CS460

Systems for Data Management and Data Science

Query Optimization

Prof. Anastasia Ailamaki

Data-Intensive Applications and Systems (DIAS) Lab

"Man plans, and God laughs"
– Yiddish proverb

Some slides adapted from Andy Pavlo

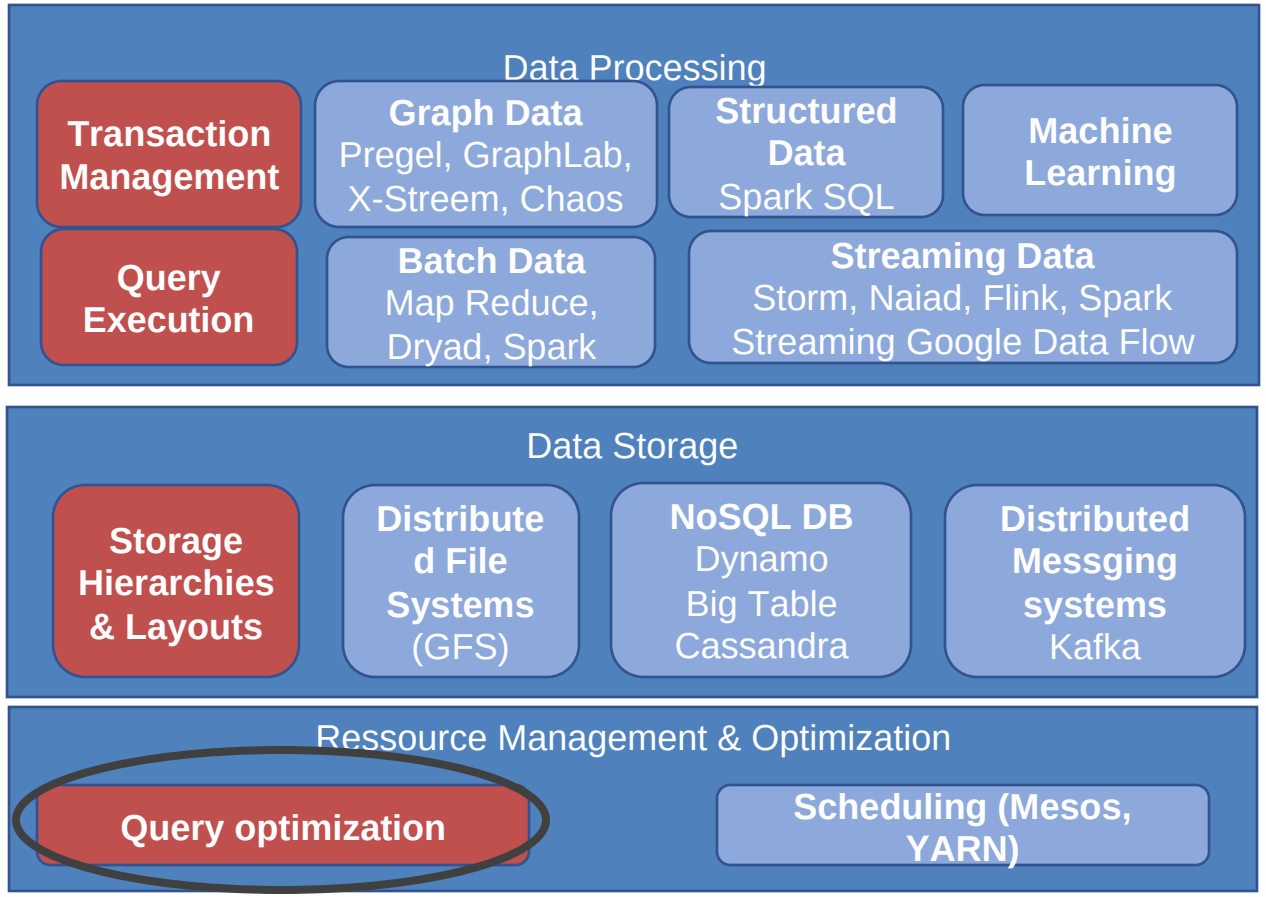
Today's topic

Gossip Protocols

Consistency protocols
CAP Theorem

Distributed/Decentralized systems

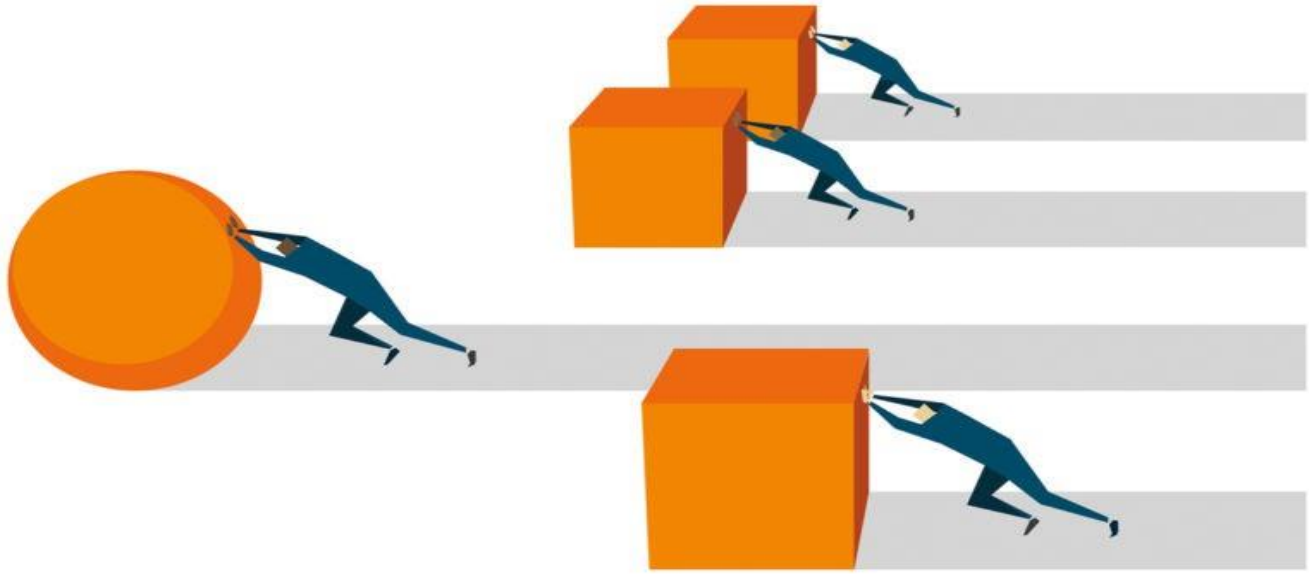
Data science software stack



Today's overview

Query Optimization

How to improve performance
by *wisely* choosing the order and implementation of the operators?



Outline

Introduction to query optimization

Relational algebra equivalences

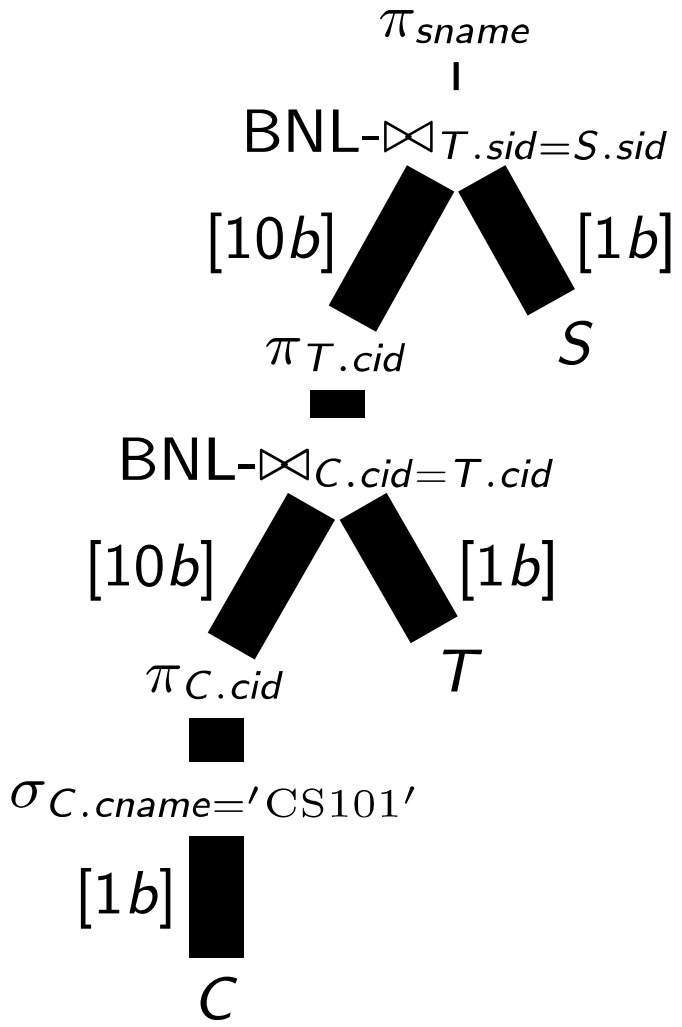
Optimizers based on heuristics – INGRES

Optimizers based on heuristics & cost - SYSTEM R

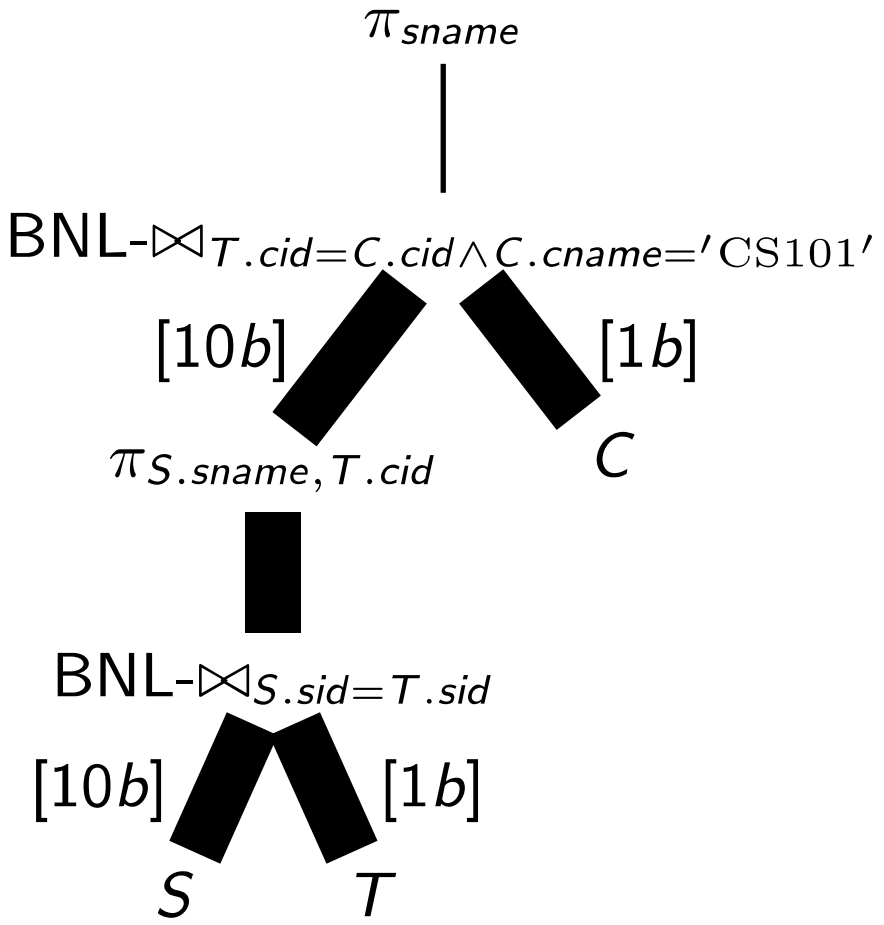
- Cost & selectivity estimation
- Principle of optimality
- System R

Query Optimization

For a given query, find the execution plan with the lowest "cost".




```
SELECT S.sname
FROM S, T, C
WHERE S.sid=T.sid
  AND T.cid=C.cid
  AND C.cname='CS101'
```



Query Optimization

- The part of a DBMS that is the hardest to implement correctly.
  - This is (mostly) what you pay for!
- No optimizer truly produces the “optimal” plan
  - Too expensive to consider all plans (NP-complete)!
  - Impossible to get the accurate cost of a plan without executing it!
- Optimizers make a huge difference in terms of
  - Performance
  - Scalability
  - Database capabilities

Decisions, decisions…

- In what order to execute operations?
  - Particularly: relative order of joins
- Which implementation is best for each operation?
  - E.g., hash joins, nested loop joins, sort-merge joins…
- Which access methods to use?
  - E.g., scan, use of an index

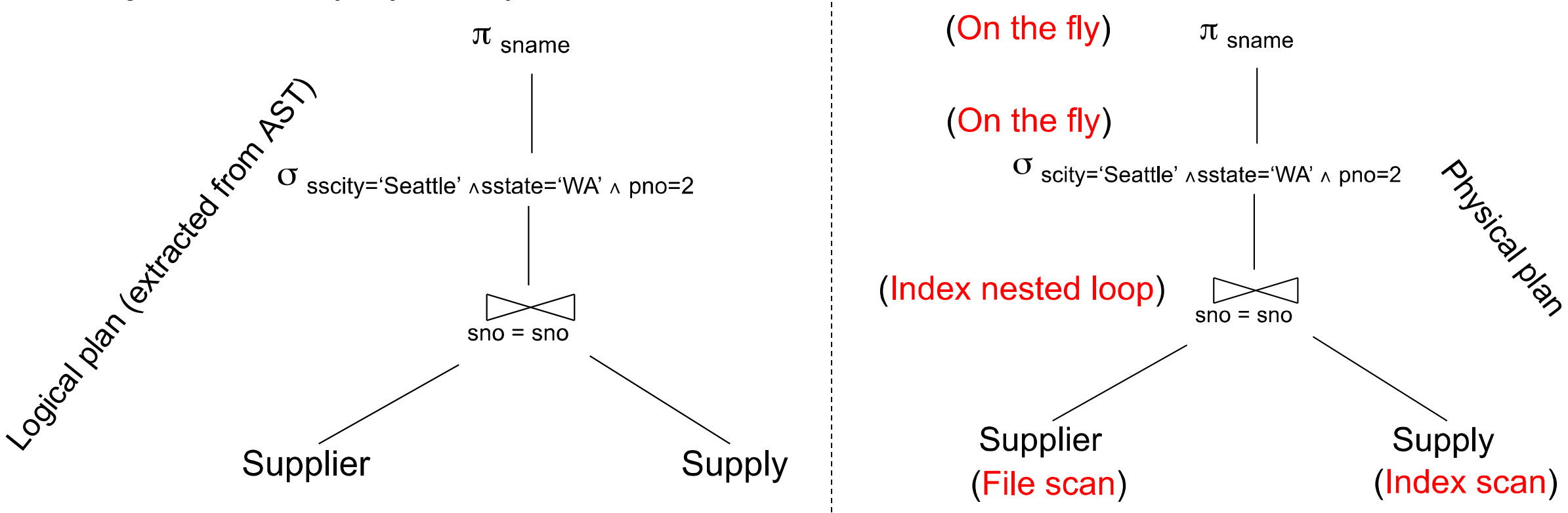
- Suboptimal decisions can have a huge impact! E.g.
  - Use of one join algorithm vs another
  - Pushing down selections (that make indexes useless)

Input/output of query optimizer

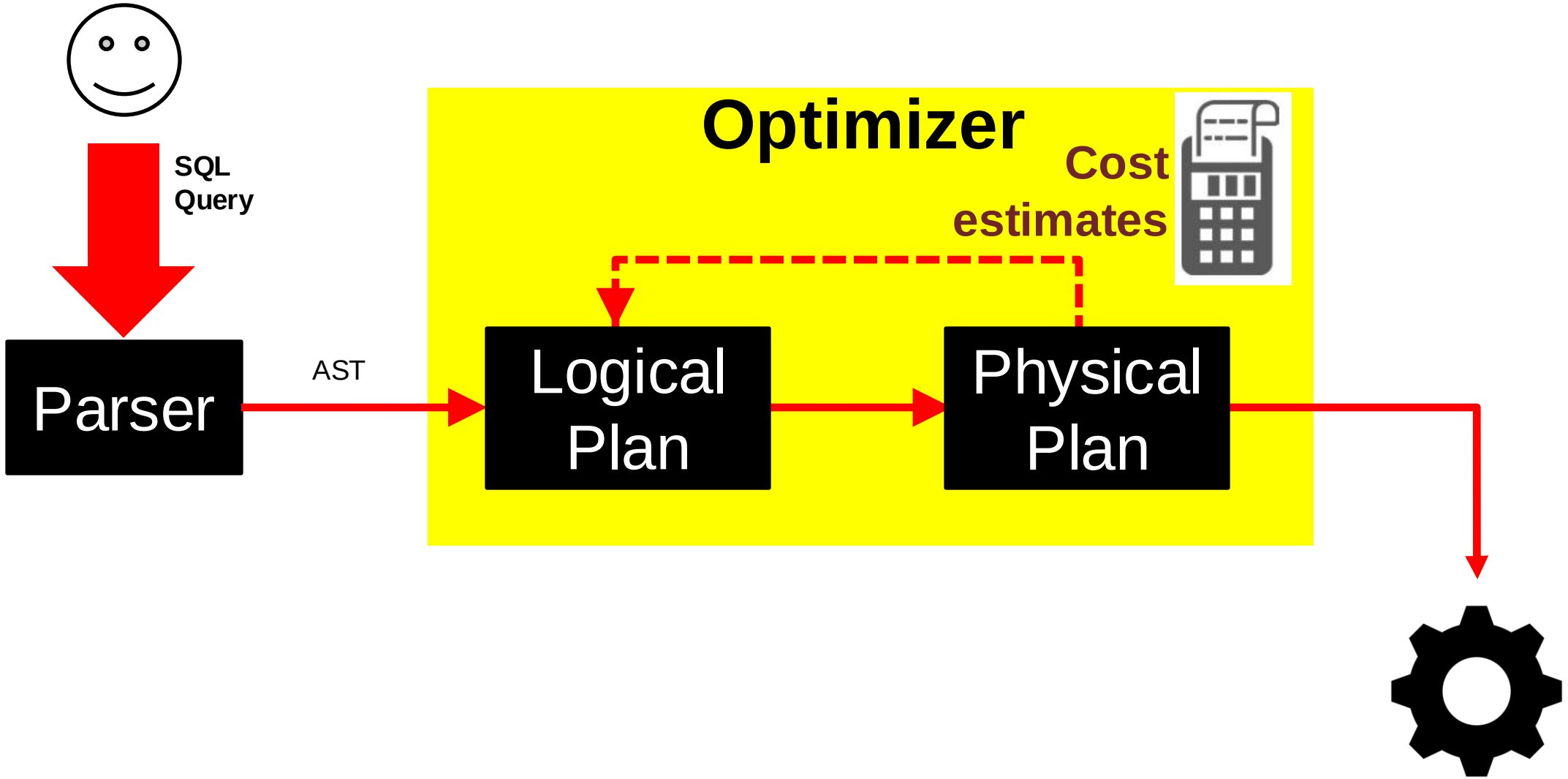
Input: An Abstract Syntax Tree (AST) representing an SQL query

```
SELECT S.sname FROM Supplier S, Supply U
WHERE S.scity='Seattle' AND S.sstate='WA'
AND S.sno = U.sno AND U.pno = 2
```

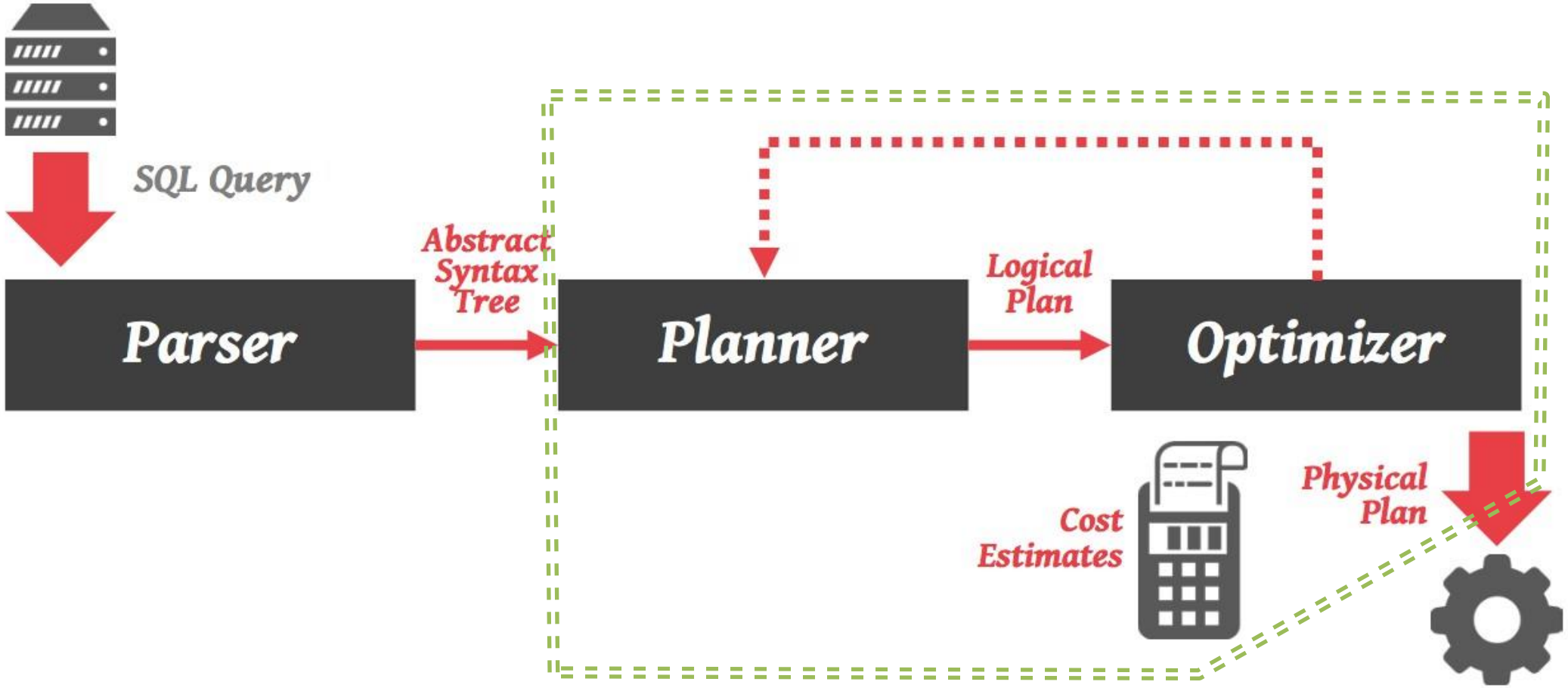
Output: A full physical plan which is *translatable to code*



Classic architecture



Classic architecture – Task-oriented



Outline

Introduction to query optimization

Relational algebra equivalences

Optimizers based on heuristics – INGRES

Optimizers based on heuristics & cost - SYSTEM R

- Cost & selectivity estimation
- Principle of optimality
- System R

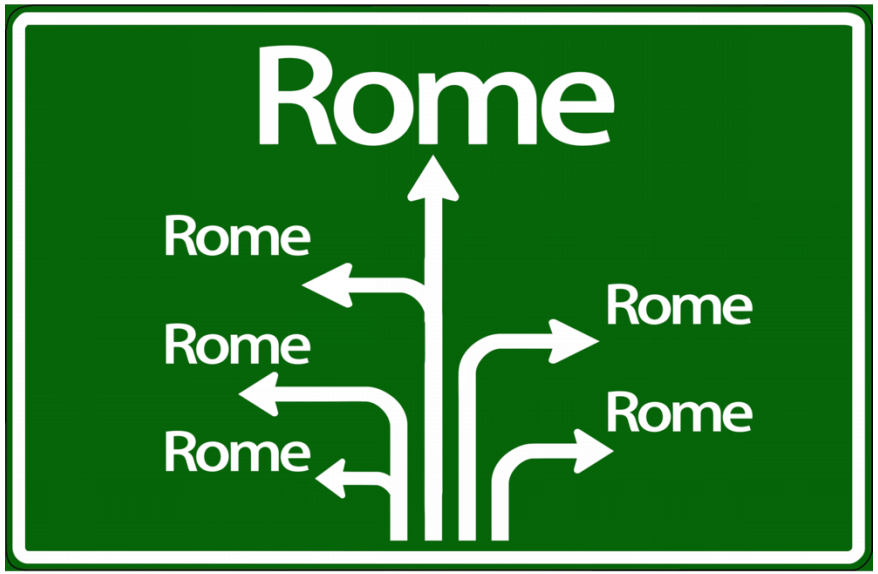
Multi-query optimization

Relational Algebra Equivalences

Key concept in optimization: ***Equivalences***

Two relational algebra expressions are said to be **equivalent** if on every legal database instance, the two expressions generate the same set of tuples.

All roads lead to Rome.
But some of them are more efficient!



Relational Algebra Equivalences

- *Selections*: $\sigma_{c_1 \wedge \cdots \wedge c_n}(R) \equiv \sigma_{c_1}\left(\ldots\left(\sigma_{c_n}(R)\right)\right)$ *(Cascade)*

  $\sigma_{c_1}\left(\sigma_{c_2}(R)\right) \equiv \sigma_{c_2}\left(\sigma_{c_1}(R)\right)$ *(Commute)*

- *Projections:* $\pi_{a_1}(R) \equiv \pi_{a_1}\left(\ldots\left(\pi_{a_n}(R)\right)\right)$ *(Cascade)*

  a_i is a set of attributes of R and $a_i \subseteq a_{i+1}$ for $i = 1 \ldots n-1$

- These equivalences allow us to 'push' selections and projections ahead of joins.

Examples …

$\sigma_{age=18\ \&\ rating>5}$ (Sailors)

$\leftrightarrow$ $\sigma_{age=18}$ ($\sigma_{rating>5}$ (Sailors))

$\leftrightarrow$ $\sigma_{rating>5}$ ($\sigma_{age=18}$ (Sailors))

$\pi_{age,rating}$ (Sailors) $\leftrightarrow$ π_{age} (π_{rating} (Sailors)) (??)

$\pi_{age,rating}$ (Sailors) $\leftrightarrow$ $\pi_{age,rating}$ ($\pi_{age,rating,sid}$ (Sailors))

Another Equivalence

A projection commutes with a selection that only uses attributes retained by the projection

$$\pi_{\text{age, rating, sid}}\,(\sigma_{\text{age<18 \& rating>5}}\,(\text{Sailors})) \leftrightarrow \sigma_{\text{age<18 \& rating>5}}\,(\pi_{\text{age, rating, sid}}\,(\text{Sailors}))$$

Equivalences Involving Joins

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad \textit{(Associative)}$$

$$(R \bowtie S) \equiv (S \bowtie R) \quad \textit{(Commutative)}$$

These equivalences allow us to choose different join orders.

Mixing Joins with Selections & Projections

Converting selection + cross-product to join

$$\sigma_{S.sid = R.sid}\,(\text{Sailors} \times \text{Reserves})$$

$$\leftrightarrow \text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves}$$

Selection on just attributes of S commutes with R $\bowtie$ S

$$\sigma_{S.age<18}\,(\text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves})$$

$$\leftrightarrow (\sigma_{S.age<18}\,(\text{Sailors})) \bowtie_{S.sid = R.sid} \text{Reserves}$$

We can also "push down" projections

CAREFUL!
Not always wise

$$\pi_{S.sname}\,(\text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves})$$

$$\leftrightarrow \pi_{S.sname}\,(\pi_{sname,sid}(\text{Sailors}) \bowtie_{S.sid = R.sid} \pi_{sid}(\text{Reserves}))$$

Example – naïve approach

S: 16000 tuples = 320 pages
T: 256000 tuples = 5120 pages
C: 1600 tuples = 32 pages
25% of courses being taken (T) are ‘cs101’

```
SELECT S.sname
FROM   S, T, C
WHERE  S.sid=T.sid
  AND  T.cid=C.cid
  AND  C.cname='CS101'
```

Tuple-by-tuple
takes > 20.8 years

Page-by-page
takes 1.5 hours

Using cross products

```
for each tuple c of C on disk do
 for each tuple s of S on disk do
  for each tuple t of T on disk do
   if the condition on (s, t, c) holds
    output s.sname;
```

Explanation

Assume relations are stored on an SSD

Each I/O to fetch a page is 0.1ms

→ **Super-Worst scenario / tuple-by-tuple**

Cartesian product of fetching a page for each tuple (1 I/O per tuple):

#tuples(C) * #tuples(S) * #tuples(T) = 1,600 * 16,000 * 256,000 = 6,553,600,000,000 I/Os. At 0.1ms per I/O → query takes 20.8 years

→ **Not-Worst-But-Very-Bad scenario (page by page):**

Cartesian product reading pages at a time, not tuples (1 I/O per page)

#pages(C) * #pages(S) * #pages(T) = 32 * 320 * 5120 = 52,428,800 I/Os

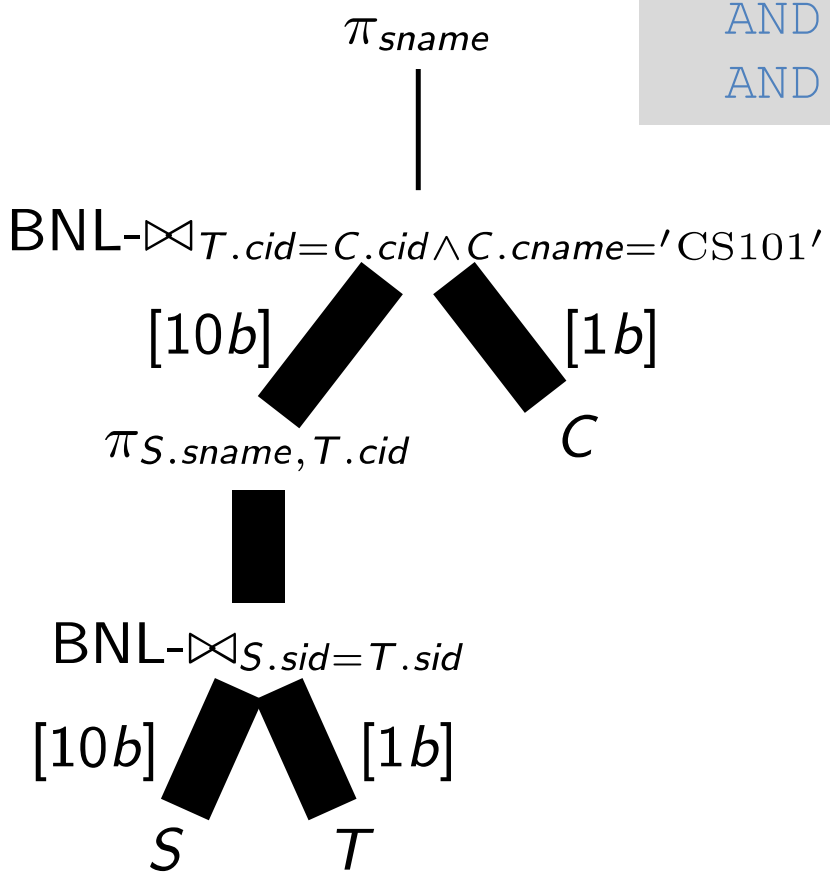
52,428,800 * 0.1ms = 5,242 s → query takes 1.5 hours

Example – educated approach

S: 16000 tuples = 320 pages
T: 256000 tuples = 5120 pages
C: 1600 tuples = 32 pages

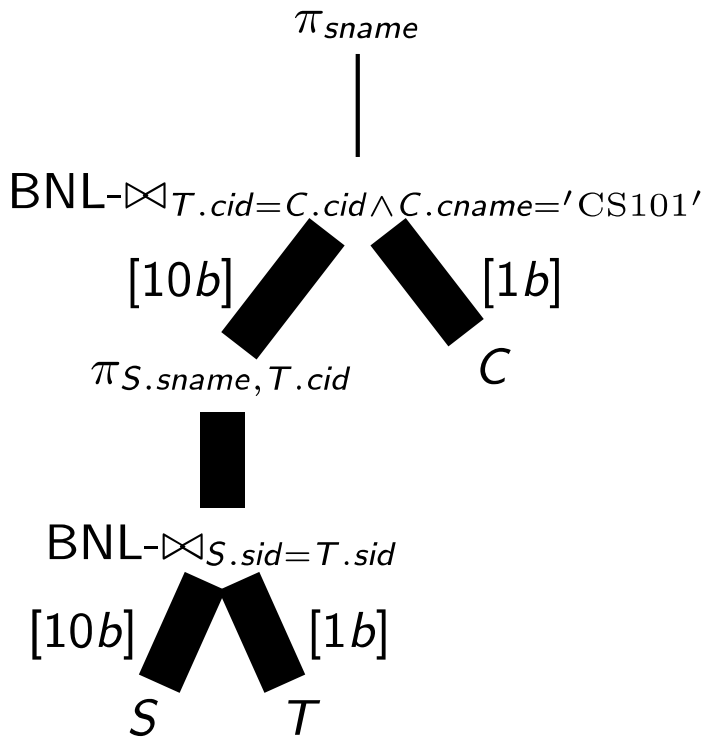
Use joins instead of cross product & push down projection
17 sec

```
SELECT S.sname
FROM   S, T, C
WHERE  S.sid=T.sid
  AND  T.cid=C.cid
  AND  C.cname='CS101'
```



Example – educated approach (cont'd)

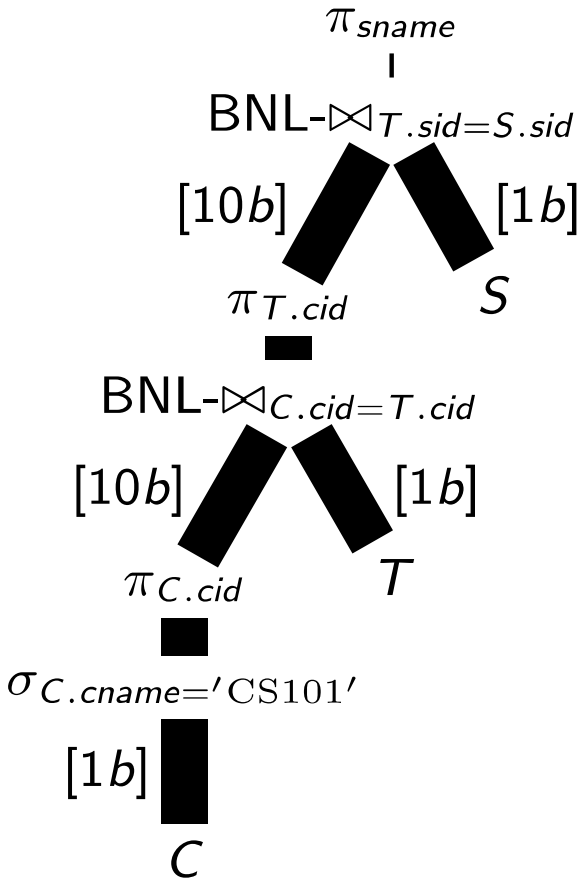
Node	output #tps/pg	output #tps	output #pgs	I/O pgs
S	50	16000	320	320
T	50	256000	5120	0
S⋈T	25	256000	10240	163840
π(ST)	125	256000	2048	0
C	50	1600	32	0
ST ⋈ C	36	64000	1778	6560
π(STC)	250	64000	256	0



Example – super optimized

S: 16000 tuples = 320 pages
T: 256000 tuples = 5120 pages
C: 1600 tuples = 32 pages

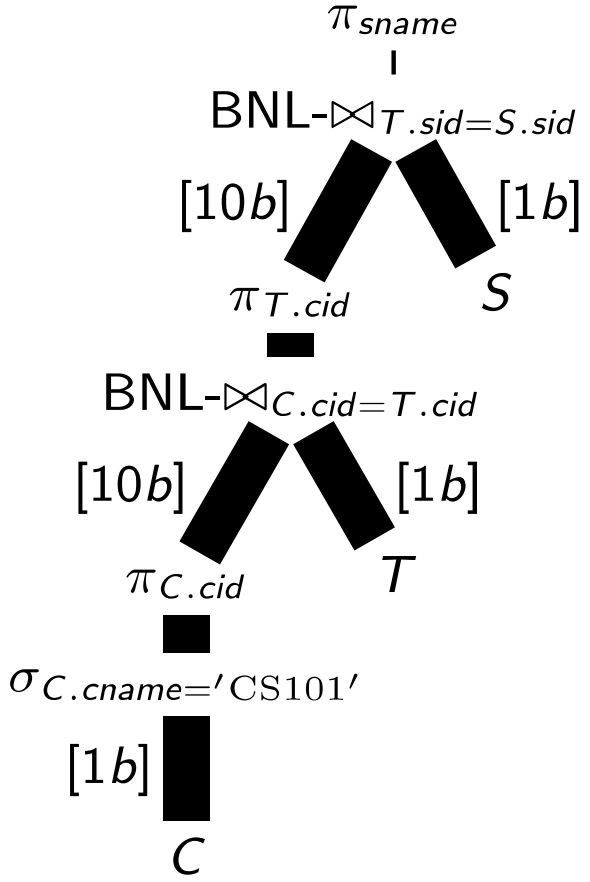
Push down selection and reorder joins
1 sec!




```
SELECT S.sname
FROM   S, T, C
WHERE  S.sid=T.sid
  AND  T.cid=C.cid
  AND  C.cname='CS101'
```

An example – super optimized (cont'd)

Node	output #tps/pg	output #tps	output #pgs	I/O pgs
C	50	1600	32	32
π(σ(C))	250	1	1	0
T	50	256000	5120	0
C⋈T	42	64000	1524	$\lceil 1/10 \rceil$*5120
π(CT)	250	64000	256	0
S	50	16000	320	0
CT ⋈ S	42	64000	1524	8320
π(CTS)	250	64000	256	0



Simple queries, straightforward plan

Query planning for OLTP queries is easy because they are **sargable**

Search Argument 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

Pat Selinger
IBM Fellow,
Paradata,
Salesforce

We focus on OLAP queries (more interesting)

Outline

Introduction to query optimization

Relational algebra equivalences

Optimizers based on heuristics – INGRES

Optimizers based on heuristics & cost - SYSTEM R

- Cost & selectivity estimation
- Principle of optimality

Heuristic-based optimization

Define static rules that transform logical operators to a physical plan

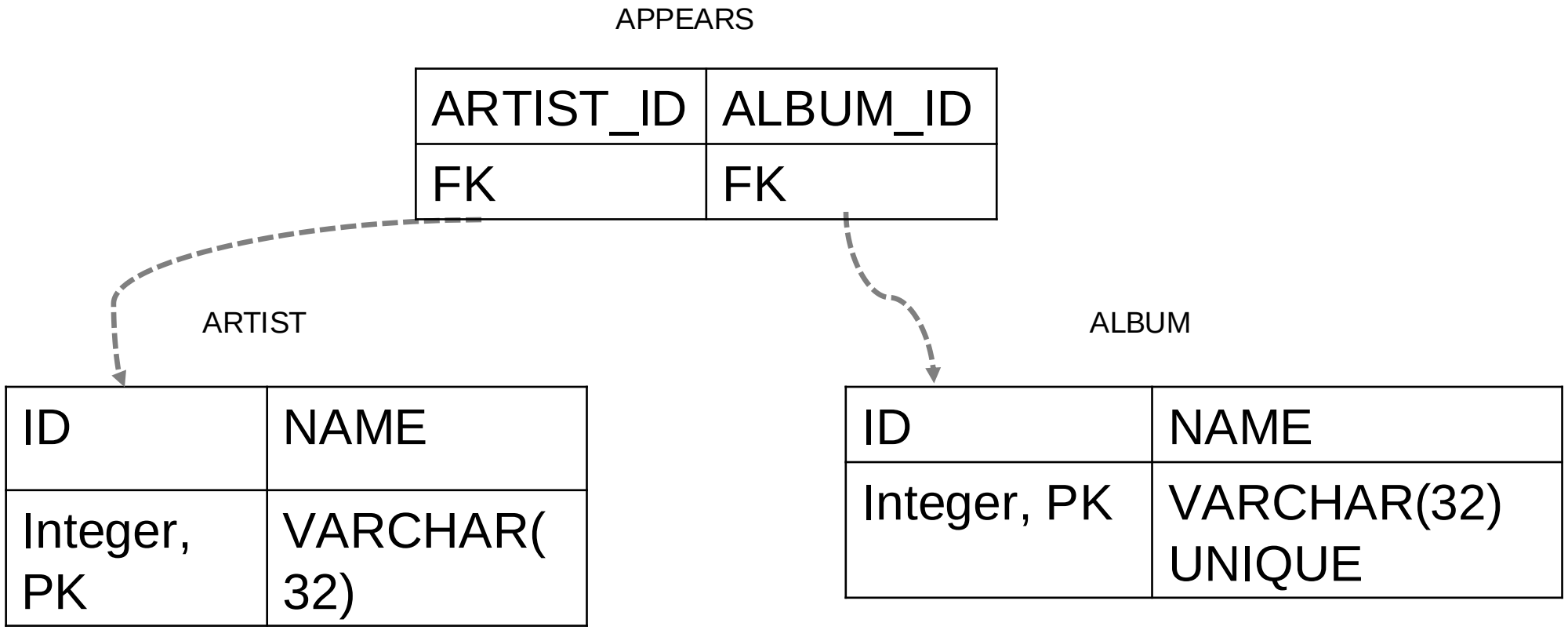
Michael Stonebraker, Turing Award 2014

→ Perform most restrictive selections early
→ Perform all selections before joins
→ Predicate/Limit/Projection pushdowns
→ Join ordering based on cardinality

Example: INGRES and Oracle (until mid 1990s)

Example - INGRES

Developed at UC Berkeley. This ultimately led to Ingres Corp., Sybase, MS SQL Server, Britton-Lee, Wang's PACE.



INGRES optimizer

Retrieve the names of artists that appear on Joy's mixtape

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

APPEARS

ARTIST_ID	ALBUM_ID
FK	FK

ARTIST

ID	NAME
Integer, PK	VARCHAR(32)

ALBUM

ID	NAME
Integer, PK	VARCHAR(32) UNIQUE

INGRES optimizer

Retrieve the names of artists that appear on Joy's mixtape

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

Q1

```
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
FROM ALBUM
WHERE ALBUM.NAME="Joy's Slag Remix"
```

Q2

```
SELECT ARTIST.NAME
FROM   ARTIST, APPEARS, TEMP1
WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
```

Step #1: Decompose into single-variable queries

APPEARS

ARTIST_ID	ALBUM_ID
FK	FK

ARTIST

ID	NAME
Integer, PK	VARCHAR(32)

ALBUM

ID	NAME
Integer, PK	VARCHAR(32) UNIQUE

INGRES optimizer

Retrieve the names of artists that appear on Joy's mixtape

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

Q1

```
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
FROM ALBUM
WHERE ALBUM.NAME="Joy's Slag Remix"
```

Step #1: Decompose into single-variable queries

APPEARS

ARTIST_ID	ALBUM_ID
FK	FK

ARTIST

ID	NAME
Integer, PK	VARCHAR(32)

ALBUM

ID	NAME
Integer, PK	VARCHAR(32) UNIQUE

Q3

```
SELECT APPEARS.ARTIST_ID INTO TEMP2
FROM APPEARS, TEMP1
WHERE APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
```

Q4

```
SELECT ARTIST.NAME
FROM ARTIST, TEMP2
WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

INGRES optimizer

Retrieve the names of artists that appear on Joy's mixtape

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

Q1

ALBUM_ID
9999

Step #1: Decompose into single-variable queries

Q3

ARTIST_ID
123
456

Step #2: Substitute the values from Q1→Q3→Q4

Q4

```
SELECT ARTIST.NAME
FROM ARTIST, ~~TEMP2~~
WHERE ARTIST.ARTIST_ID=~~TEMP2.ARTIST_ID~~
```

123

NAME
George

INGRES optimizer

Retrieve the names of artists that appear on Joy's mixtape

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

Q1

ALBUM_ID
9999

Step #1: Decompose into single-variable queries

Q3

ARTIST_ID
123
456

Step #2: Substitute the values from Q1→Q3→Q4

Q4

```
SELECT ARTIST.NAME
FROM ARTIST, TEMP2                    456
WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

NAME	NAME
George	John

We are judging the optimizer based on today's database complexity. How about 1975?

Advantages:

→ Easy to implement and debug.

→ Works reasonably well and is fast for simple queries & small tables.

Disadvantages:

→ Doesn't *truly* handle joins.

→ Join ordering based only on cardinalities.

→ Naïve, nearly impossible to generate good plans when operators have complex interdependencies.

Outline

Introduction to query optimization

Relational algebra equivalences

Optimizers based on heuristics – INGRES

Optimizers based on heuristics & cost

- **Cost & selectivity estimation**
- Principle of optimality
- System R

Heuristics + cost-based optimizer

Use static rules to perform initial optimization.

Then use dynamic programming
to determine the best join order for tables.

→ **Bottom-up planning** using divide-and-conquer

→ The first cost-based query optimizer

Example: **System R**, early IBM DB2, most open- source DBMSs

A small parenthesis

Cost & selectivity estimation

Cost Estimation

Generate an estimate of the cost of executing a plan for the current state of the database.

→ Resource utilization (CPU, I/O, network)
→ Size of intermediate results
→ Choices of algorithms, access methods
→ Interactions with other work in DBMS
→ Data properties (skew, order, placement)

Cost Estimation – reminder

- Estimate cost for each physical operator
  - **Simplification:** Only consider I/O cost, number of pages
  - How valid is this?
    - Requires specialization to become main-memory-aware
- Examples
  - Selection without index, unsorted
  - Page-Oriented Nested Loop Join

Selection

Selection without index, unsorted

```
for each record r in R
 if (r.age<18)
  add r to result
```

Let's unveil I/O

Selection

Selection without index, unsorted

```
for each record r in R
 if (r.age<18)
  add r to result
```

```
for each page p in R
 for each record r in p
  if (r.age<18)
   add r to result
```

- I/O Cost: number of read pages
  - **Here** we don't consider #pages written. Why?
- Cost will change if
  - Records are sorted based on the condition attribute
  - We can utilize an index to filter out some records
  - We need to materialize the output result
- We will also use different physical implementation

Page-oriented Nested Loop join

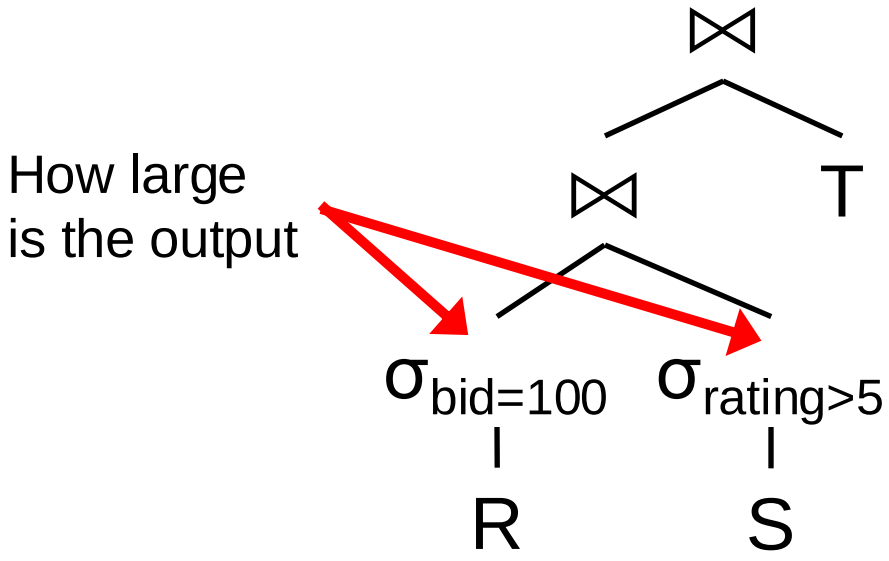
```
for each record r in R
 for each record s in S
  if (r.id=s.id)
   add <r,s> to result
```

```
for each page p1 in R
 for each page p2 in S
  for each record r in p1
   for each record s in p2
    if (r.id=s.id)
     add <r,s> to result
```

} I/O

- For each tuple in the outer relation R, we scan the entire inner relation S
  - But use per-page loading!
- I/O Cost: #pages of R + #pages of R * #pages of S
- How to choose the outer relation to minimize the cost?
  - Choose order of R, S, so that #pages of R < #pages of S
  - Order benefits cost if tables are of different size

Selectivity estimates



Required for cost estimation

Output of selection is input of another operator!

Selectivity estimates

Estimating the number of (intermediary) results

```
SELECT * FROM R WHERE r.age=18
```

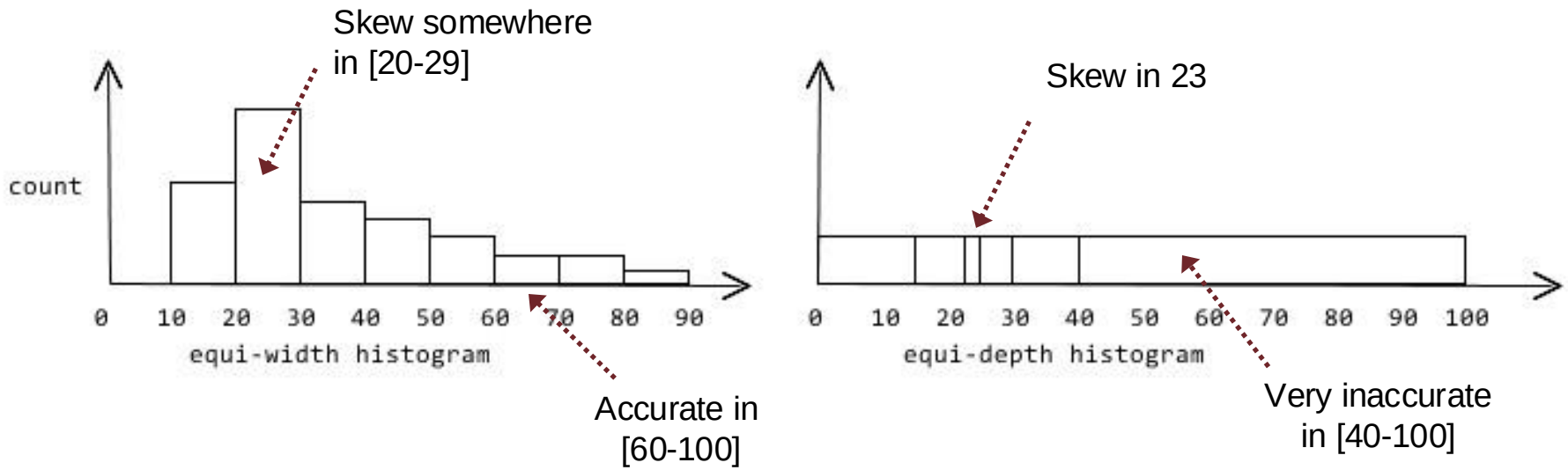
- Necessary to estimate cost of operators, e.g., join
- Crude estimation
  - Selectivity= $\frac{1}{\#Keys(R.age)}$
  - Estimated #results = $\frac{\#Records(R)}{\#Keys(R.age)}$ ← Number of distinct values
  - Range queries: length of the range/length of the domain
  - Free if there is an index!
  - Good estimates when values are uniformly distributed

Selectivity estimates

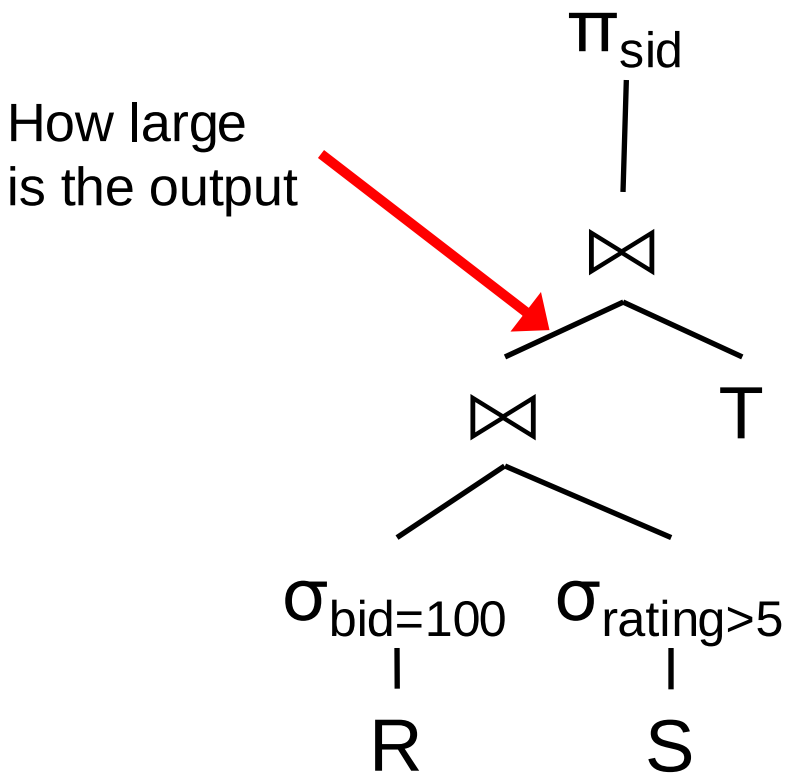
- Estimating the number of (intermediary) results

```
SELECT * FROM R WHERE r.age=18
```

- Histograms: Equi-width and Equi-depth
  - Higher cost to build and maintain, but better accuracy.



Join cardinality estimates



Required for cost estimation of each join

Reorder joins so that records are filtered as fast as possible!

Join cardinality estimates

$$\text{Selectivity} = \frac{1}{\max[\#Keys(R.sid), \#Keys(S.sid)]}$$

If unknown, use 10

$$\text{Cardinality estimate} = \frac{\#Records(R) * \#Records(S)}{\max[\#Keys(R.sid), \#Keys(S.sid)]}$$

More for cost estimation & statistics in the book. Chapters:

- “Evaluation of rel. operators” (CS-300 material)
- “Introduction to query optimization”
- “A typical relational query optimizer”

Outline

Introduction to query optimization

Relational algebra equivalences

Optimizers based on heuristics – INGRES

Optimizers based on heuristics & cost

- Cost & selectivity estimation
- **Principle of optimality**
- **System R**

SYSTEM R Optimizer

IBM SYSTEM R

- Seminal project from the 70s
- Drastic influence on succeeding DBs!

Pat Selinger
IBM Fellow,
Paradata,
Salesforce

Too many plans. Use heuristics to reduce the search space

High-level idea

- Iterate over the possible plans
  - Order of operators, physical implementations of operators, access paths
- Estimate cost of each plan
- Return the cheapest to the user

SYSTEM R Optimizer

Step 1: Break query up into blocks and generate the logical operators for each block.

Reduces complexity of each plan

Block: No nested queries, exactly one `SELECT` & `FROM`, and at most one `WHERE, GROUP BY, HAVING`

```
SELECT S.sid, MIN (R.day)
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
 AND S.rating = (SELECT MAX (S2.rating) FROM Sailors S2)
GROUP BY S.sid
HAVING COUNT(*)>1
```

SYSTEM R Optimizer

Step 1: Break query up into blocks and generate the logical operators for each block.

Reduces complexity of each plan

Block: No nested queries, exactly one `SELECT` & `FROM`, and at most one `WHERE, GROUP BY, HAVING`

Nested Block:
SELECT MAX (S2.rating)
FROM Sailors S2

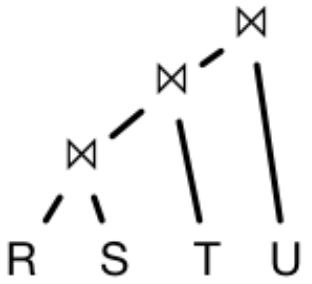
```
SELECT S.sid, MIN (R.day)
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
 AND S.rating = Reference to Nested Block
GROUP BY S.sid
HAVING COUNT(*)>1
```

SYSTEM R Optimizer

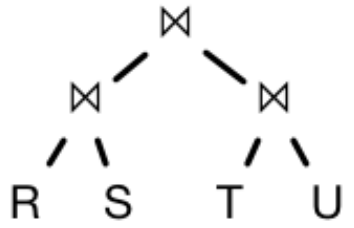
Step 1: Break query up into blocks and detect the logical operators in each block.

Step 2: For each individual block:

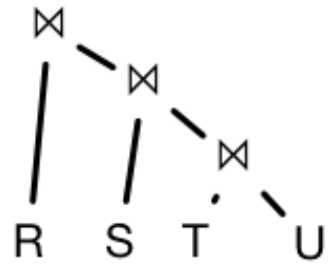
- For each logical operator, consider a set of physical operators & offered access paths.
- *Iteratively construct* a "left-deep" tree that minimizes the estimated amount of work to execute the plan.
  - Why left-deep?



Left-deep join plan



Bushy join plan



Right-deep join plan

SYSTEM R Optimizer

Retrieve the names of artists that appear on Joy's mixtape

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
      AND APPEARS.ALBUM_ID=ALBUM.ID
      AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

ARTIST: Sequential scan
APPEARS: Sequential scan
ALBUM: Index lookup on name

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

APPEARS

ARTIST_ID	ALBUM_ID
FK	FK

Q4

ARTIST

ID	NAME
Integer, PK	VARCHAR(32)

ALBUM

ID	NAME
Integer, PK	VARCHAR(32) UNIQUE

ARTIST, APPEARS: No predicate in the WHERE clause implies table scan

ALBUM: Index would help (assume it exists)

SYSTEM R Optimizer

Retrieve the names of artists that appear on Joy's mixtape

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

ARTIST: Sequential scan
APPEARS: Sequential scan
ALBUM: Index lookup on name

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

Step #2: Enumerate all possible join orderings for tables

All possible join orders

ARTIST ⋈ APPEARS ⋈ ALBUM
APPEARS ⋈ ALBUM ⋈ ARTIST
ALBUM ⋈ APPEARS ⋈ ARTIST
… … …

How many are these?
$3 \times 2 \times 1$

SYSTEM R Optimizer

Retrieve the names of artists that appear on Joy's mixtape

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

ARTIST: Sequential scan
APPEARS: Sequential scan
ALBUM: Index lookup on name

keep cheapest access methods
+
those producing an *interesting order

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

Step #2: Enumerate all possible join orderings for tables

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

possible join orders?

ARTIST ⋈ APPEARS ⋈ ALBUM
APPEARS ⋈ ALBUM ⋈ ARTIST
ALBUM ⋈ APPEARS ⋈ ARTIST
… … …

**tuple ordering as needed by a group-by, an order-by, or a join*

Join ordering

Scales BADLY with #joins N

Naïve: Just try all possible orders

$$N * (N-1) * (N-2) \ldots * 3 * 2 * 1 = N!$$

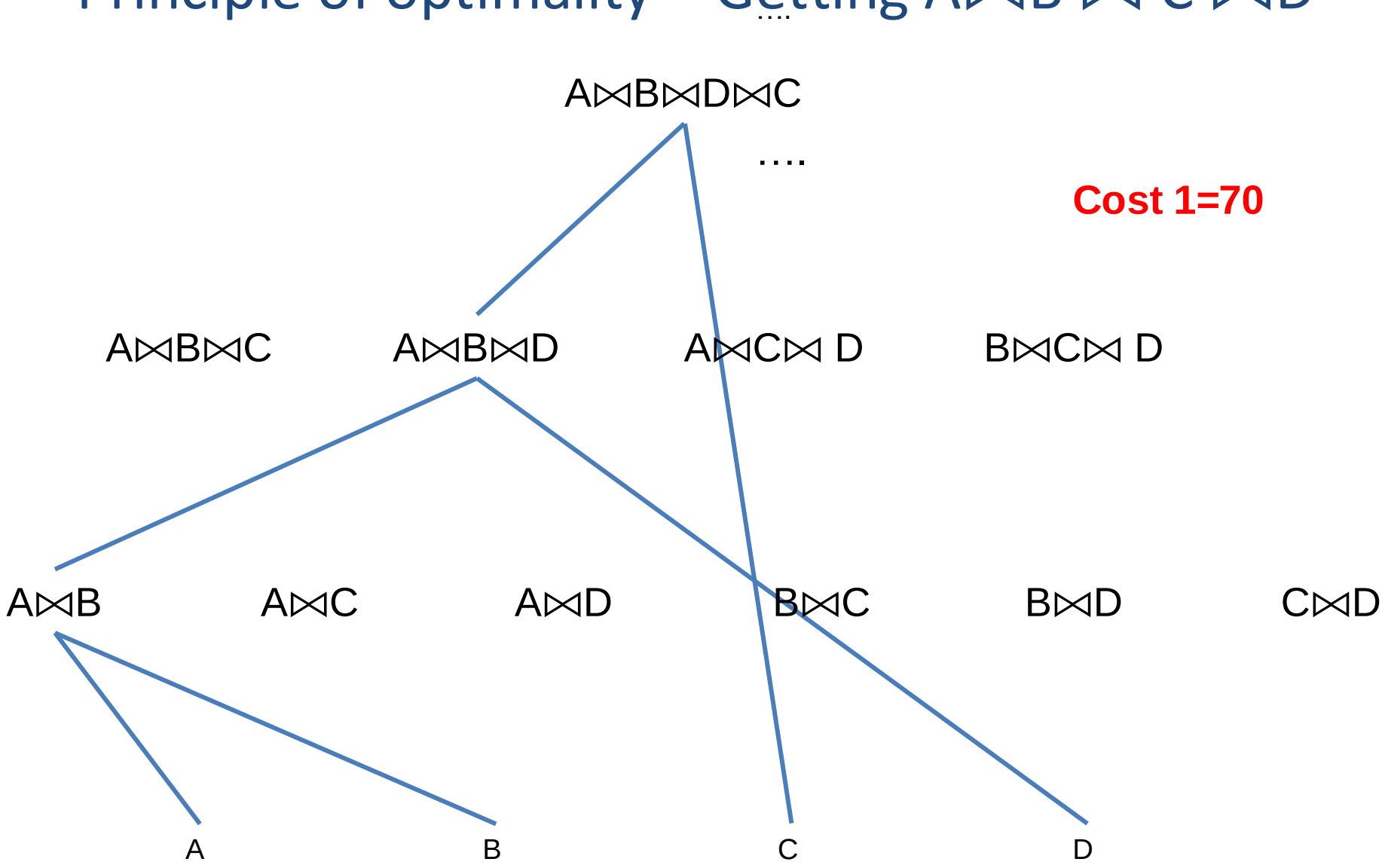
Principle of optimality: The optimal plan for k joins is produced by extending the optimal plan(s) for k-1 joins!

- To find optimal order of A ⋈ B ⋈ C ⋈ D, reuse partial solutions for optimal order of A ⋈ B ⋈ C, A ⋈ B ⋈ D,
  A ⋈ C ⋈ D, and B ⋈ C ⋈ D.
- Dynamic programming : $O(N \times 2^{N-1})$
- N=10 : 5120 Vs 3.6 Million

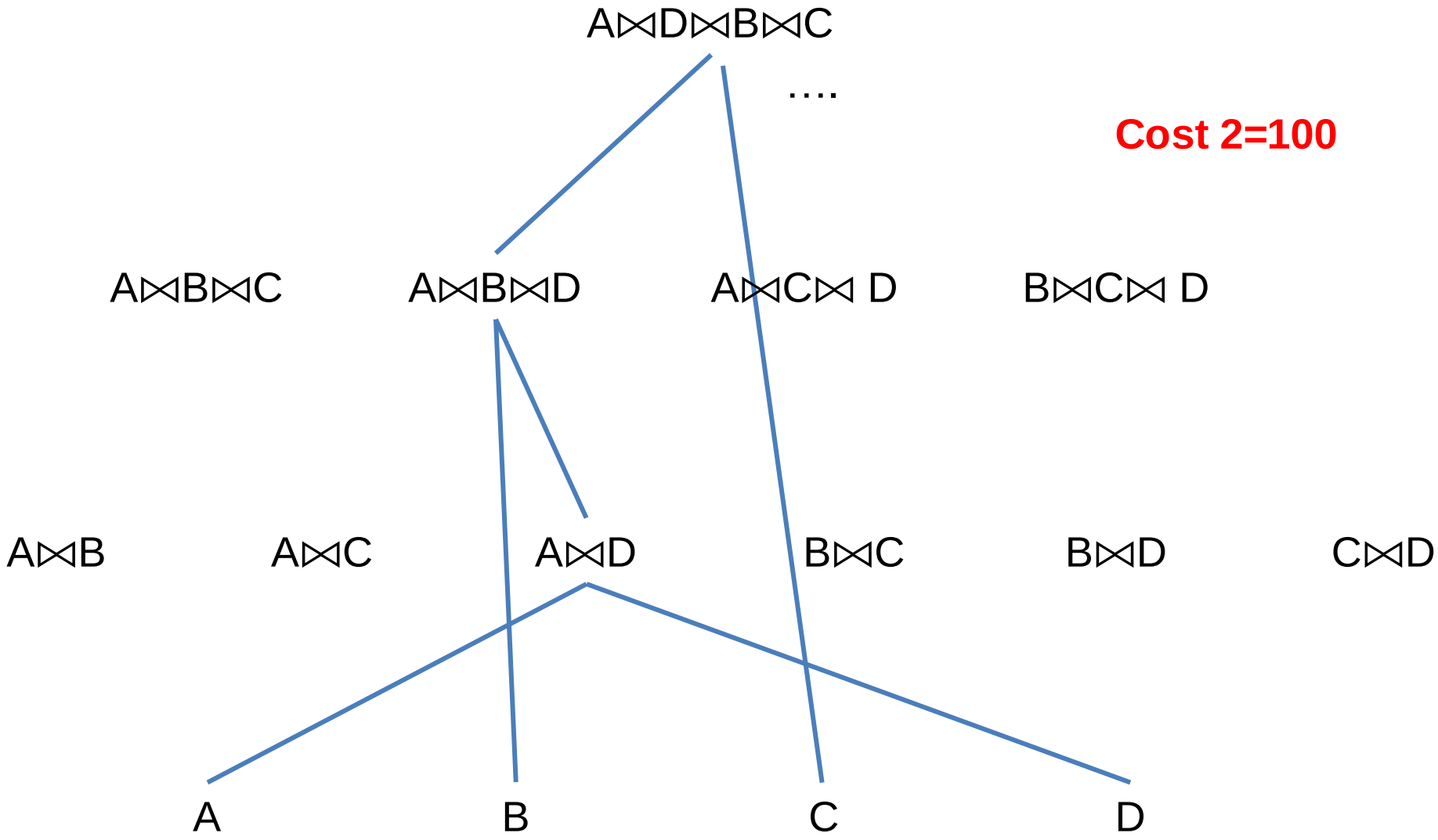
Still expensive but **feasible**

Assume principle of optimality!

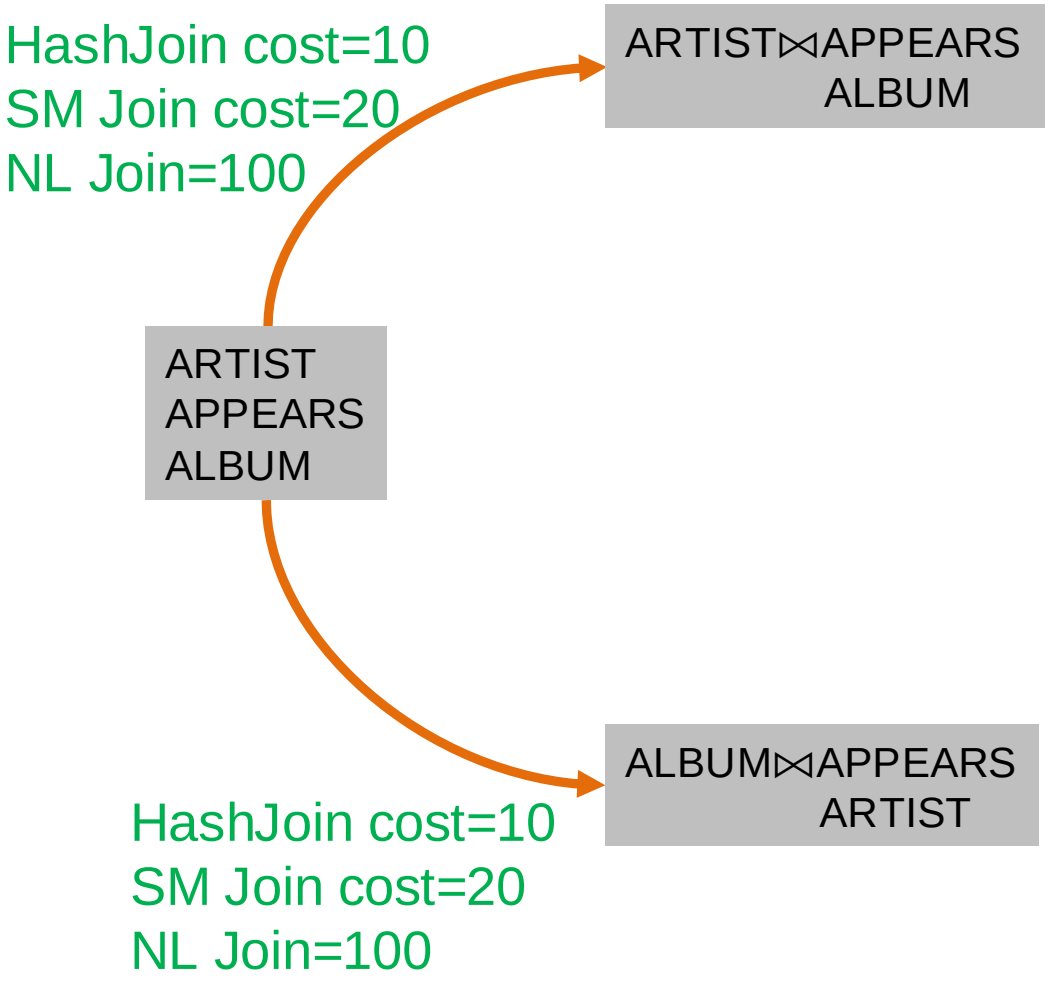
Principle of optimality – Getting A⋈B ⋈ C ⋈D



Choose the **cheapest path** to create A⋈B⋈D, and consider it fixed for higher levels!



Back to our running example…

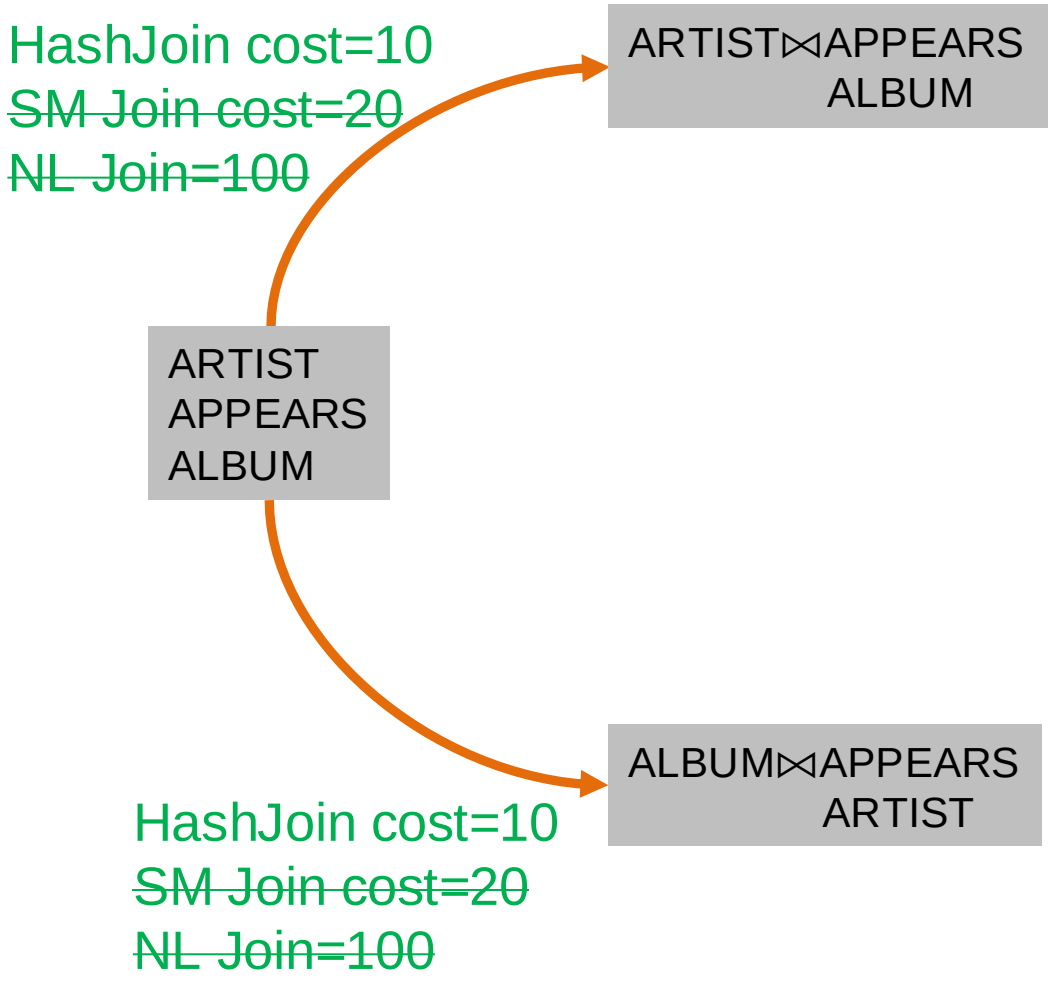


ARTIST⋈APPEARS⋈ALBUM

…

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

Back to our running example…

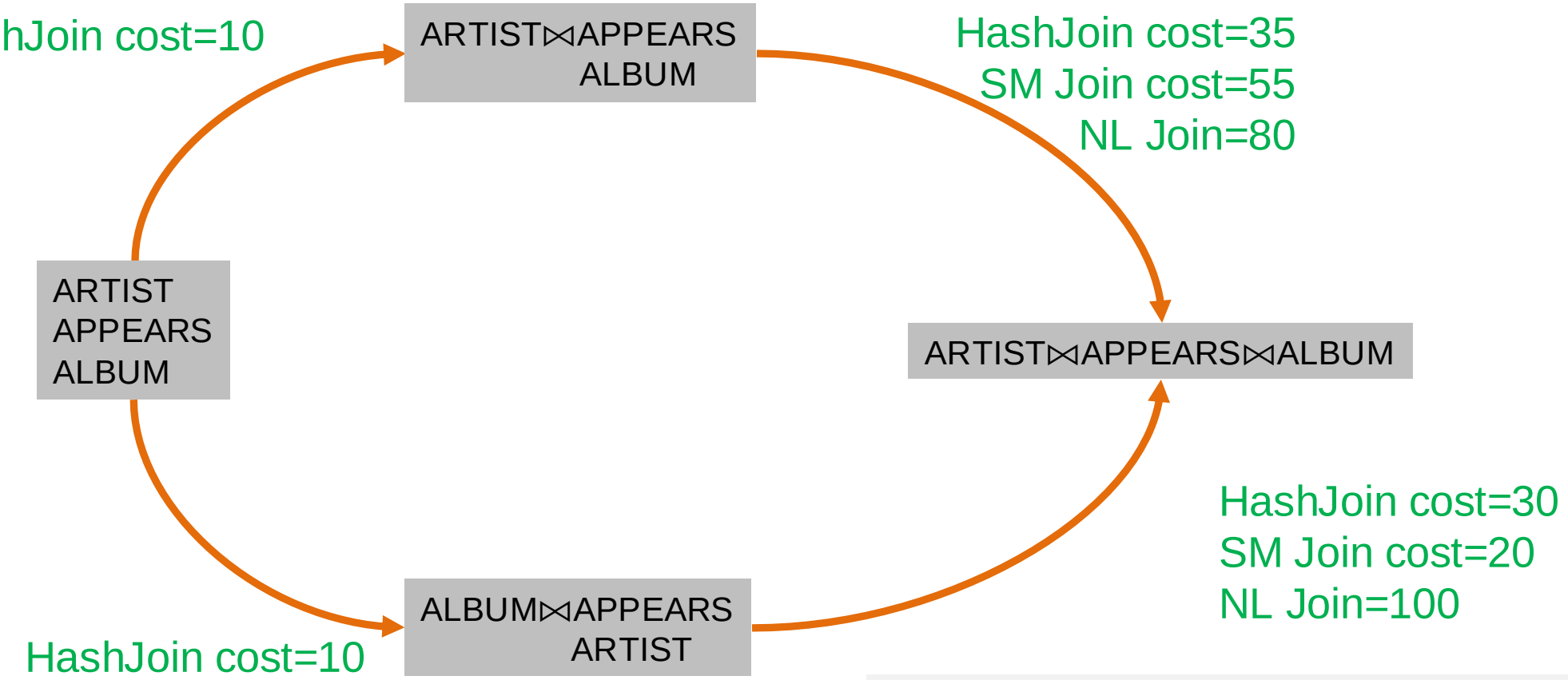


ARTIST⋈APPEARS⋈ALBUM

…

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

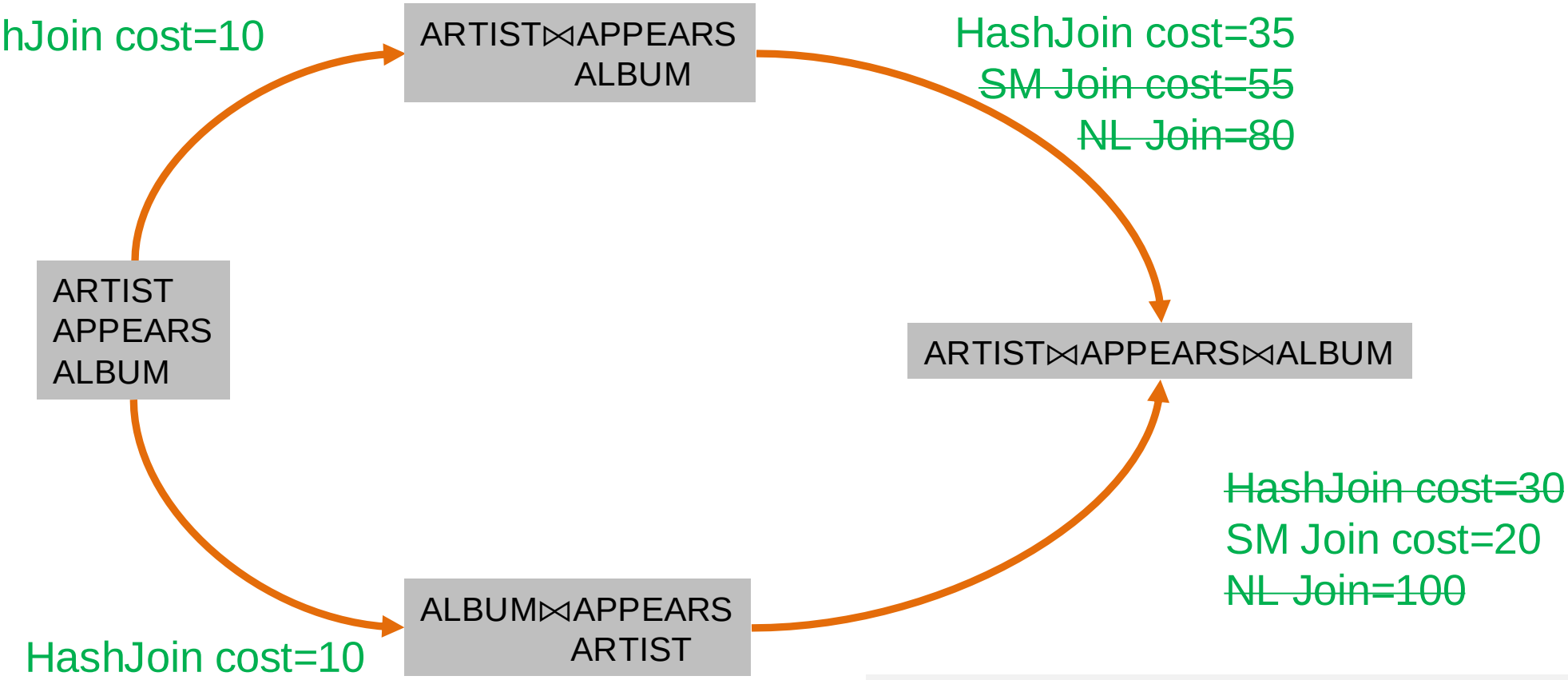
Back to our running example…



…

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

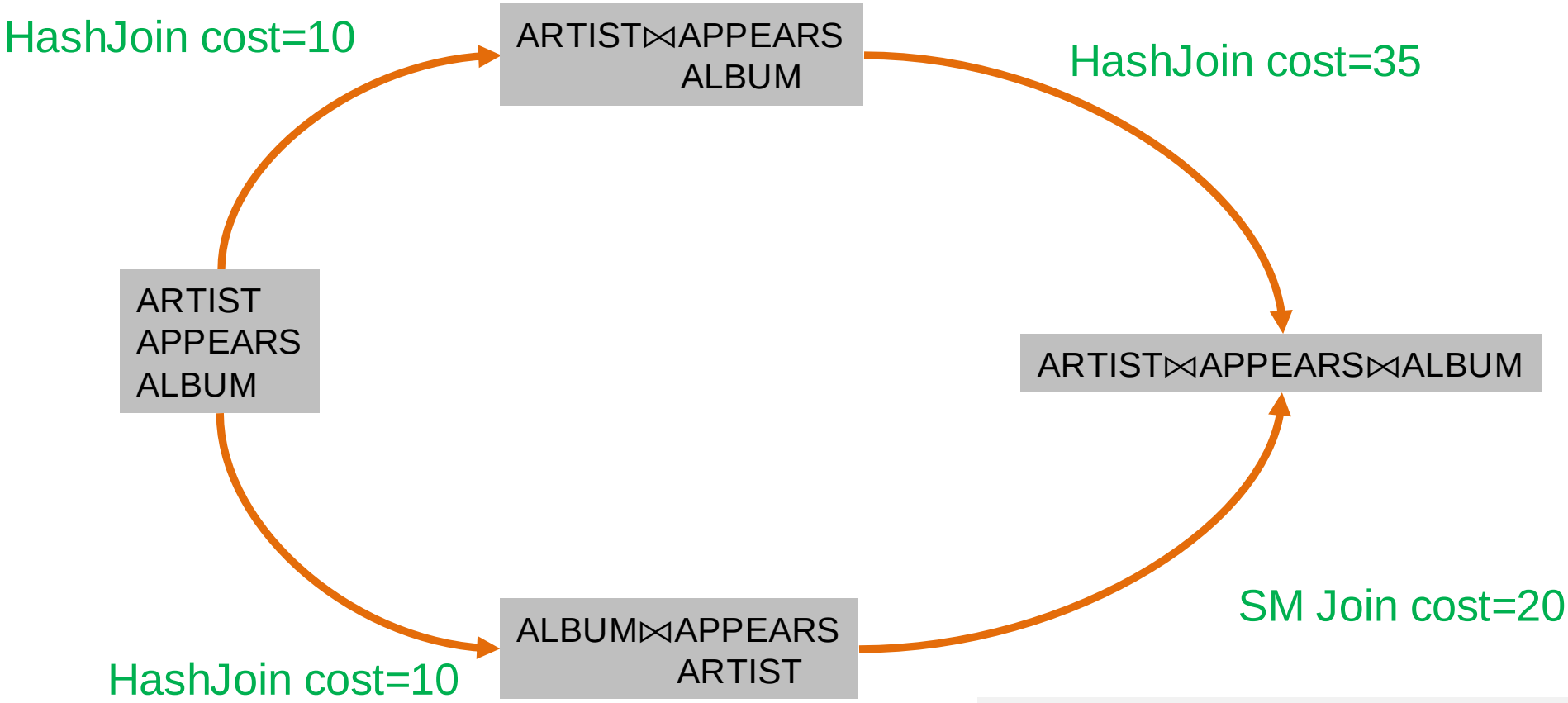
Back to our running example…



…

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

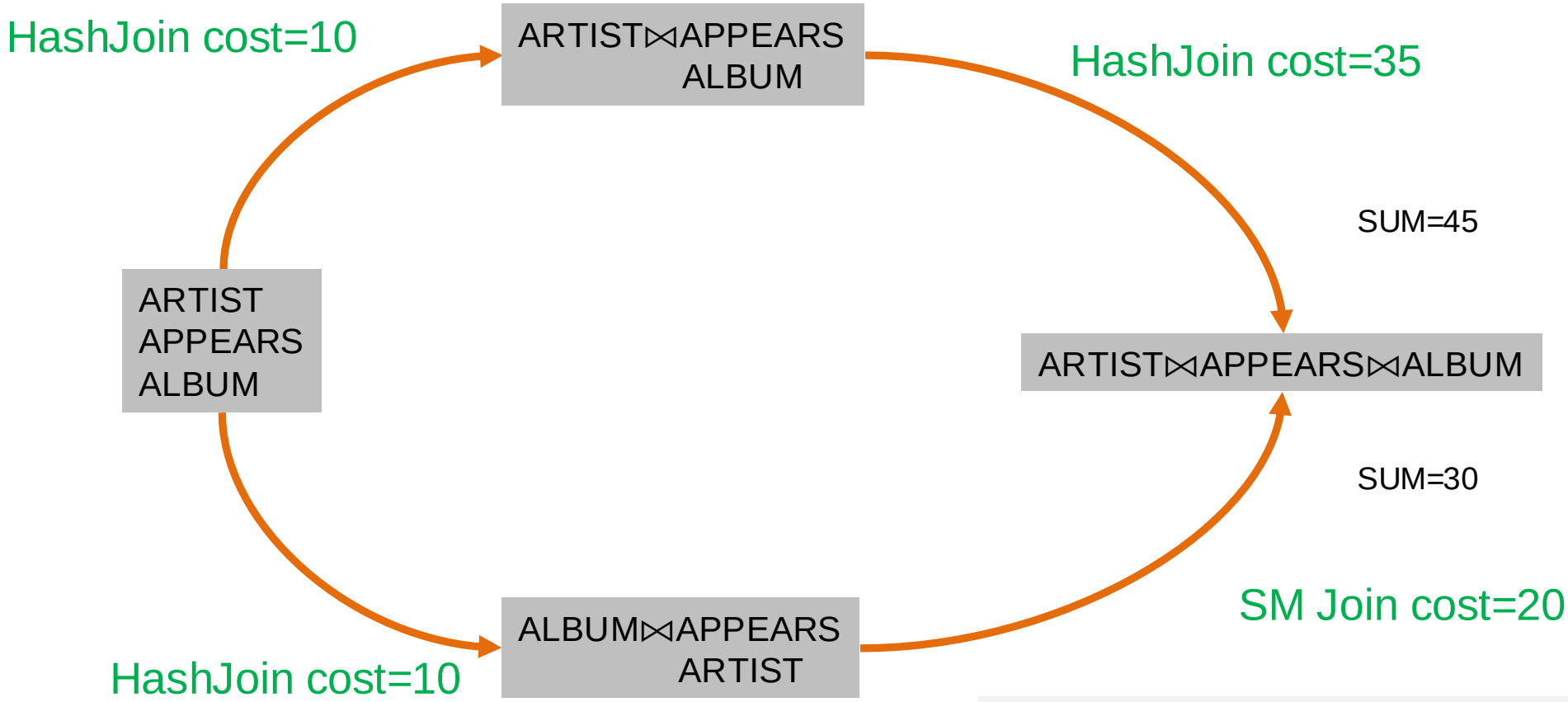
Back to our running example…



…

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

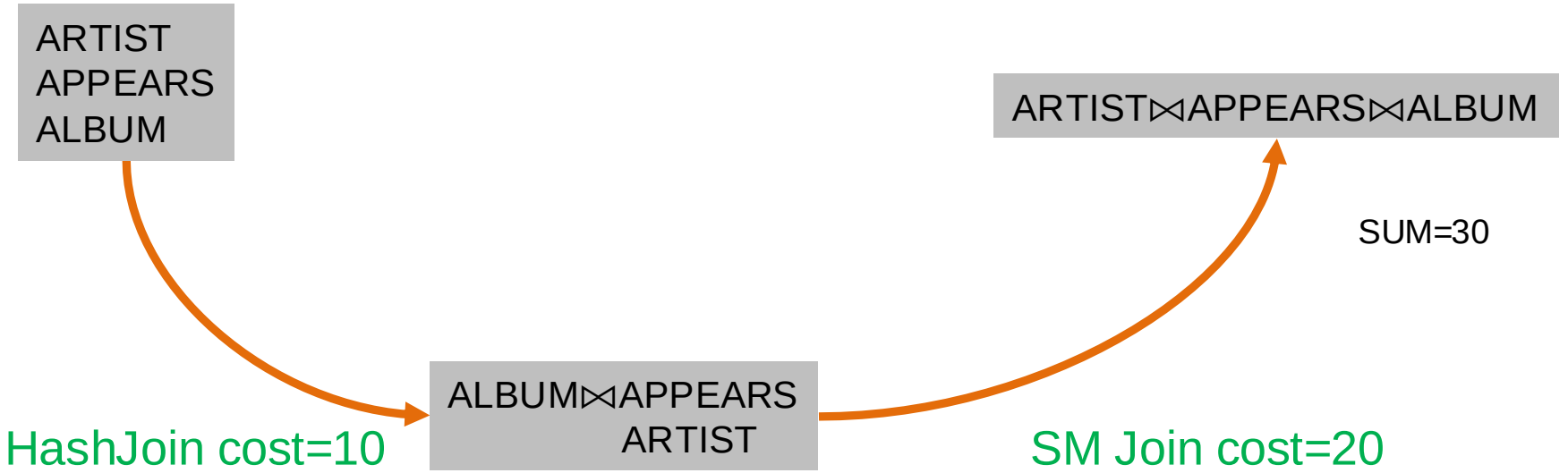
Back to our running example…



…

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

Back to our running example…



Constructed in parallel:

- Logical plan (join order)
- Physical plan (join implementation)

What did we use?

- Logical & Physical statistics
  - size of each record, #records, #distinct values in column, #data pages, #pages in index, …
- Selectivity estimates
  - Histograms (or other distribution assumptions)
  - Formulas for selectivity estimates: assumption of selectivity independence!
- Formulas to estimate IO costs (possibly also CPU) per operator
  - Access methods (index or scan), natural orders (e.g., primary key, …)
  - Order of output data stream
- The principle of optimality

Principle of optimality Revisited!

- Principle of optimality may lead to suboptimal plans
  - E.g., order not considered
  - Additional cost at the end – avoided by sort merge join!

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
        AND APPEARS.ALBUM_ID=ALBUM.ID
        AND ALBUM.NAME="Joy's Slag Remix"
ORDER BY ARTIST.ID
```

- **Relaxed principle of optimality**
  - Consider order-by clause! A plan is compared with all other plans that produce the same order

Interesting orders

A tuple order is interesting order if that order is specified in:

- Group By clause
- Order By clause

} Will cause an additional sorting

Choose plans that produce the correct order, e.g.,

- Sort-merge join instead of NL join

Later generalized to include any *physical properties*!

Selectivity estimates, revisited

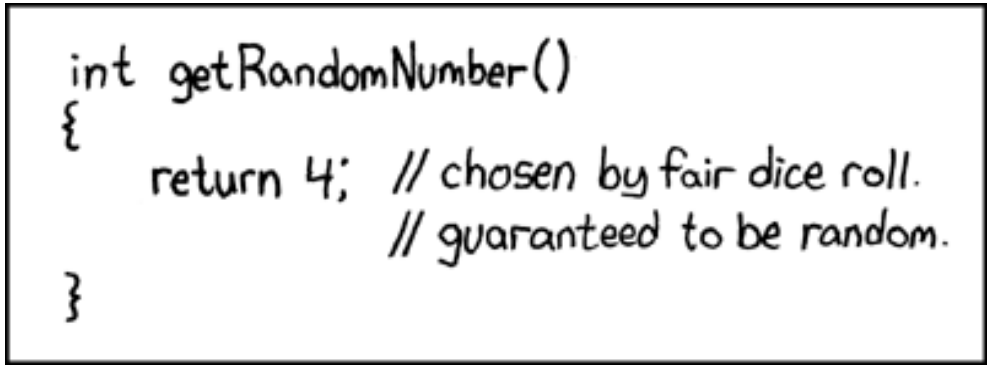
if there is no index, and no histogram or complex predicates

Cannot estimate $\#\mathrm{Keys}(\mathrm{R.\,age})$

When everything else fails, revert to magic

Set $\#Keys(R.age) = 10$

$$\frac{\#Records(R)}{\#Keys(R.age)} = \frac{\#Records(R)}{10}$$



Source: https://xkcd.com/221/

Anecdote

Guy Lohman

"Is Query Optimization a "Solved" Problem?"

The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities. Everything in cost estimation depends upon how many rows will be processed, so the entire cost model is predicated upon the cardinality model. In my experience, the cost model may introduce errors of at most 30% for a given cardinality, but the cardinality model can quite easily introduce errors of ***many orders of magnitude***! I'll give a real-world example in a moment. With such errors, the wonder isn't "Why did the optimizer pick a bad plan?" Rather, the wonder is "Why would the optimizer ever pick a decent plan?"

Summary of System R optimizer

Both heuristics and cost

Efficient and usually derives reasonable plans

Relies on

- Principle of optimality
- Interesting orders

System R was never commercialized but was hugely influential!

Heuristics + cost-based optimizer

Advantages:

→ Usually finds a reasonable plan without having to perform an exhaustive search.

→ Order of results also considered!

Disadvantages:

→ Depends on heuristics: Left-deep join trees are not always optimal (particularly for modern hardware)

→ Space exploration **only** considers joins

AI/ML for database systems

- AI captures non-obvious patterns/topologies
- Can help multi-objective optimization
- Efficiently explore large design spaces
- Adaptive optimization
- User workload affinity
- But: overhead can overpower benefits

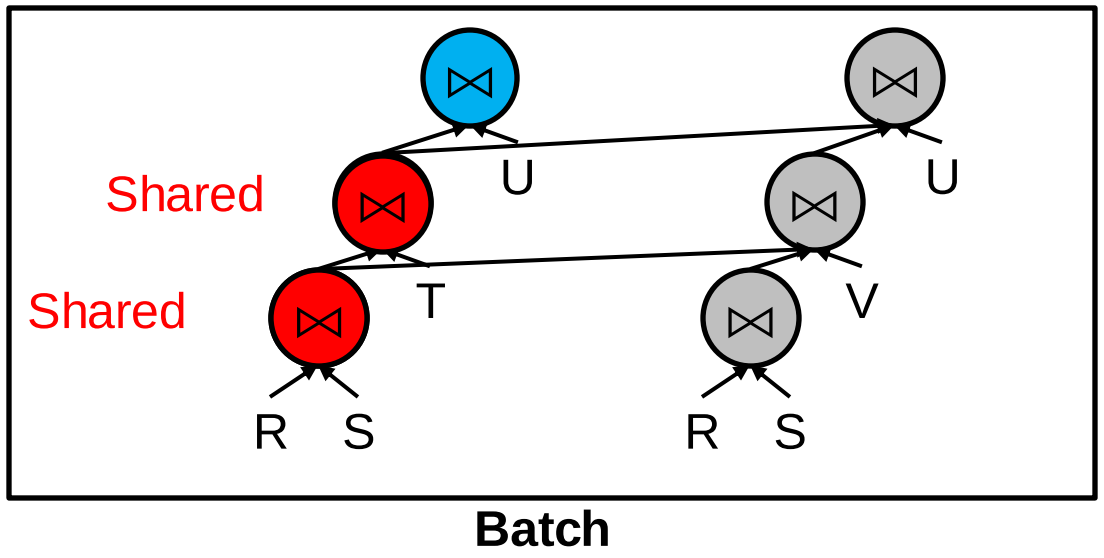
Use case: dynamic query optimization

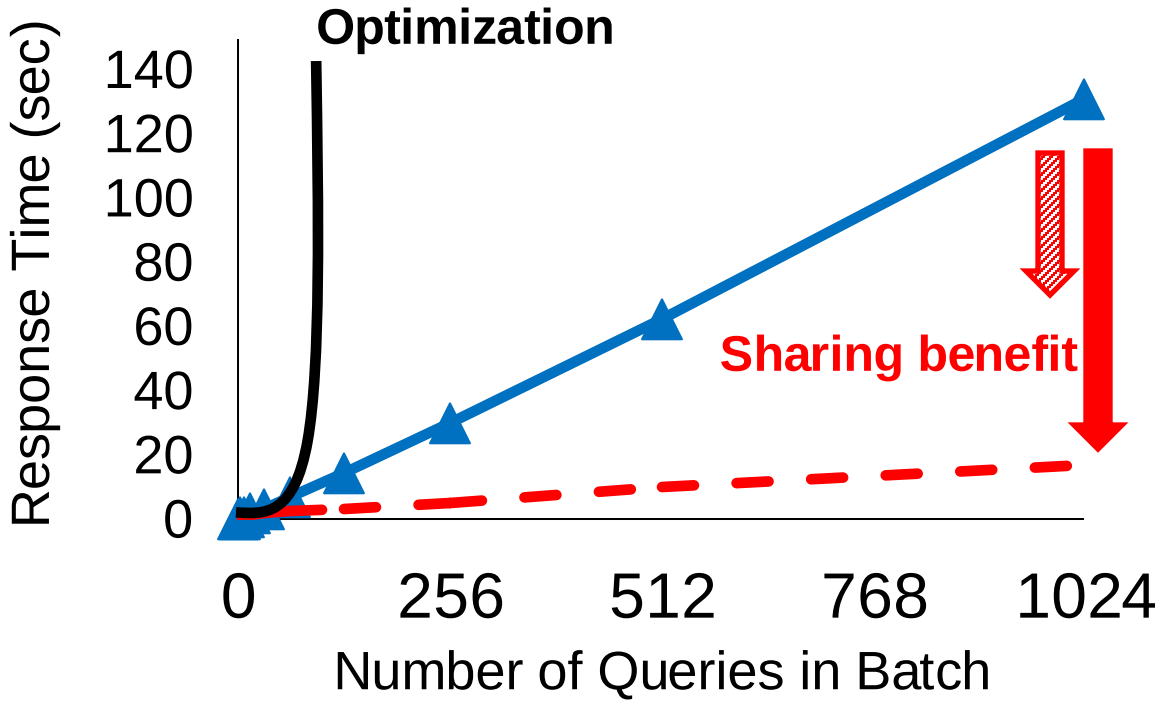
The Shared Query Execution Dilemma

[Sioulas SIGMOD21]

Q1: … WHERE **R.a=S.a and R.b=U.b** and R.c=T.c

Q2: … WHERE **R.a=S.a and R.b=U.b** and S.d=V.d





Heuristics can detect **some** opportunities **fast**

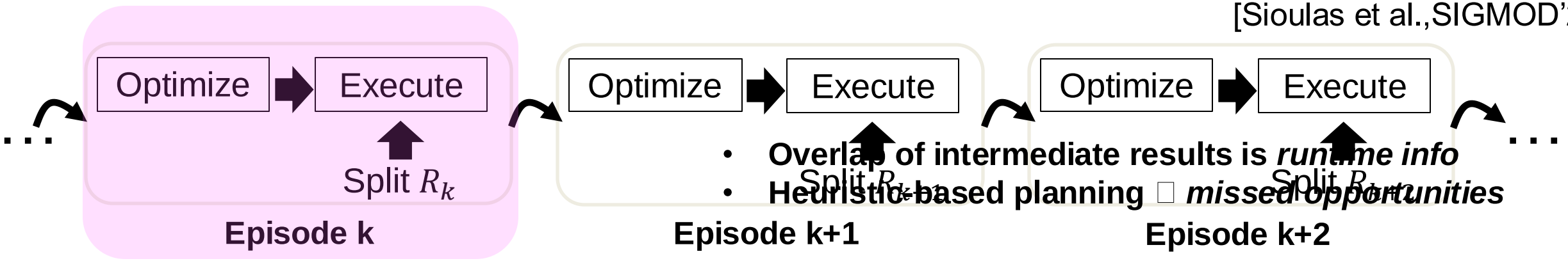
Optimization can choose **best** opportunities **at high cost**

Too much and too little optimization hurt performance!

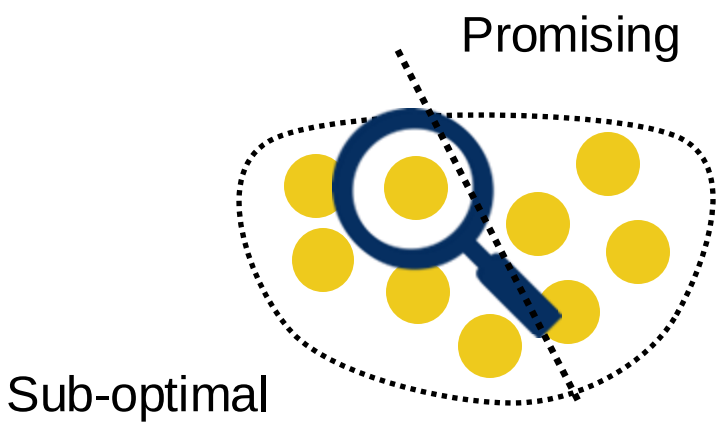
All-or-nothing exploration sacrifices scalability or

RL-enabled workload-conscious sharing

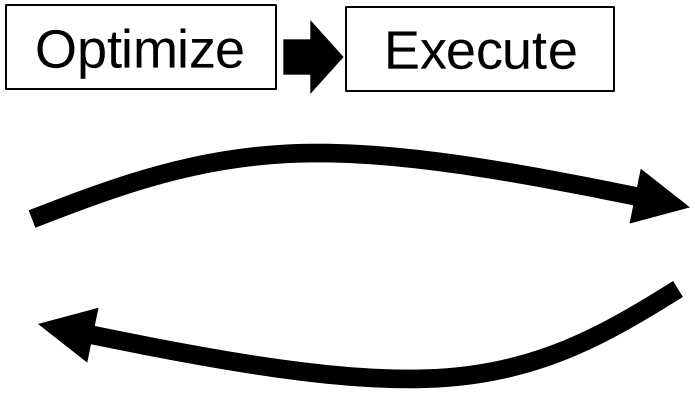
[Sioulas et al.,SIGMOD'21]

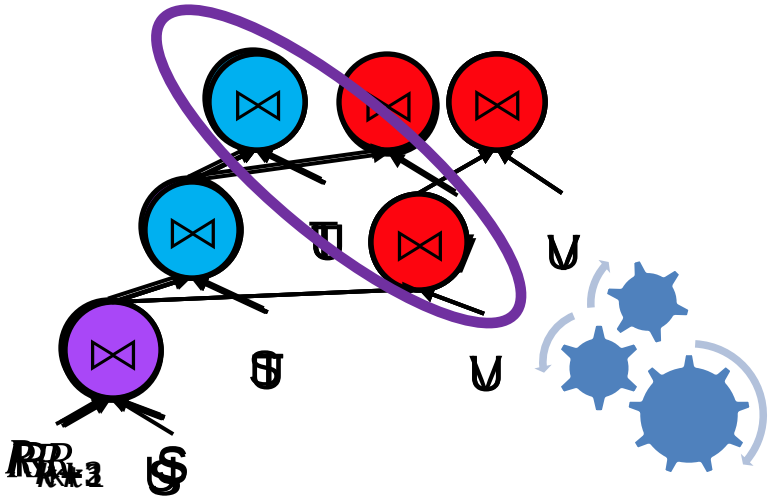


- **Overlap of intermediate results is *runtime info***
- **Heuristic-based planning → *missed opportunities***



Global Plans





Executor

Learn more and better sharing opportunities

Reading material

- COW Book chapters 13 and 14 or Database System Concepts chapter 16