CS422
Database systems

Query Optimization

Data-Intensive Applications and Systems (DIAS) Laboratory
École Polytechnique Fédérale de Lausanne

"Man plans, and God laughs"
– Yiddish proverb

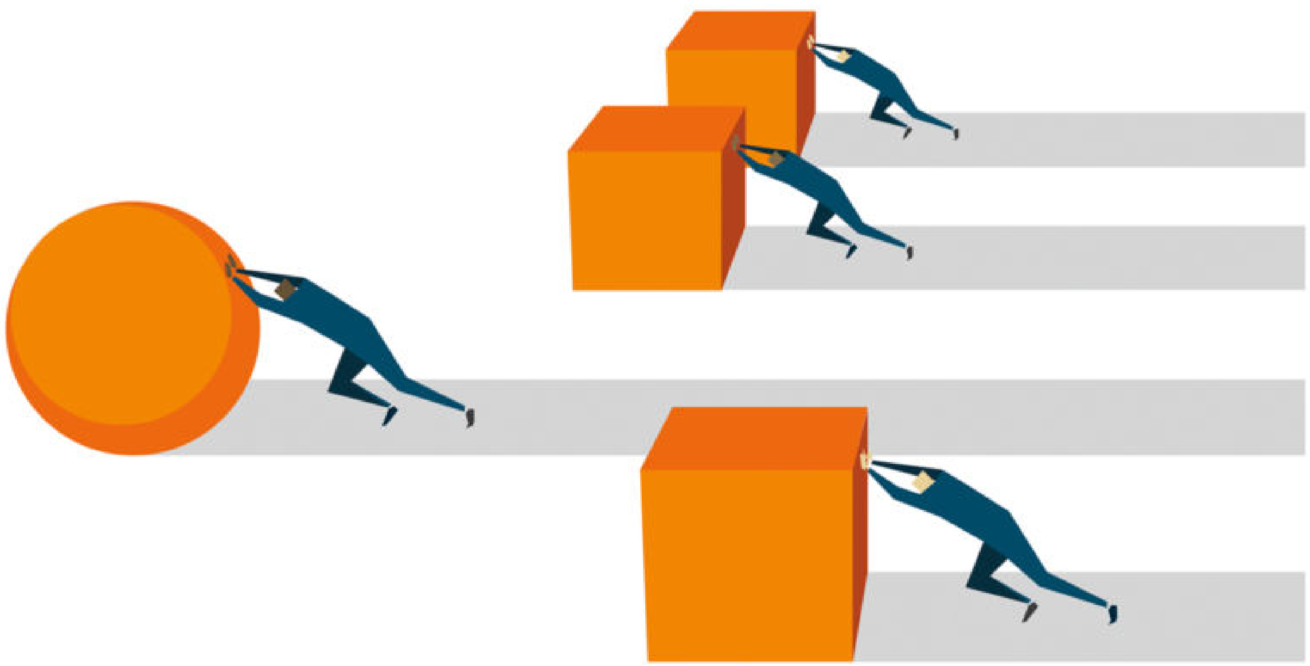
Some slides adapted from:

- Andy Pavlo
- CS-322

Today's overview

<u>Query Optimization</u>

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



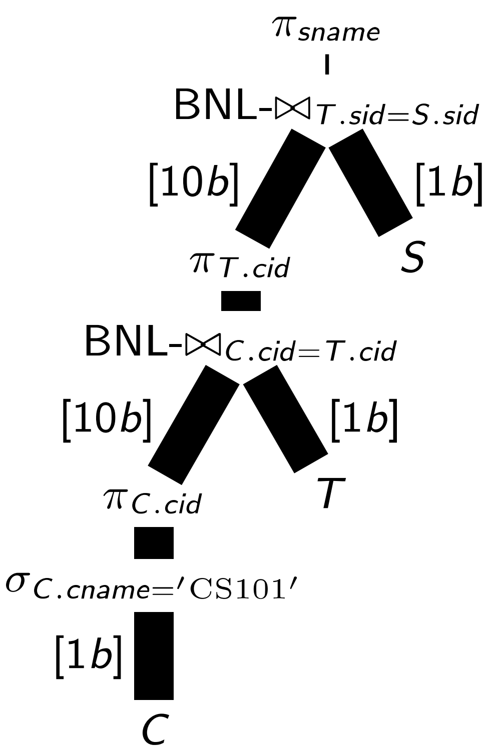
Today's overview

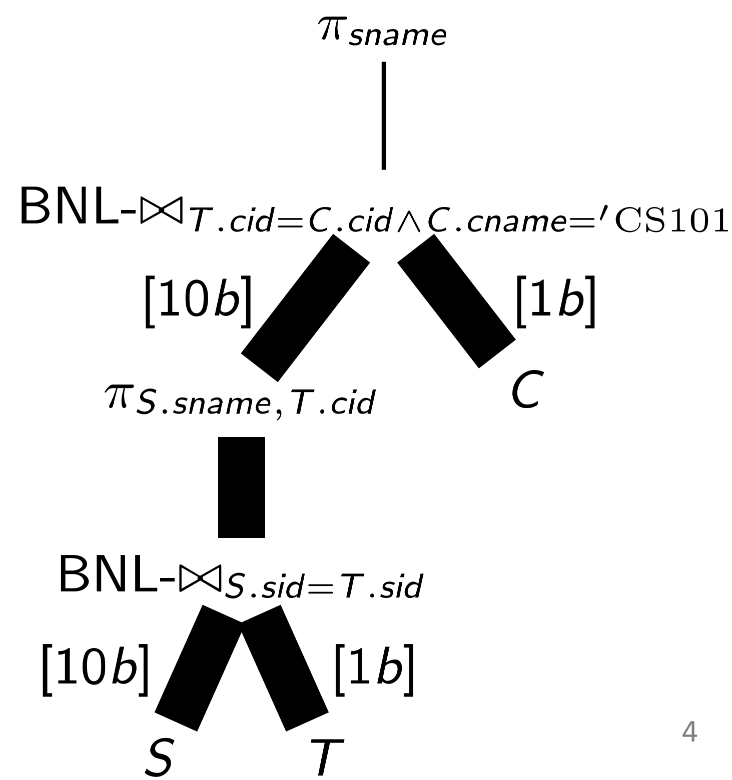
- Introduction to query optimization
- Relational algebra equivalences
- Optimizers based on heuristics – INGRES
- Optimizers based on heuristics & cost - SYSTEM R
  - Cost & selectivity estimation
  - Principle of optimality
- Multi-query optimization

Query Optimization

- For a given query, find an 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 well.
  - 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 to be taken

- Order of operations
  - Particularly: relative order of joins
- Implementation for each operation
  - E.g., hash joins, nested loop joins, sort-merge joins…
- Access methods
  - E.g., scan, use of an index
- Bad decisions can have a huge impact! E.g.
  - Use of a join algorithm
  - Pushing down selections (that make indexes useless)

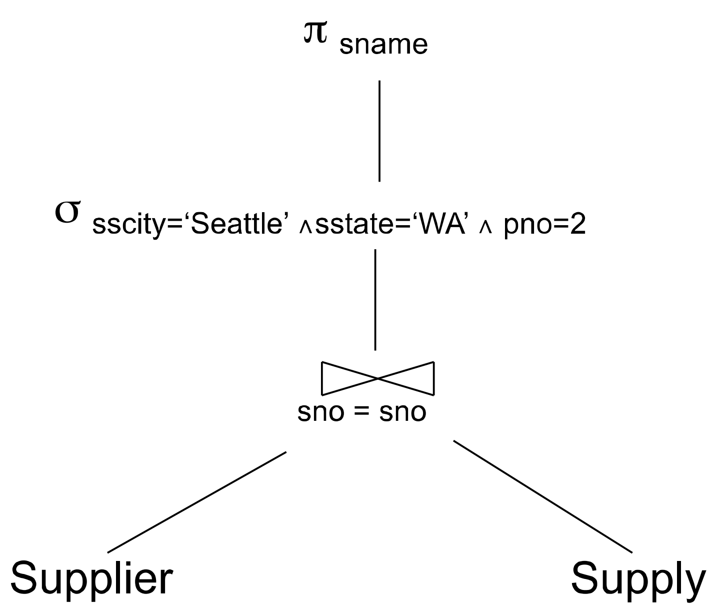
Input/output of query optimizer

- Input: An AST representing an SQL query

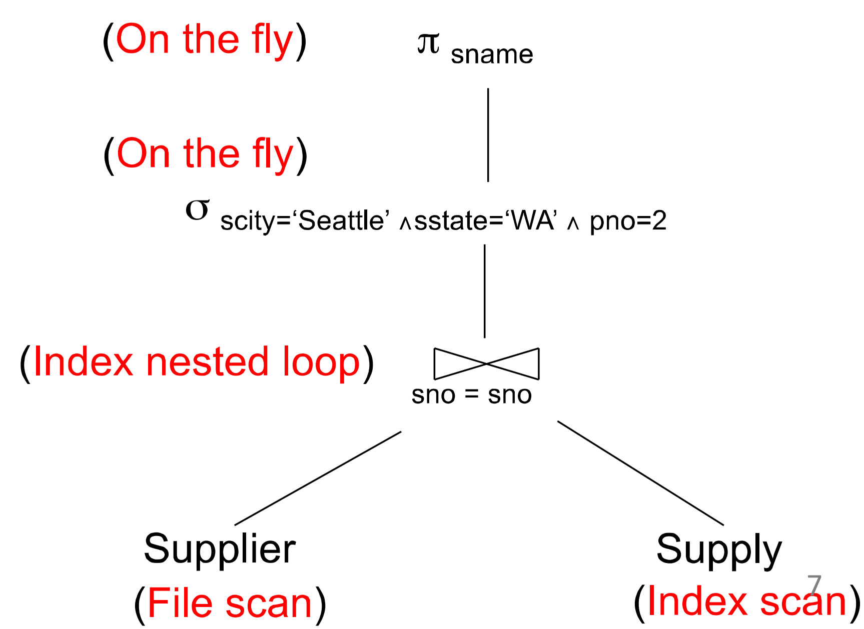
```
SELECT S.sname FROM Supplier S, Supply U
WHERE S.scity='Seattle' AND S.sstate='WA'
AND S.sno = U.sno AND U.pno = 2
```

- Output: A full physical plan → translatable **to code**

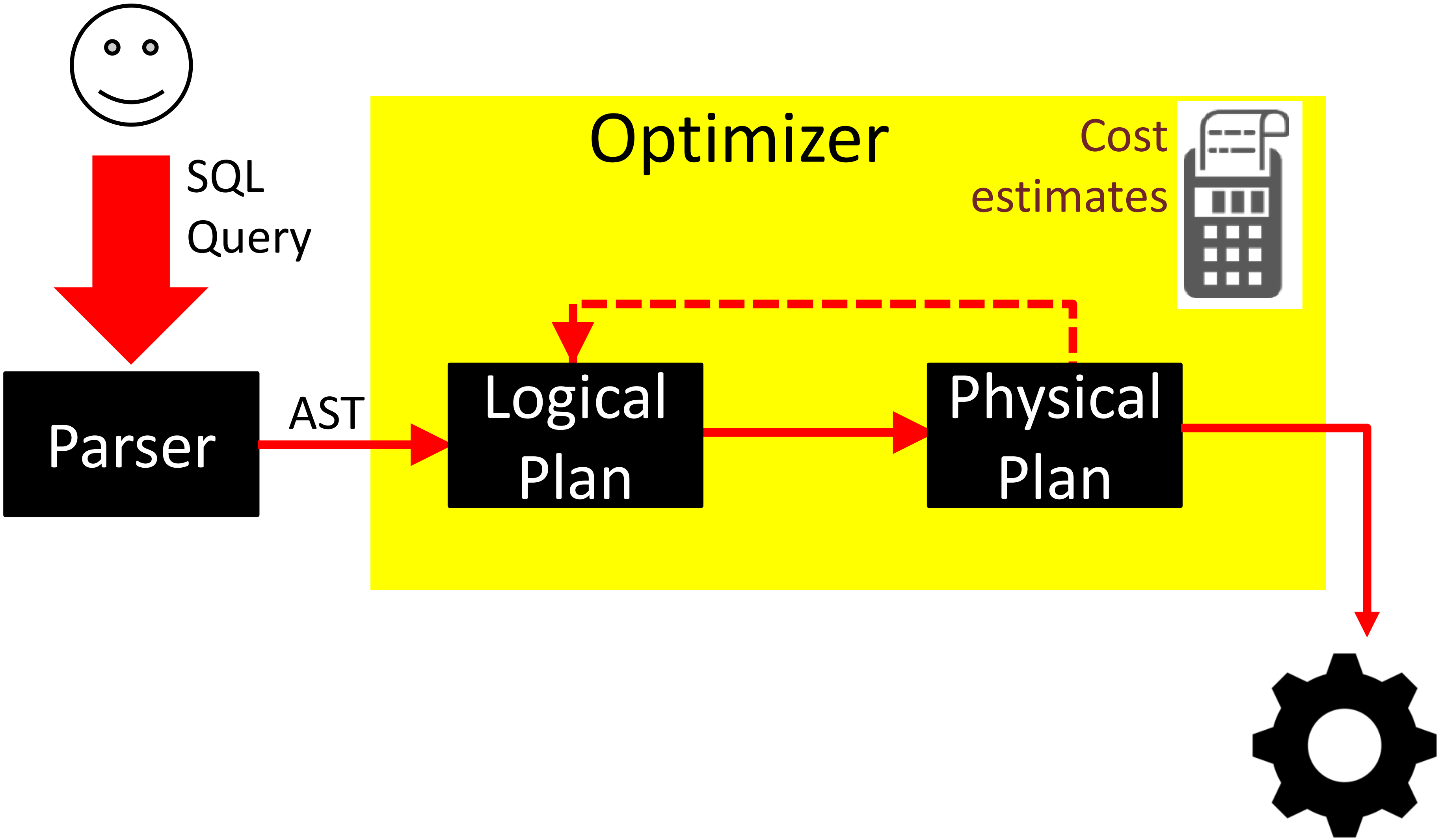
Logical plan (extracted from AST)



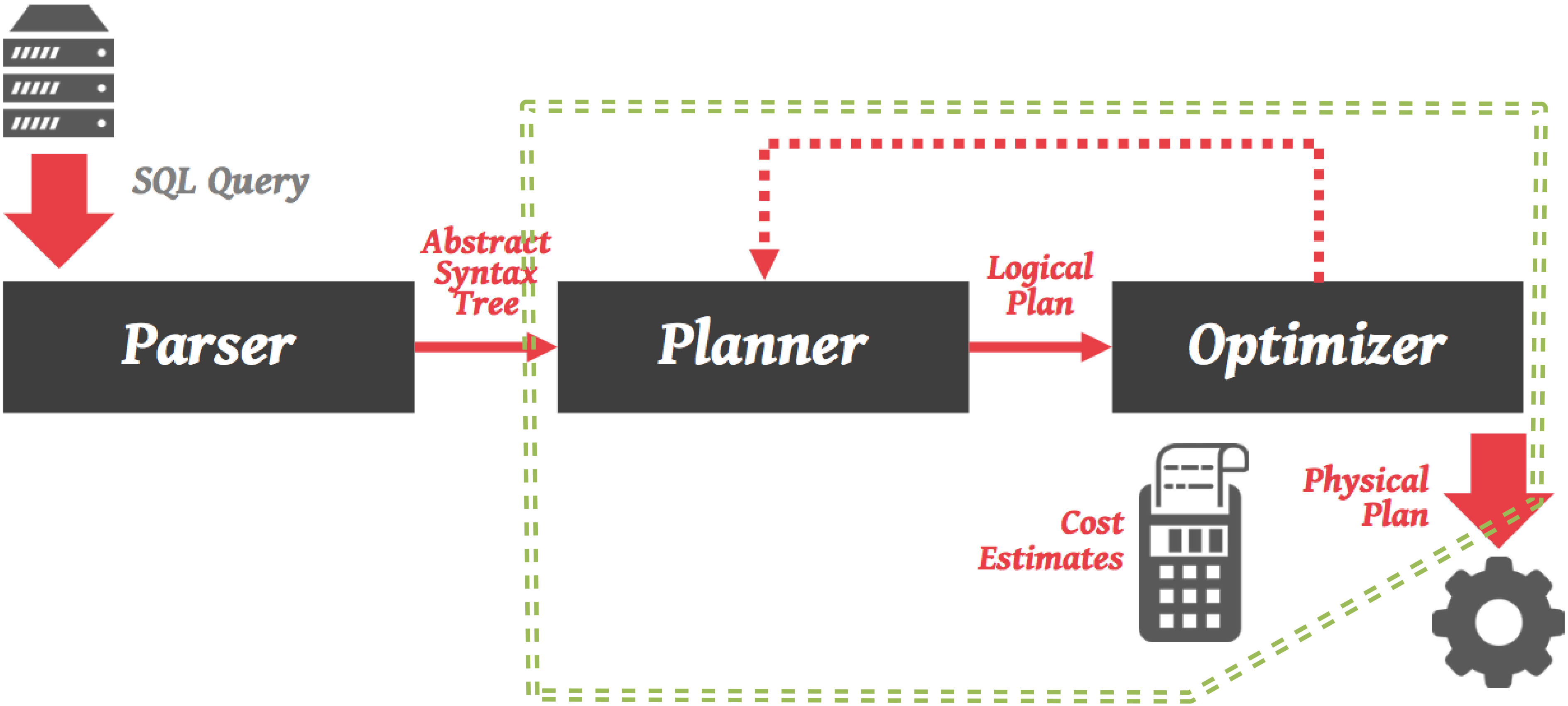
Physical plan



Classic architecture

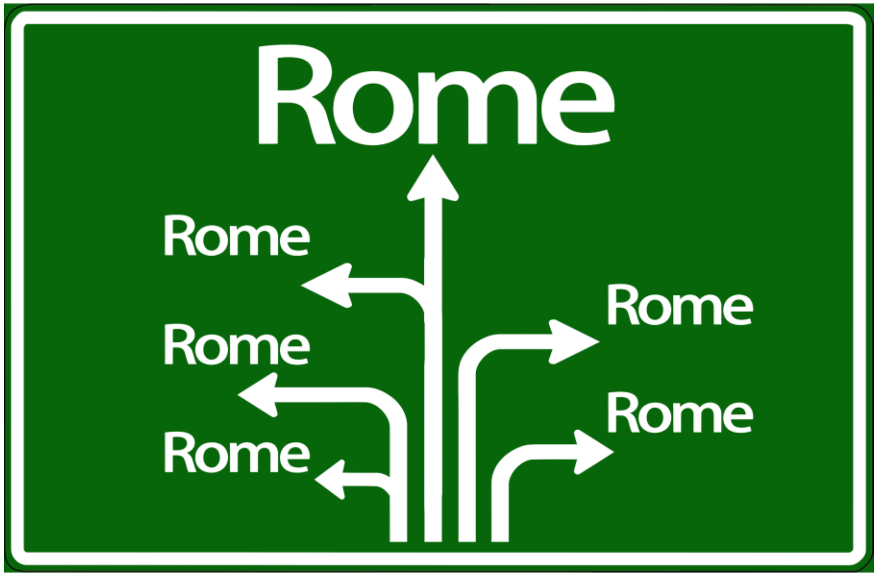


Classic architecture – Alternative view



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 …

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

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

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

$$\cancel{\pi_{age,rating}\ (\text{Sailors}) \leftrightarrow \pi_{age}\ (\pi_{rating}\ (\text{Sailors}))} \qquad (??)$$

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

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

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

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

- These equivalences allow us to choose different join orders

Mixing Joins with Selections & Projections

- Converting selection + cross-product to join

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

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

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

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

CAREFUL!
Not always wise

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

- We can also “push down” projections

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

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

An example

- Assumption:
- S: 16000 tuples
  T: 256000 tuples
  C: 1600 tuples
- First attempt
  takes > 2000 years

```
select S.sname from S, T, C
where S.sid=T.sid
and T.cid=C.cid
and C.cname='CS101';
```

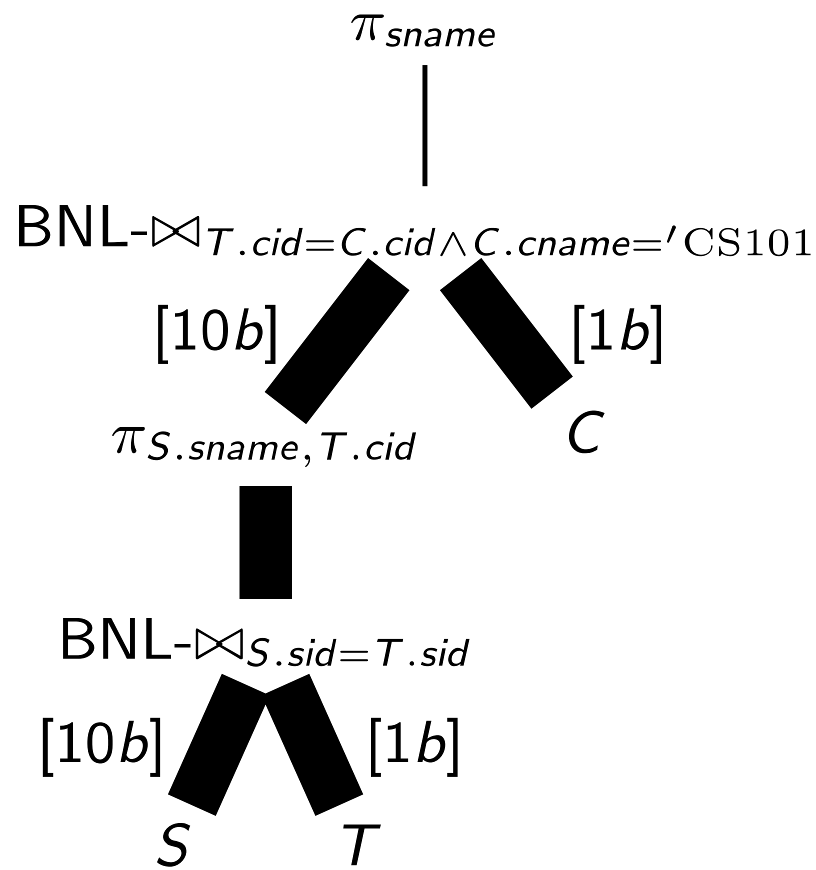
Using cross products!

```
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;
```

An example

- Assumption:
- S: 16000 tuples
  T: 256000 tuples
  C: 1600 tuples

```
select S.sname from S, T, C
where S.sid=T.sid
and T.cid=C.cid
and C.cname='CS101';
```

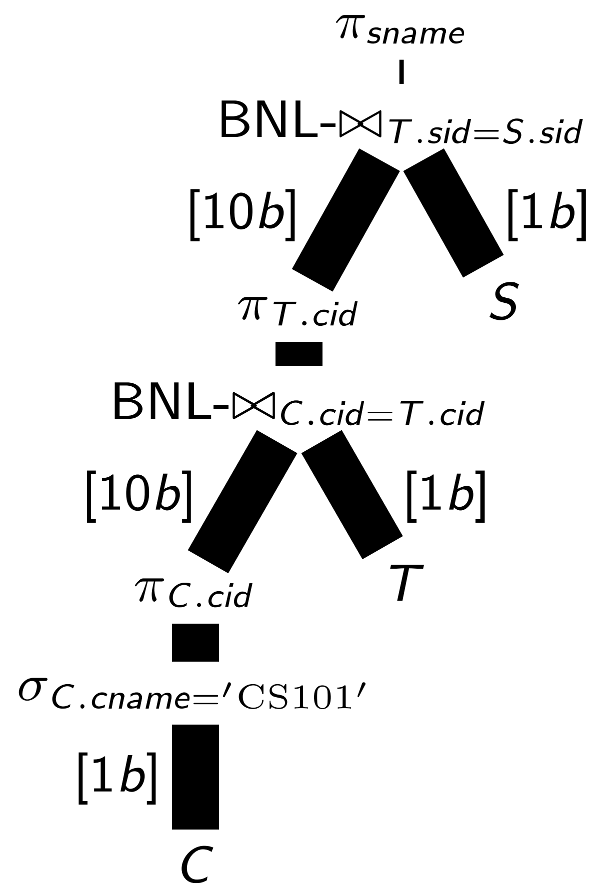


- Use joins instead of cross product & push down projection → 70 seconds

An example

- Assumption:
- S: 16000 tuples
  T: 256000 tuples
  C: 1600 tuples
- Push down selection and reorder joins → 0.35 seconds!

```
select S.sname from S, T, C
where S.sid=T.sid
and T.cid=C.cid
and C.cname='CS101';
```



Observation

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

Search Argument Able

- We will focus on OLAP queries in this lecture.

Optimization search strategies

- Heuristics
- Heuristics + Cost-based Join Order Search

TODAY

- Randomized Algorithms
- Stratified Search
- Unified Search

Not today

Heuristic-based optimization

- Define static rules that transform logical operators to a physical plan.
  → 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)

Stonebraker, Turing award 2014

Example - INGRES

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

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 people 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 people 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

Q2

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

INGRES optimizer

- ***Retrieve the names of people 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***”

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

Q1

SELECT ALBUM.ID AS ALBUM_ID **INTO** TEMP1
FROM ALBUM
WHERE ALBUM.NAME=“***Joy's Slag Remix***”

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 people 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

Q3

SELECT APPEARS.ARTIST_ID **INTO** TEMP2
FROM APPEARS, ~~TEMP1~~
WHERE APPEARS.ALBUM_ID=~~TEMP1.ALBUM_ID~~ 9999

Q4

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

Step #1: Decompose into single-variable queries

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

INGRES optimizer

- ***Retrieve the names of people 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 people 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"
```

Step #1: Decompose into single-variable queries

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

Q1

ALBUM_ID
9999

Q3

ARTIST_ID
123
456

Q4

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

NAME
George

NAME
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.

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 a divide-and-conquer search method
  → The first cost-based query optimizer
- Example: **System R**, early IBM DB2, most open- source DBMSs

Pat Selinger
IBM Fellow &
Paradata

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

Cost Estimation – brief reminder

- Estimate cost for each physical operator
  - **Simplification:** Only consider I/O cost, part. number of pages
  - How valid is this?
    - Requires specialization to become main-memory-aware
- Material of CS322
  - Details at book chapter "Evaluation of Relational Operators"
- 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

- 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

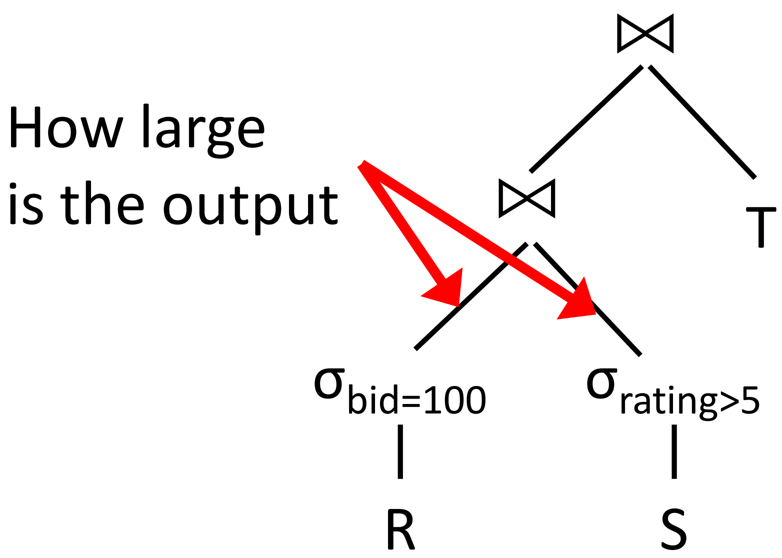
Page-oriented Nested Loop join

```
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    } I/O
 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
```

- 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, s.t. #pages of R < #pages of S
  - Benefits of this are typically minor

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= $\frac{1}{\#Keys(R.age)}$ Number of distinct values
  - Estimated #results = $\frac{\#Records(R)}{\#Keys(R.age)}$
  - 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

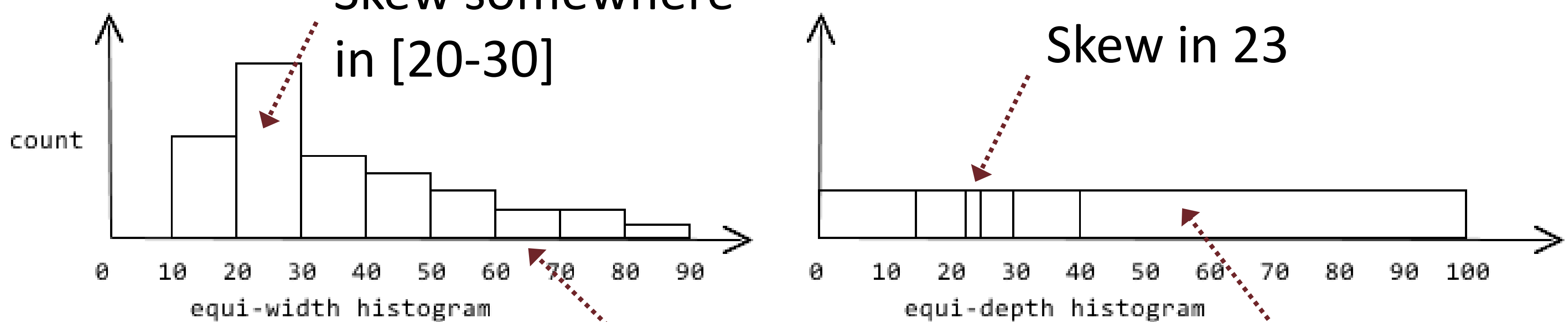
Estimating the number of (intermediary) results

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

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

Skew somewhere in [20-30]

Skew in 23

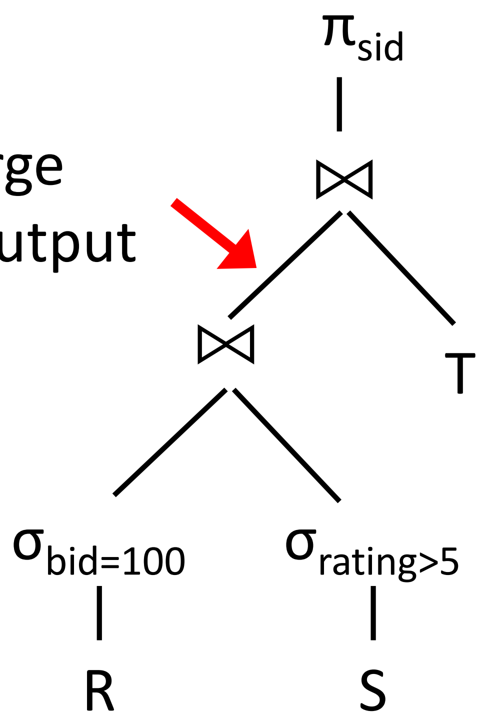


Accurate in [40-100]

Very inaccurate in [40-100]

Join cardinality estimates

How large
is the output



- Required for cost estimation of each join
- Reorder joins so that records are filtered as fast as possible!

Join cardinality estimates

- Reduction factor $\frac{1}{\max[\#Keys(R.sid), \#Keys(S.sid)]}$

If unknown,
use 10!

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

- More for cost estimation & statistics in the book. Chapters:
  - "Evaluation of rel. operators" (CS-322 material)
  - "Introduction to query optimization"
  - "A typical relational query optimizer"

Now back to

Selinger’s work

and

SYSTEM R

SYSTEM R Optimizer

- IBM SYSTEM R
  - Seminal project from the 70s
  - Drastic influence on succeeding dbs!

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?

SYSTEM R Optimizer

- ***Retrieve the names of people 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

ARTIST

ID	NAME
Integer, PK	VARCHAR(32)

ALBUM

ID	NAME
Integer, PK	VARCHAR(32) UNIQUE

ARIST, APPEARS: No index, no predicate in the WHERE clause that could use an index

ALBUM: Index would help (assume it exists)

SYSTEM R Optimizer

- ***Retrieve the names of people 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 people 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

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

All possible join orders

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

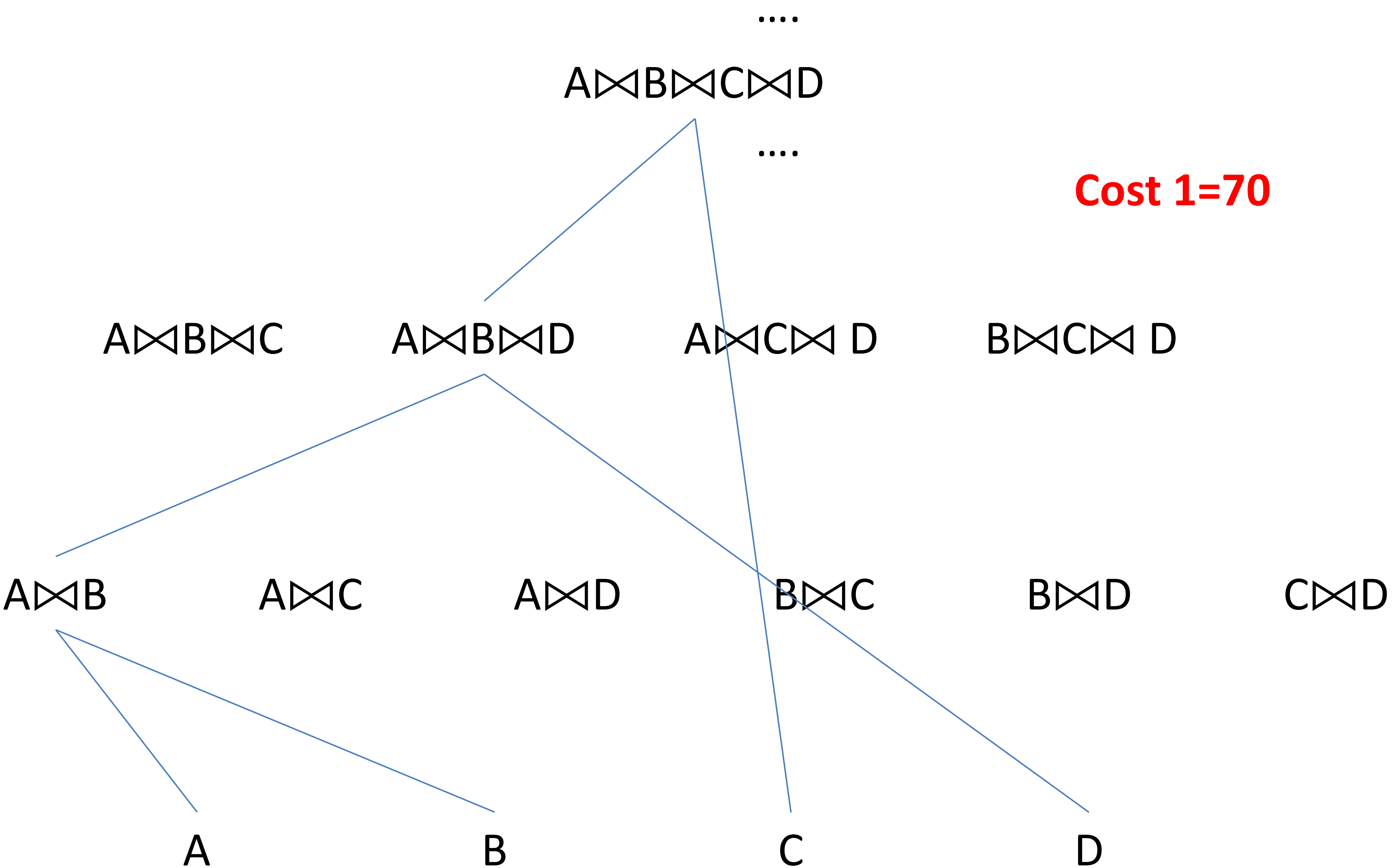
Join ordering

Scales BADLY with #joins N

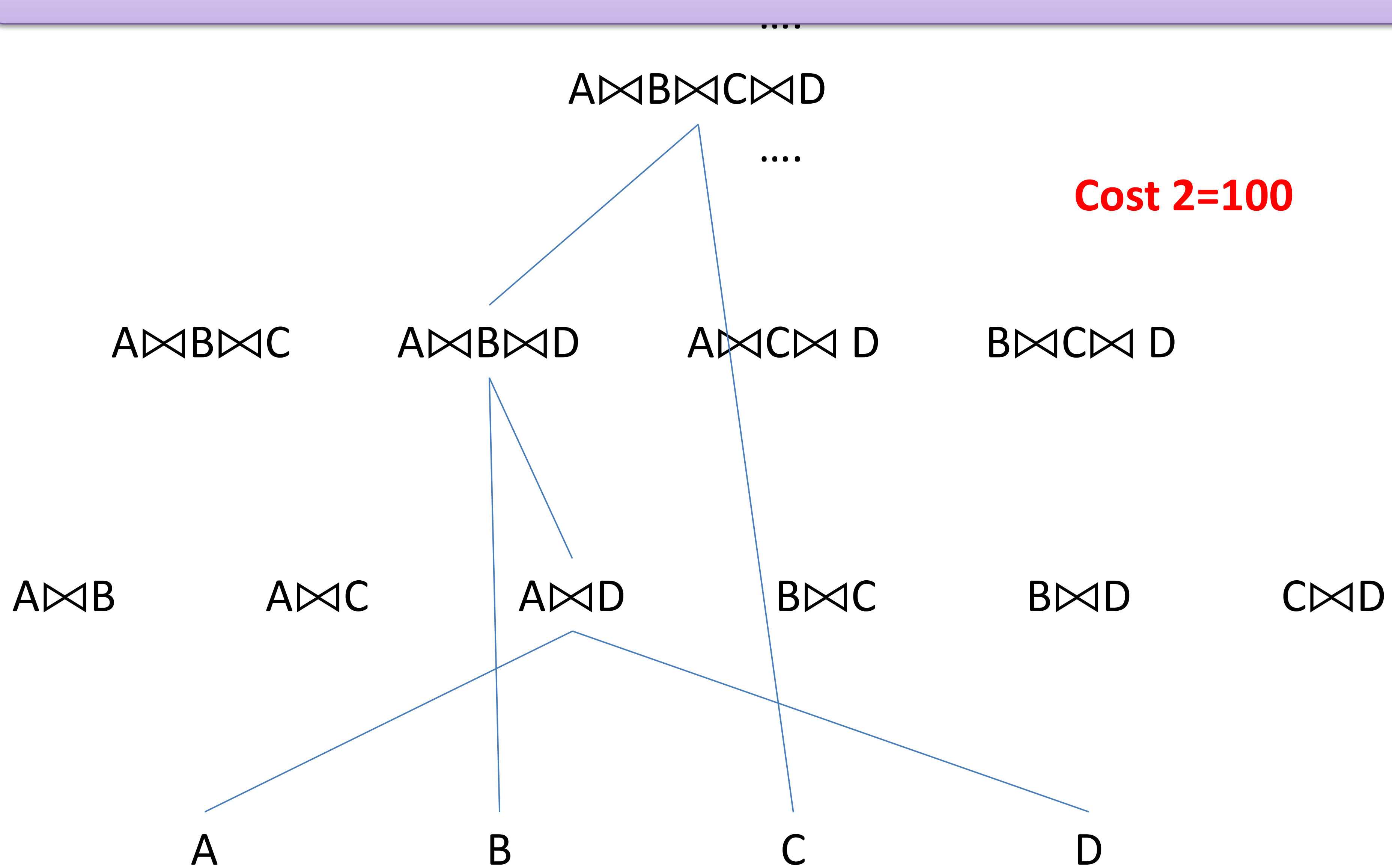
- Naïve: Just try all possible orders
  - $N * (N-1) * (N-2) \ldots * 3 * 2 * 1 = N!$
- Principle of optimality: The optimal plan for k joins is produced by extending the optimal plan(s) for k-1 joins!
  - To find optimal order of A ⋈ B ⋈ C ⋈ D, reuse partial solutions for optimal order of A ⋈ B ⋈ C, A ⋈ B ⋈ D, A ⋈ C ⋈ D, and B ⋈ C ⋈ D.
  - Dynamic programming → $O(N \times 2^{N-1})$
  - N=10 → 5120 Vs 3.6 Millions
- **Assume** principle of optimality!

Still expensive but **feasible**

Principle of optimality – Getting A⋈B⋈D



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



Back to our running example…

HashJoin cost=10
SM Join cost=20
NL Join=100

ARTIST⋈APPEARS
ALBUM

ARTIST
APPEARS
ALBUM

ARTIST⋈APPEARS⋈ALBUM

HashJoin cost=10
SM Join cost=20
NL Join=100

ALBUM⋈APPEARS
ARTIST

…

```
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…

HashJoin cost=10
~~SM Join cost=20~~
~~NL Join=100~~

ARTIST⋈APPEARS
ALBUM

ARTIST
APPEARS
ALBUM

ARTIST⋈APPEARS⋈ALBUM

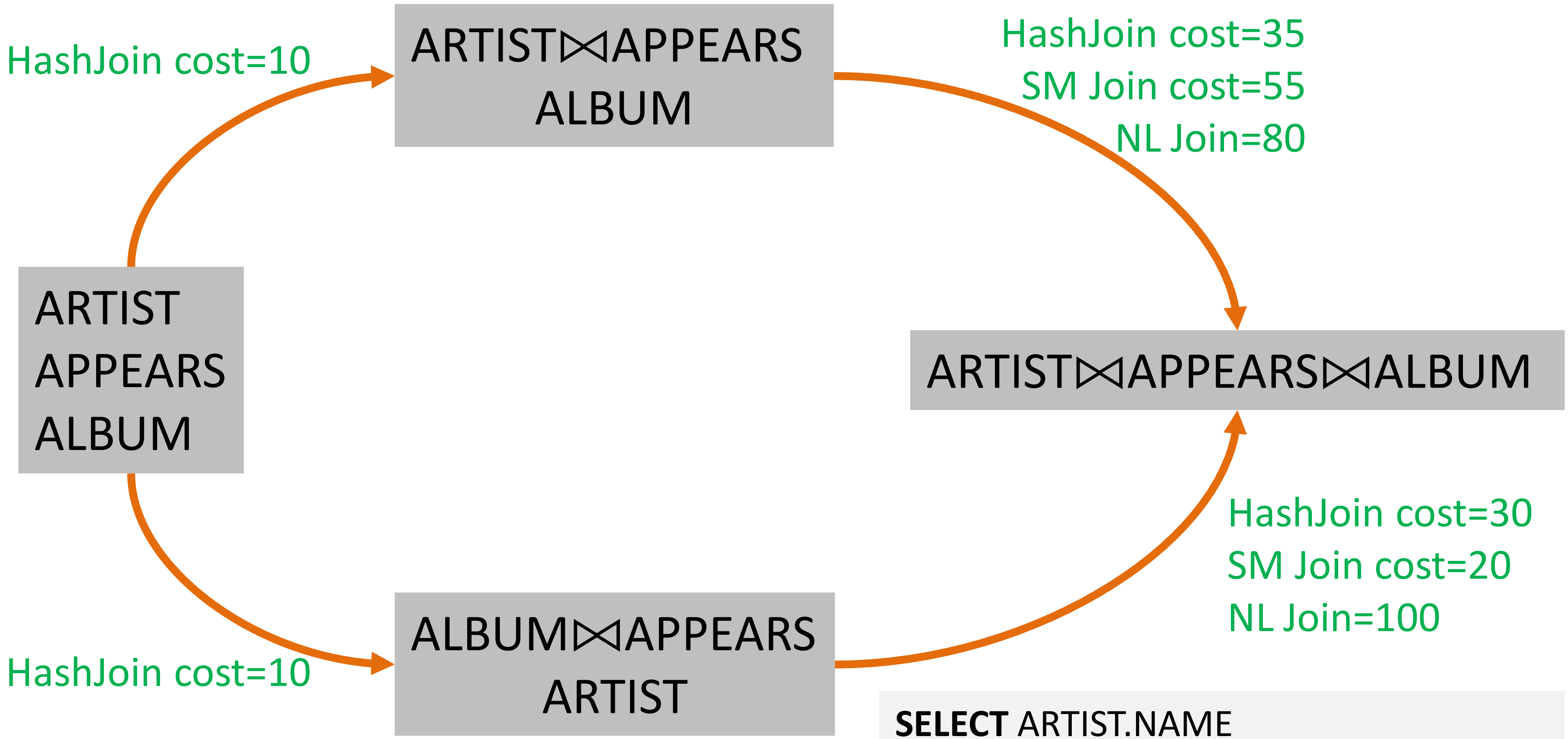
ALBUM⋈APPEARS
ARTIST

HashJoin cost=10
~~SM Join cost=20~~
~~NL Join=100~~

…

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

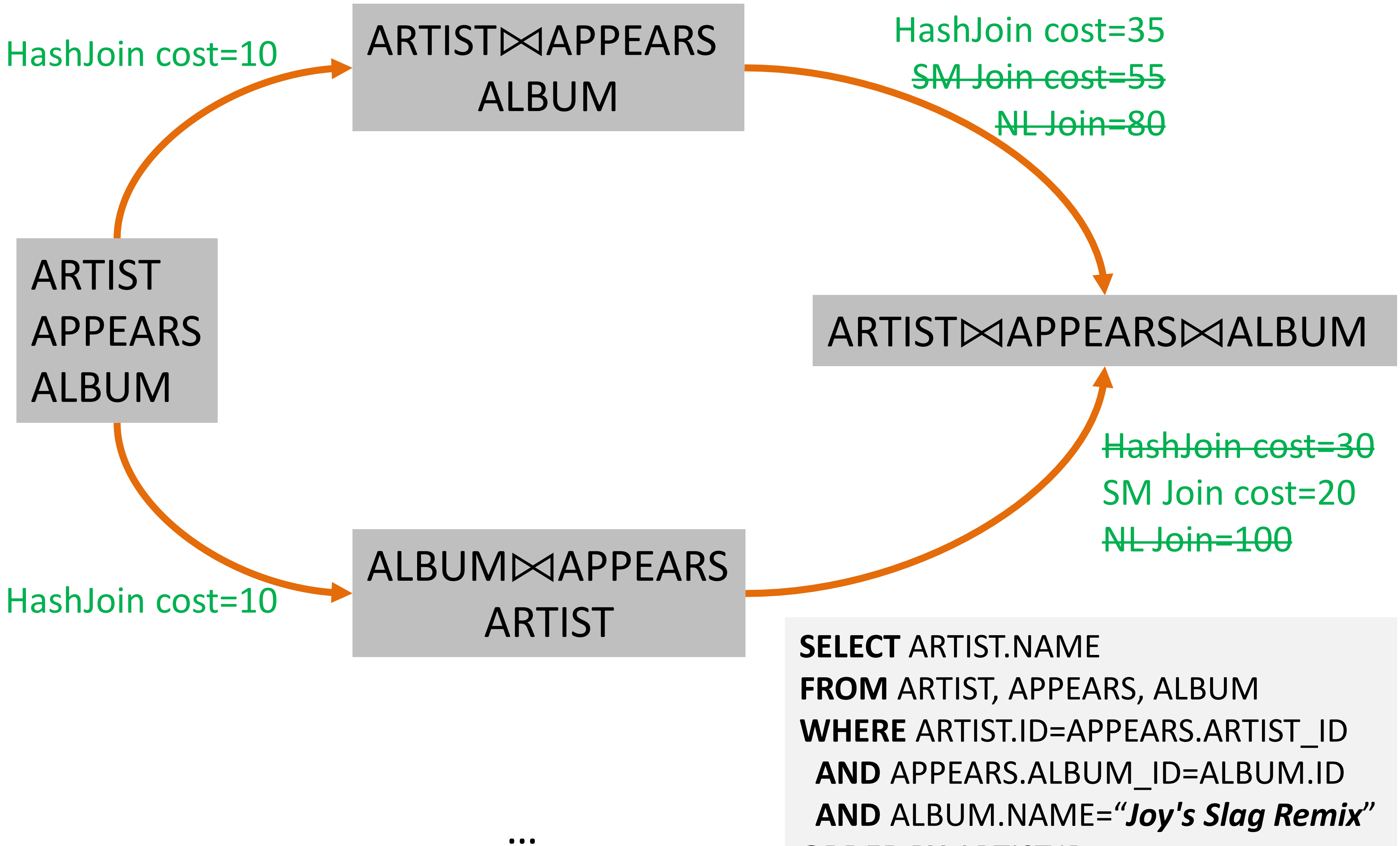
Back to our running example…



…

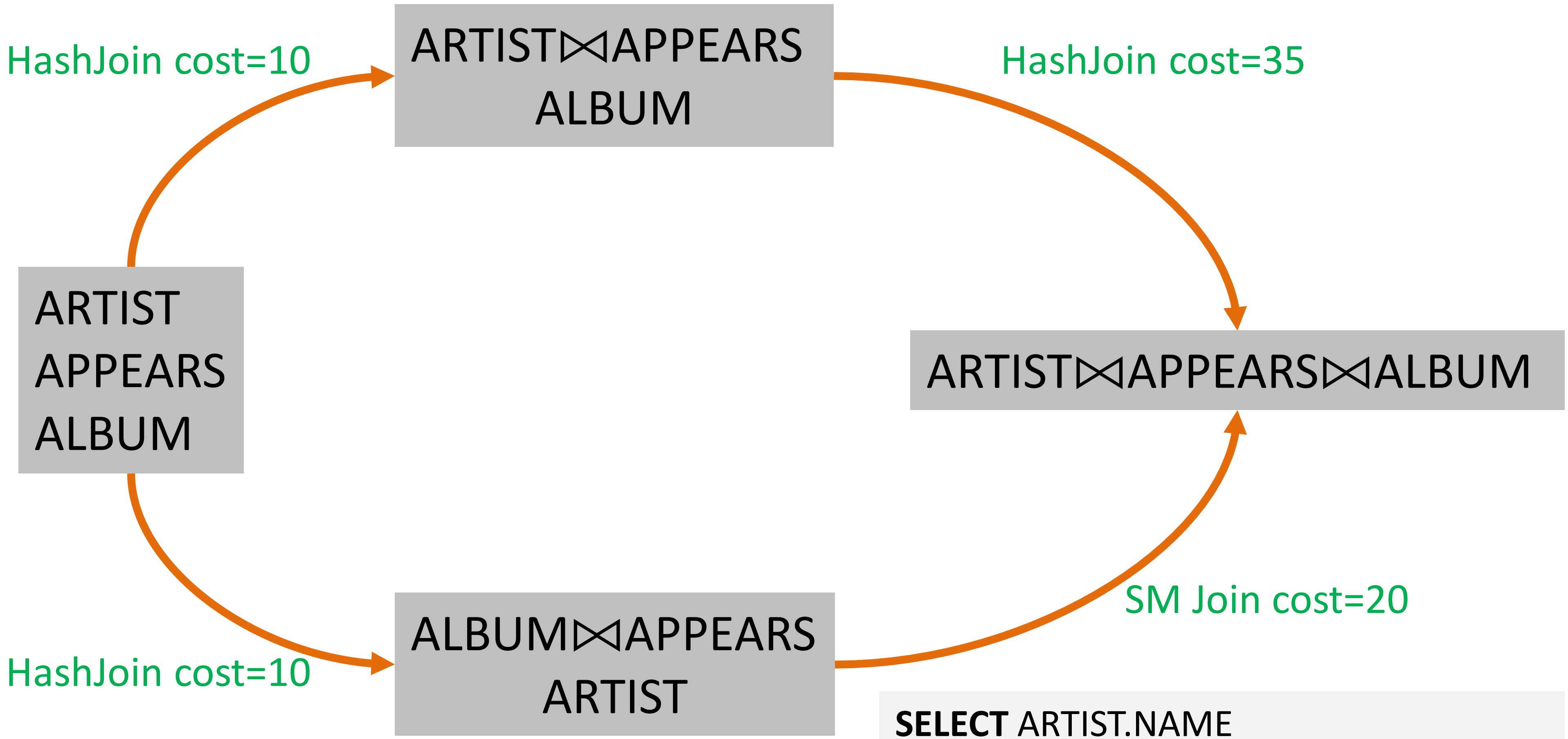
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
  AND APPEARS.ALBUM_ID=ALBUM.ID
  AND ALBUM.NAME=“***Joy's Slag Remix***”
ORDER BY ARTIST.ID

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
```

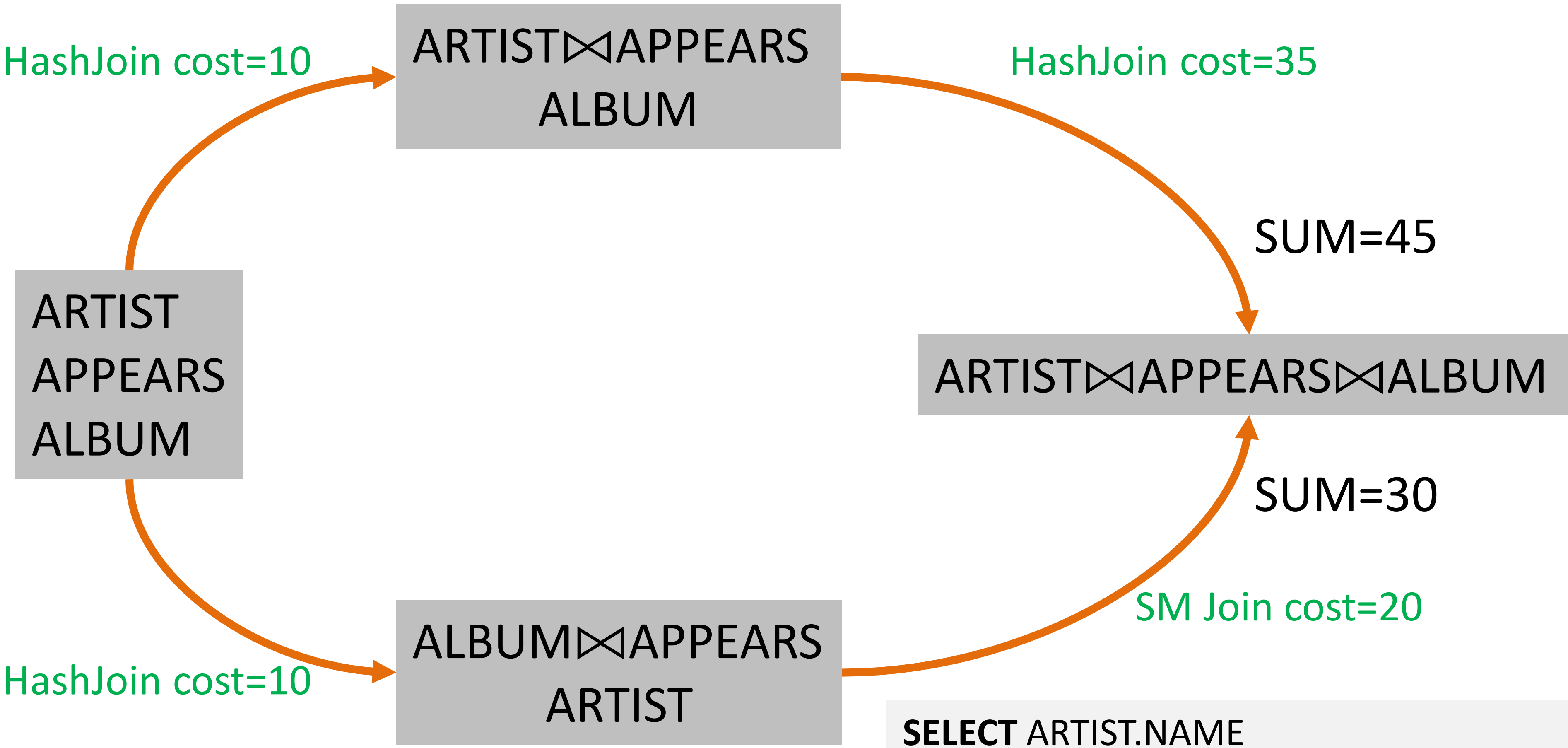
Back to our running example…



…

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

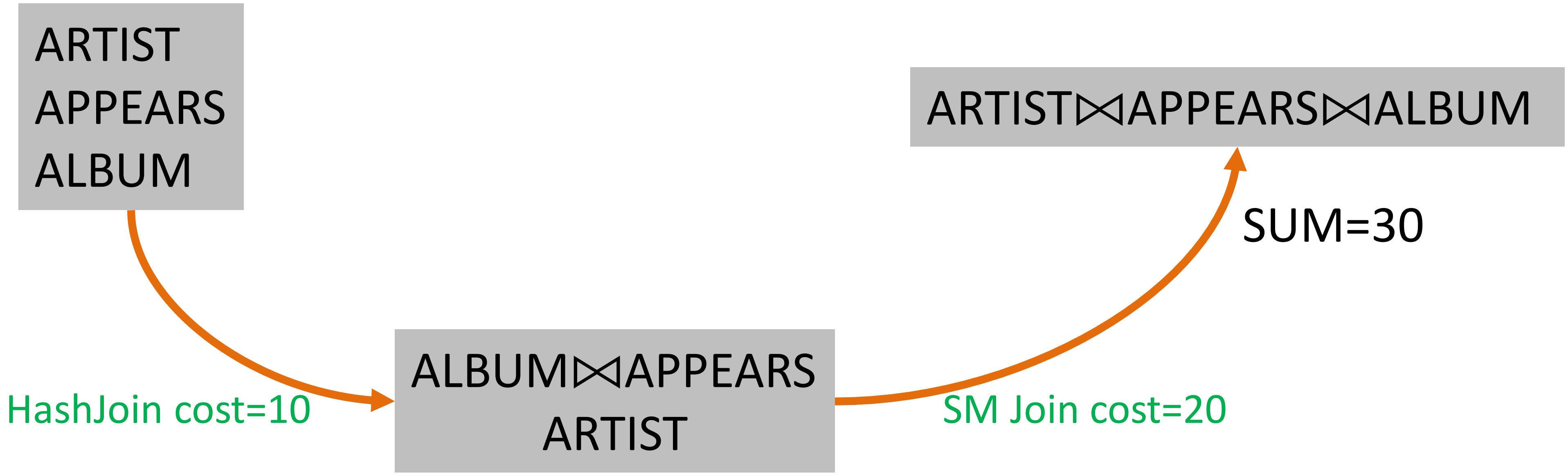
Back to our running example…




```sql
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 independence of distributions!
- 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


OTHERSIDE
RED HOT CHILI PEPPERS

Is *principle of optimality* correct?

- 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

- **Weaker principle of optimality**
  - Consider order! A plan is compared with all other plans that produce the same order
  - Does this ring a bell?

Interesting orders

- Attribute has interesting order if it participates in a join predicate, and:
  - it occurs in the Group By clause
  - it occurs in the Order By clause

  Will cause an additional sorting

- Choose plans that produce the correct order, e.g.,
  - Sort-merge join instead of NL join
  - Decide on the outer/inner table order

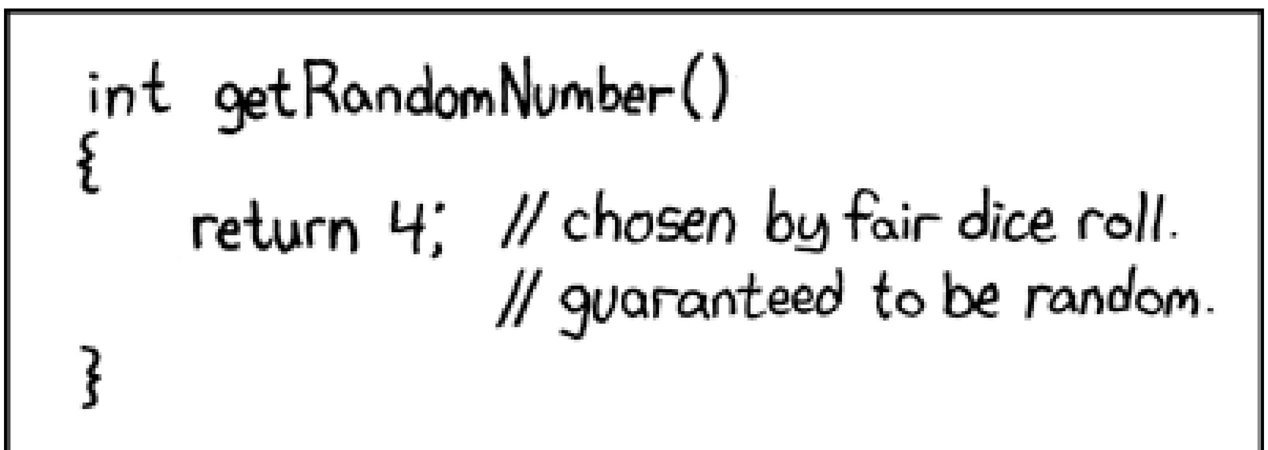
- Later generalized to include any *physical properties* not common across the plans!

Selectivity estimates, revisited

What if there is no index, and no histogram?

- Cannot estimate $\#\text{Keys}(\text{R.age})$
- When everything else fails, revert to magic
  - Set $\#Keys(R.age) = 10$

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



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

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.

→ Does not consider complex operator interdependencies & resource usage.

→ Space exploration **only** considers joins.

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

Additional optimization opportunities

- Optimization Granularity
- Optimization Timing
- Plan Stability

Optimization granularity

- **Choice #1: Single Query**
  → Much smaller search space.
  → DBMS cannot reuse results across queries.
  → In order to account for resource contention, the cost model must account for what is currently running.
- **Choice #2: Multiple Queries**
  → More efficient if there are many similar queries.
  → Search space is much larger.
  → Useful for scan sharing.

Optimization timing

- **Choice #1: Static Optimization**
  → Select the best plan prior to execution.
  → Plan quality is dependent on cost model accuracy.
  → Can amortize over executions with prepared stmts.
- **Choice #2: Dynamic Optimization**
  → Select operator plans on-the-fly as queries execute.
  → Will have to reoptimize for multiple executions.
  → Difficult to implement/debug (non-deterministic)
- **Choice #3: Hybrid Optimization**
  → Compile using a static algorithm.
  → If the error in estimate > threshold, reoptimize

Plan stability

- **Choice #1: Hints**
  → Allow the DBA to provide hints to the optimizer.

- **Choice #2: Fixed Optimizer Versions**
  → Set the optimizer version number and migrate queries one-by-one to the new optimizer.

- **Choice #3: Backwards-Compatible Plans**
  → Save query plan from old version and provide it to the new DBMS.

Fixed Optimizer Versions

Question: I just upgraded to Oracle 10g and I'm seeing very bad SQL performance. I had to set *optimizer_features_enable=9.0.5.* What can I do to fix Oracle10g upgrade & migration performance tuning problems?

Answer: Oracle has improved the cost-based Oracle optimizer in 9.0.5 and again in 10g, so you need to take a close look at your environmental parameter settings (init.ora parms) and your optimizer statistics. I have complete directions in my book "Oracle Tuning - The Definitive Reference", but here are some notes. See also Oracle tips for 10g migration. and Oracle 11g upgrade performance problems.

ORACLE®

It happens to the best of us!

Optimization for multiple queries

- Running multiple OLAP queries concurrently can saturate the I/O bandwidth (disk, memory)
  - If they access different parts of a table at the same time, buffer manager won't help.

- Solution: SCAN SHARING
  - Explicitly changing the loading order of pages!

Scan sharing

- Queries are able to reuse data retrieved from storage or operator computations.

- Allow multiple queries to attach themselves to a single cursor that scans a table.
  → Queries do not have to be exactly the same.
  → Can also share intermediate results.

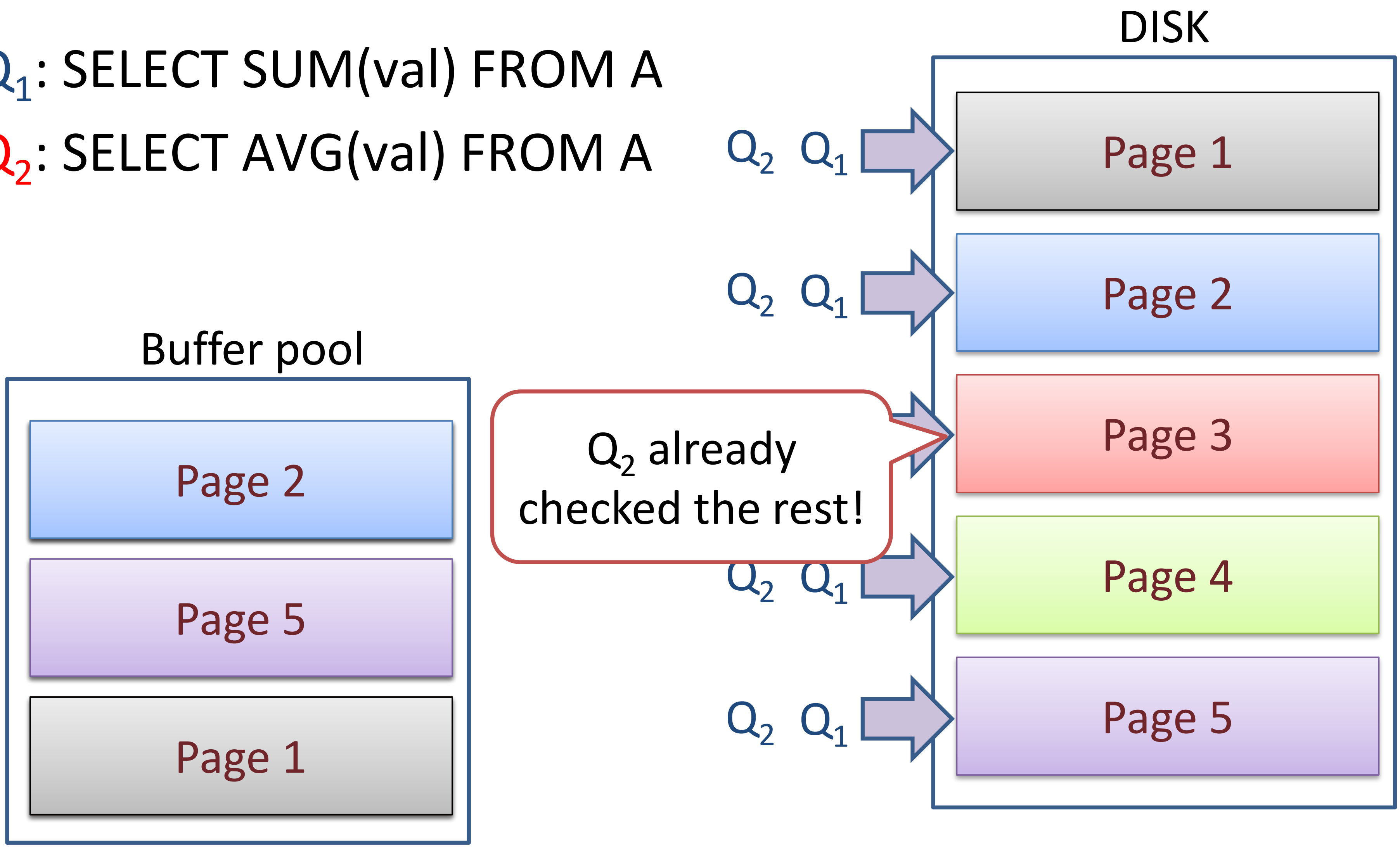
Disk-based scan sharing

- Q_2 needs to perform a scan, Q_1 already doing it → DBMS will attach the Q_2 to Q_1's scan cursor.
  → The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure.
- Fully supported in IBM DB2 and Microsoft SQL. Oracle only supports ***cursor sharing*** for identical queries.

Disk-based scan sharing

Q_1: SELECT SUM(val) FROM A

Q_2: SELECT AVG(val) FROM A



Disk-based scan sharing

- Drastically reduces I/O when queries can share cursor
  - Queries share cursor when they have the same access path
- Can a nested-loop join and a selection share a cursor?
  - What if the common table is the outer table of the join?

In-memory scan sharing

- Nuanced: Limited cache space (&convoy phenomenon)
- In practice: IBM BLINK -- Query accelerator for IBM DB2 with support for explicit scan sharing.
- Each worker thread will execute a separate table scan operation.
- A dedicated ***reader thread*** feeds blocks of tuples to the worker threads.
  → Load a block of tuples in the cache and share it among multiple queries.
  → **All worker threads need to acknowledge they are finished with the current block before the reader can move on to the next one.**

Summary

- Query optimizer has a huge impact on DBMS performance, scalability, capabilities for OLAP queries
- Challenges
  - Cost estimation
  - Efficient space exploration (both physical & logical plans)
- Heuristics & cost-based optimization
  - Particularly important for joins!
- Multi-query optimization