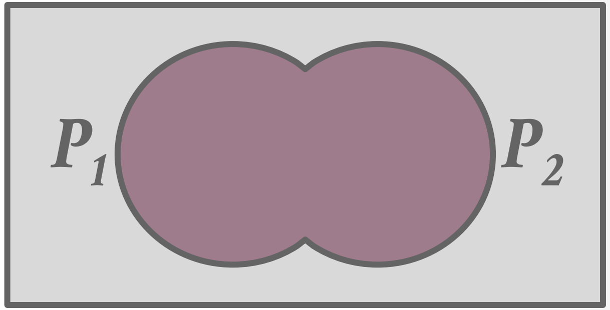
→ Example: $sel(age{=}2 \lor name\ LIKE\ 'A\%')$

$\approx 0.053 + 0.1 - (0.053 \times 0.1)$

≈ 0.1477

This again assumes that the selectivities are **independent**.

```
SELECT * FROM people
 WHERE age = 2
    OR name LIKE 'A%'
```



CORRELATED ATTRIBUTES

Consider a database of automobiles:
→ # of Makes = 10, # of Models = 100

Then the following query shows up:
→ `WHERE (make='Honda' AND model='Accord')`

With the independence and uniformity assumptions, the selectivity is:
→ $1/10 \times 1/100 \approx 0.001$

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

Source: Guy Lohman

JOIN SIZE ESTIMATION

Given a join of ***R*** and ***S***, what is the range of possible result sizes in # of tuples?

→ In other words, for a given tuple of ***R***, how many tuples of ***S*** will it match?

Assume each key in the inner relation will exist in the outer table.

ASSUMPTIONS

Assumption #1: Uniform Data

→ The distribution of values (except for the heavy hitters) is the same within a histogram bucket.

Assumption #2: Independent Predicates

→ The selectivity of the conjunction of two or more predicates is estimated as the product of their individual selectivities.

Assumption #3: Containment Principle

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

JOIN SIZE ESTIMATION

General case: $R_{cols} \cap S_{cols} = \{A\}$ where A is not a primary key for either table.

→ Match each ***R***-tuple with ***S***-tuples:
  $estSize \approx N_R \times N_S / V(A,S)$

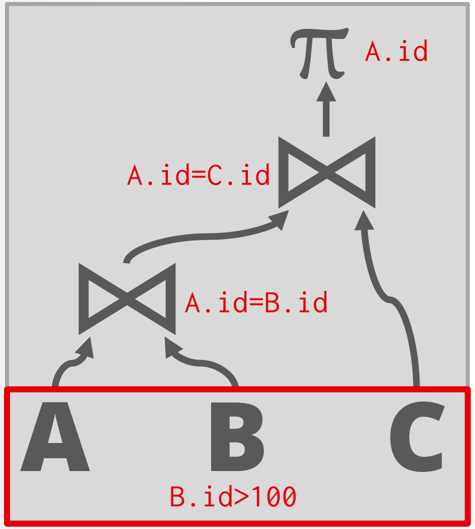
→ Symmetrically, for ***S***:
  $estSize \approx N_R \times N_S / V(A,R)$

The cardinality estimate of a join is:

→ $estSize \approx N_R \times N_S / max(\{V(A,S), V(A,R)\})$

ERROR PROPAGATION

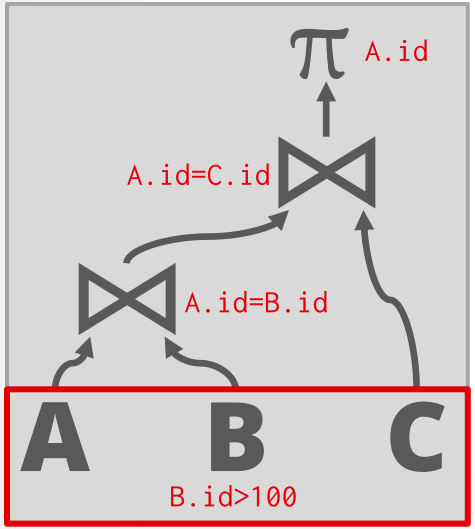
```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

ERROR PROPAGATION

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

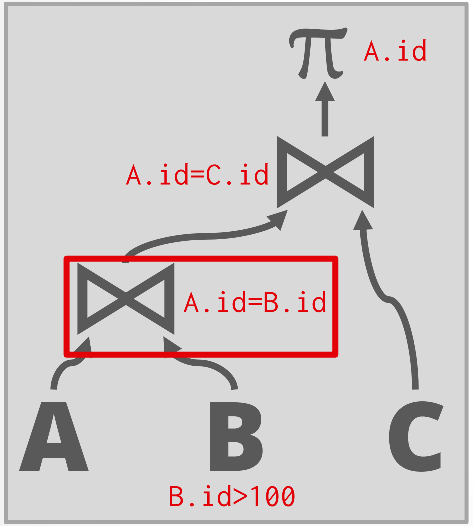
$$A \rightarrow |A|$$

$$B.id > 100 \rightarrow |B| \times sel(B.id > 100)$$

$$C \rightarrow |C|$$

ERROR PROPAGATION

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

$A \rightarrow |A|$

$B.id > 100 \rightarrow |B| \times sel(B.id > 100)$

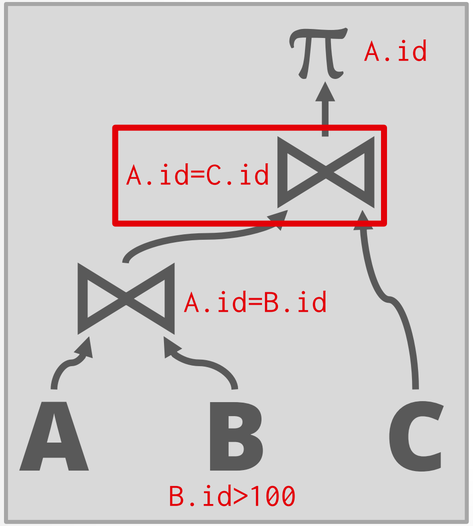
$C \rightarrow |C|$

Compute the cardinality of join results

$$A \bowtie B \approx (|A| \times |B|) \,/\, \max(sel(A.id{=}B.id),\ sel(B.id{>}100))$$

ERROR PROPAGATION

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

$$A \rightarrow |A|$$

$$B.id > 100 \rightarrow |B| \times \mathbf{sel}(B.id > 100)$$

$$C \rightarrow |C|$$

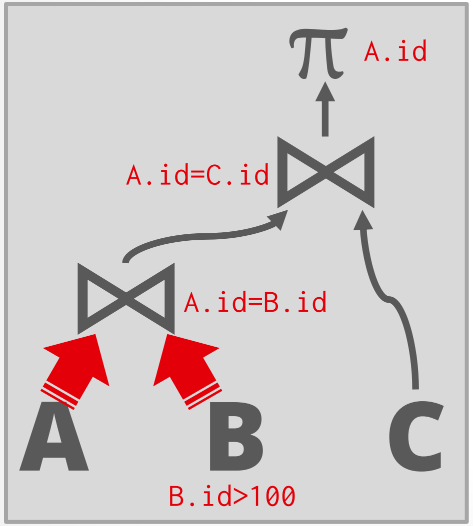
Compute the cardinality of join results

$$A \bowtie B \approx (|A| \times |B|) \,/\, \mathbf{max}(\mathbf{sel}(A.id{=}B.id),\ \mathbf{sel}(B.id{>}100))$$

$$(A \bowtie B) \bowtie C \approx (|A| \times |B| \times |C|) \,/\, \mathbf{max}(\mathbf{sel}(A.id{=}B.id),\ \mathbf{sel}(B.id{>}100),\ \mathbf{sel}(A.id{=}C.id))$$

ERROR PROPAGATION

```sql
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

$A \rightarrow |A|$

$B.id > 100 \rightarrow |B| \times sel(B.id > 100)$

$C \rightarrow |C|$

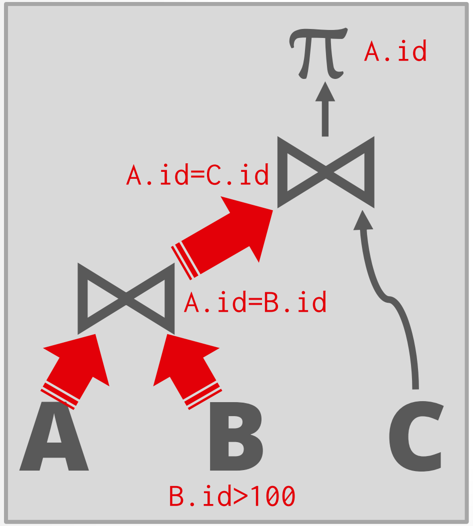
Compute the cardinality of join results

$A \bowtie B \approx (|A| \times |B|) \,/\, max(sel(A.id{=}B.id),\ sel(B.id{>}100))$

$(A \bowtie B) \bowtie C \approx (|A| \times |B| \times |C|) \,/\, max(sel(A.id{=}B.id),\ sel(B.id{>}100),\ sel(A.id{=}C.id))$

ERROR PROPAGATION

```
SELECT A.id
  FROM A, B, C
 WHERE A.id = B.id
   AND A.id = C.id
   AND B.id > 100
```



Compute the cardinality of base tables

$A \rightarrow |A|$

$B.id > 100 \rightarrow |B| \times sel(B.id > 100)$

$C \rightarrow |C|$

Compute the cardinality of join results

$A \bowtie B \approx (|A| \times |B|) \,/\, max(sel(A.id{=}B.id),\ sel(B.id{>}100))$

$(A \bowtie B) \bowtie C \approx (|A| \times |B| \times |C|) \,/\, max(sel(A.id{=}B.id),\ sel(B.id{>}100),\ sel(A.id{=}C.id))$

CONCLUSION

Statistics allow the optimizer to summarize the contents of the database.

→ These data structures are only approximations of real data.

Then the optimizer guesses how many tuples it will examine or emit at each operator in a query plan.

→ Another approximation of what a real predicate will do.

This entire process is fraught with errors.

CONCLUSION

Statistics allow
of the database
→ These data str

Then the opti
examine or em
→ Another appr

This entire pr

NEXT CLASS

Transactions!

→ aka the second hardest part about database systems