Query Optimization

CSE 532 · Database Systems

November 2, 2026

Selinger, Astrahan, Chamberlin, Lorie, and Price

Access Path Selection in a Relational Database Management System · SIGMOD 1979

The Denver clerks query

```
SELECT NAME, TITLE, SAL, DNAME
FROM   EMP, DEPT, JOB
WHERE  TITLE = 'CLERK'
  AND  LOC = 'DENVER'
  AND  EMP.DNO = DEPT.DNO
  AND  EMP.JOB = JOB.JOB
```

Name, salary, job title and department of every clerk in a Denver department.

SQL says what rows you want. It never says how to get them.

Figure 1’s query, verbatim. Three tables, two local predicates, two join predicates.

Four choices nobody made

The query fixes which rows come back. It says nothing about how to get them.

1. **Order:** which table do we read first, and in what order do we join the rest?
2. **Access:** do we scan a whole table, or go through an index? Which one?
3. **Join:** how do we join two tables?
4. **Sorting:** do we sort rows along the way, or never?

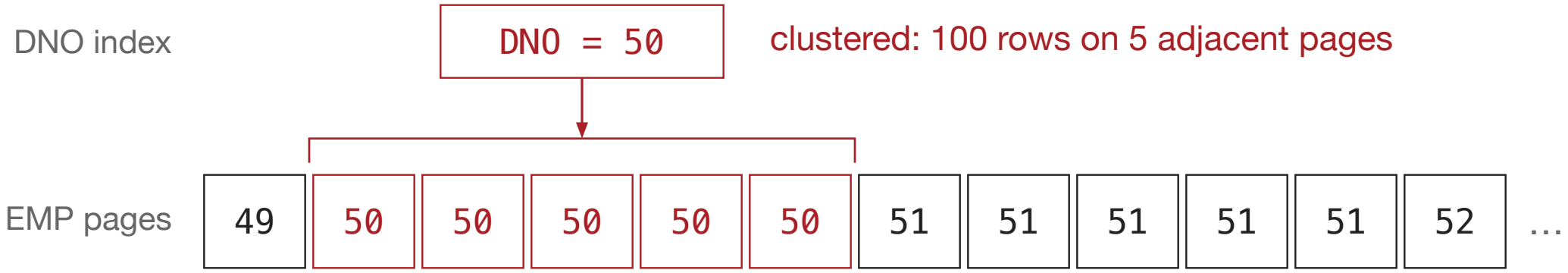
Same rows every way. Which way is cheapest, and can we find it fast?

The three tables

EMP (NAME, DNO, JOB, SAL)		10,000 rows, stored in DNO order	**500 pages**
index on DNO	clustered	100 values, 100 rows each	50 pages
index on JOB	non-clustered	50 values, 200 rows each	50 pages
index on SAL	non-clustered	values 5,000 to 25,000	40 pages
DEPT (DNO, DNAME, LOC)		100 rows, stored in DNO order	**5 pages**
index on DNO	clustered, unique	100 values, 1 row each	2 pages
index on LOC	non-clustered	10 cities, 10 rows each	2 pages
JOB (JOB, TITLE)		50 rows, stored in JOB order	**2 pages**
index on JOB	clustered, unique	50 values, 1 row each	1 page
index on TITLE	non-clustered, unique	50 titles, 1 row each	1 page

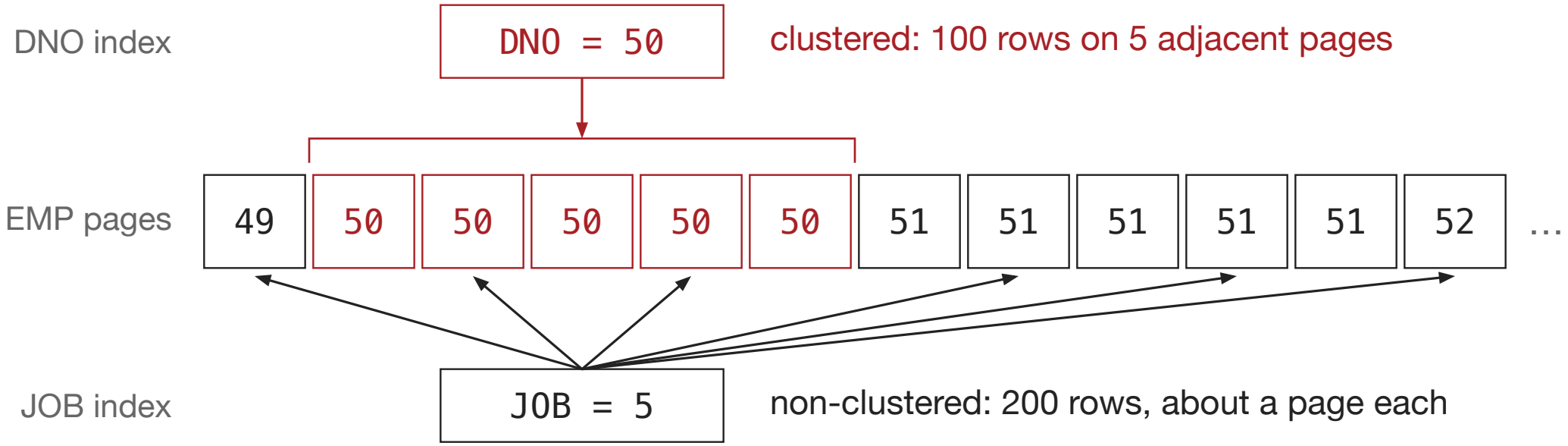
Stored in DNO order

EMP's rows were inserted sorted by DNO and kept that way: a choice made at setup.



Stored in DNO order

EMP's rows were inserted sorted by DNO and kept that way: a choice made at setup.



A table is stored in one order, so at most one index is clustered.

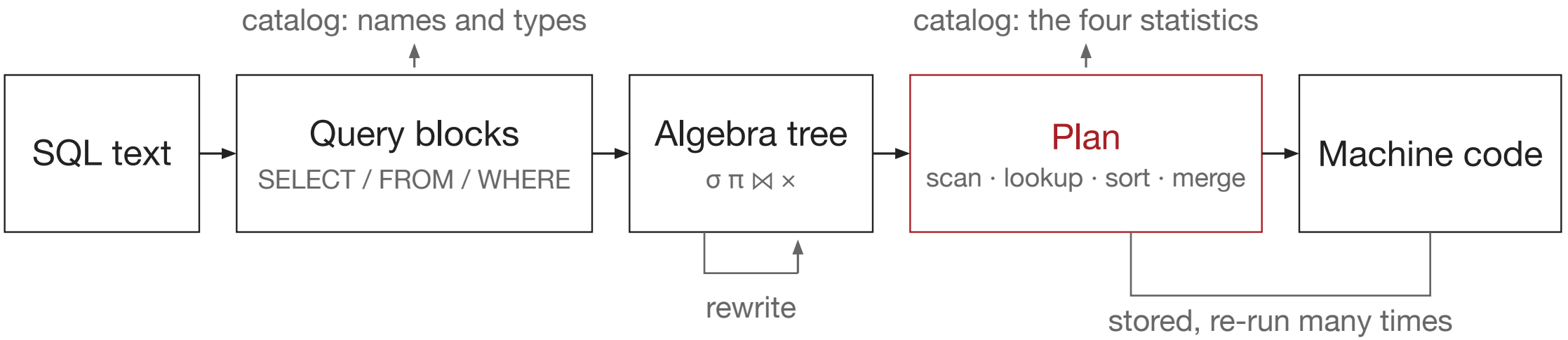
An index lists rows in its own column's order, and is small: 40 to 50 pages against EMP's 500.

The answer to our query: 200 clerks, 1 in 10 of them in a Denver department, about 20 rows.

From text to a tree

Where the optimizer sits

Parse into **query blocks** (one SELECT, FROM and WHERE each), check them against the **catalog**, rewrite the tree, choose a plan, generate code, store it.



System R compiled plans to machine code. Postgres interprets. Either way: optimize once, run many times.

Four signs, read bottom-up

Sign	Reads as	In our query
σ	keep the rows where	σ LOC = 'DENVER' on DEPT: 10 rows of 100

Four signs, read bottom-up

Sign	Reads as	In our query
σ	keep the rows where	σ LOC = 'DENVER' on DEPT: 10 rows of 100
π	keep only these columns	π NAME, TITLE, SAL, DNAME, at the root

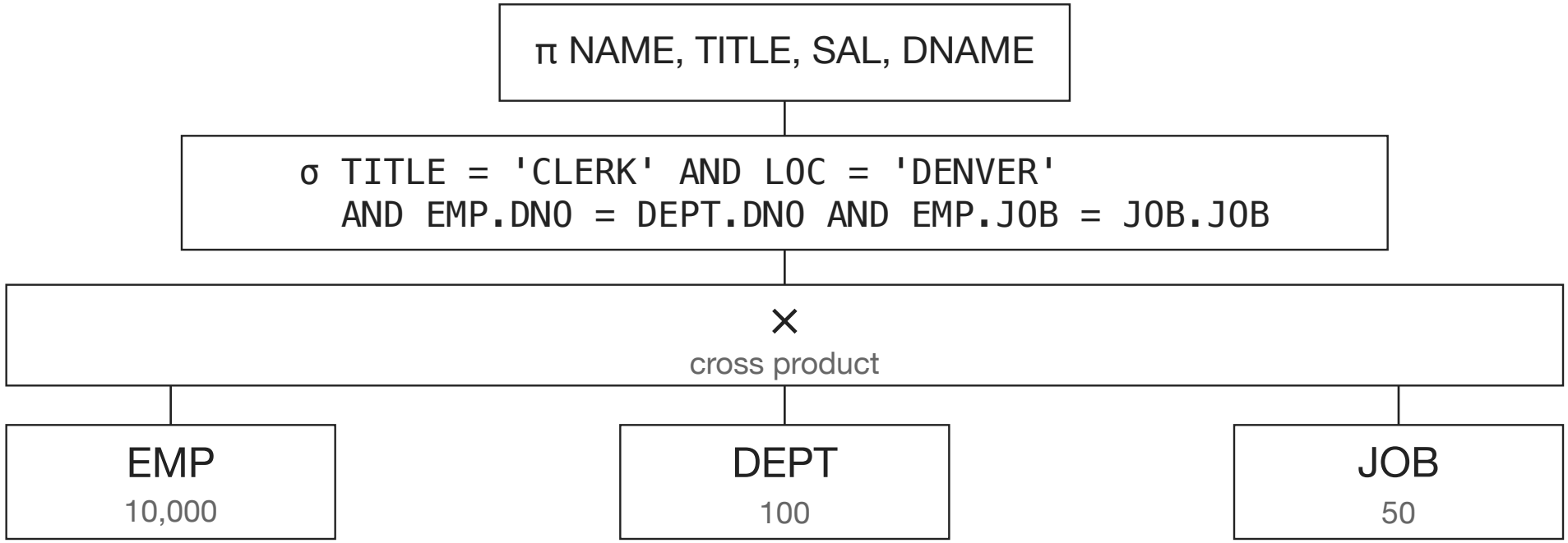
Four signs, read bottom-up

Sign	Reads as	In our query
σ	keep the rows where	σ LOC = 'DENVER' on DEPT: 10 rows of 100
π	keep only these columns	π NAME, TITLE, SAL, DNAME, at the root
×	every row with every row	EMP × DEPT: 10,000 × 100 = 1,000,000 rows

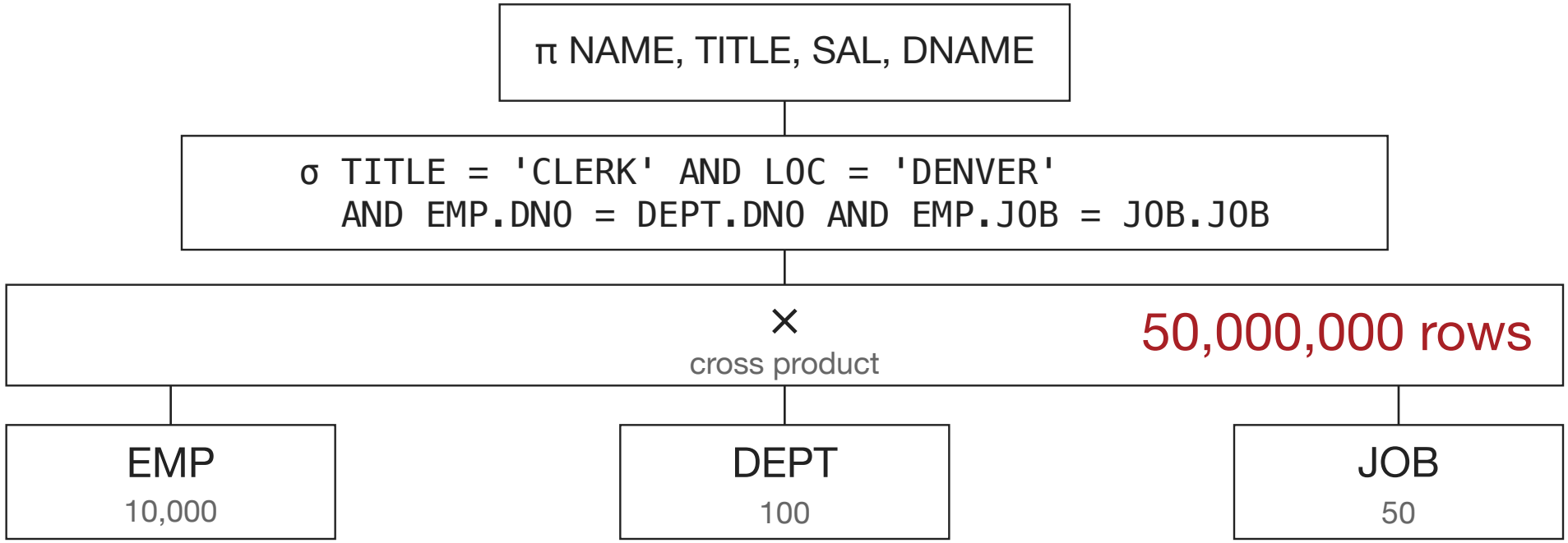
Four signs, read bottom-up

Sign	Reads as	In our query
σ	keep the rows where	σ LOC = 'DENVER' on DEPT: 10 rows of 100
π	keep only these columns	π NAME, TITLE, SAL, DNAME, at the root
×	every row with every row	EMP × DEPT: 10,000 × 100 = 1,000,000 rows
⋈	a × with a σ on the join column	EMP ⋈ DEPT on DNO: 10,000 of those

Strawman 1: run the tree as parsed



Strawman 1: run the tree as parsed

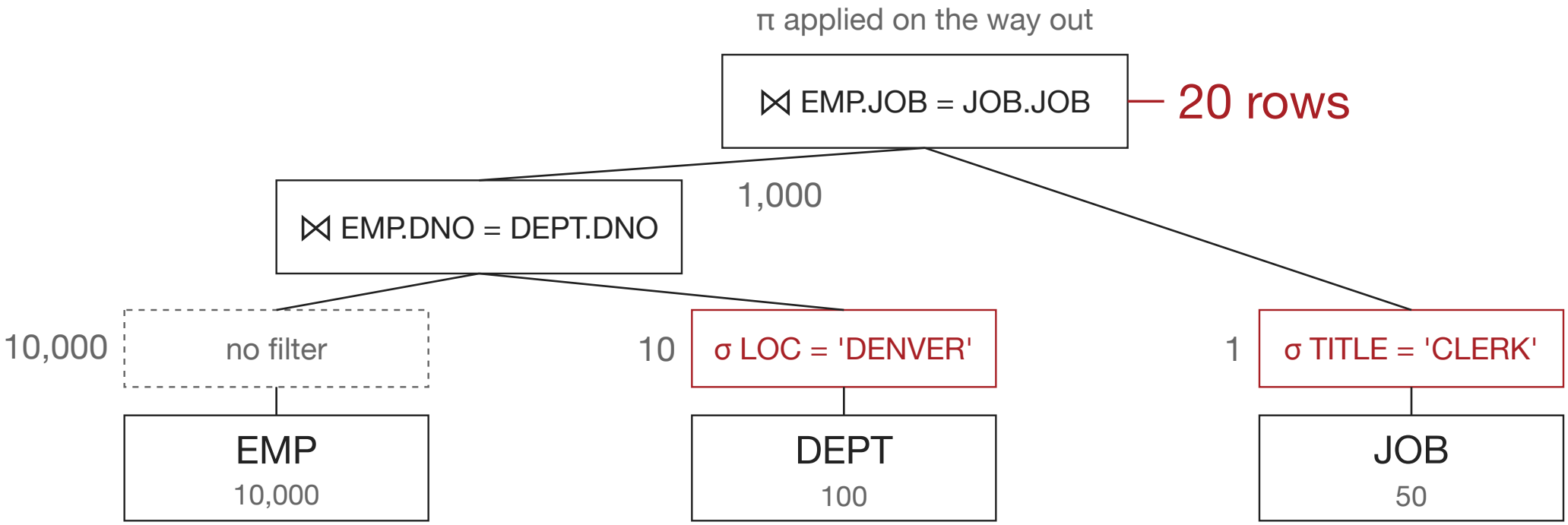


50,000,000 rows built and thrown away to return 20.

50,000,000 rows at 20 per page is 2,500,000 pages, written once and read back: about 5,000,000 disk page reads and writes, by our arithmetic.

Pushdown turns × into ⋈

Pushdown moves each filter onto the table it names; the AND-ed parts are **boolean factors**.



Strawman 2: fixed rules

Two rules that sound right

Replace the search with a rule book. Each rule is right on plenty of queries.

	Rule	Why it sounds right
1	Always use an index if one exists	an index skips pages
2	Always join in FROM order	it needs no search

Rule 1: where the index wins

```
SELECT NAME FROM EMP WHERE DNO = 50
SELECT NAME FROM EMP WHERE JOB = 5
```

Rule 1: where the index wins

```
SELECT NAME FROM EMP WHERE DNO = 50
SELECT NAME FROM EMP WHERE JOB = 5
```

WHERE	Rows the index finds	Index	Scan
DNO = 50	100, together: clustered	**about 5**	500
JOB = 5	200, scattered: non-clustered	**about 200**	500

Few rows, or rows that sit together: the index wins.

One department’s 100 employees sit on 5 pages, 20 to a read. One job’s 200 employees are scattered, about a read each, and 200 reads still beat 500.

Rule 1: where it breaks

```
SELECT NAME FROM EMP WHERE SAL > 20000
```

Rule 1: where it breaks

```
SELECT NAME FROM EMP WHERE SAL > 20000
```

Path	What it reads	Reads
SAL index	a quarter of EMP, 2,500 rows scattered across the table: a read each	about 2,500
Scan	every page once, testing SAL on the way past	**500**

The index pays per row it finds, the scan per page.

The scan wins 5×. Past about 500 scattered rows, as many as EMP has pages, scan.

Rule 2: where FROM order is right

People tend to name first the table they are filtering. This query does.

```
SELECT DNAME, NAME
FROM   DEPT, EMP
WHERE  DEPT.LOC = 'DENVER' AND EMP.DNO = DEPT.DNO
```

Rule 2: where FROM order is right

People tend to name first the table they are filtering. This query does.

```
SELECT DNAME, NAME
FROM   DEPT, EMP
WHERE  DEPT.LOC = 'DENVER' AND EMP.DNO = DEPT.DNO
```

Start from	It keeps	Then
DEPT, as written	10 Denver rows	**10 lookups** into EMP
EMP	all 10,000	10,000 lookups into DEPT

The order the user wrote is the best one, found with no search.

A **lookup** takes one row of the first table and finds its matches in the second.

Rule 2: where it breaks

```
SELECT DNAME, NAME
FROM   EMP, DEPT
WHERE  EMP.SAL > 20000 AND EMP.DNO = DEPT.DNO
```

Rule 2: where it breaks

```
SELECT DNAME, NAME
FROM   EMP, DEPT
WHERE  EMP.SAL > 20000 AND EMP.DNO = DEPT.DNO
```

Start from	It keeps	Then
EMP, as written	2,500, a quarter	2,500 lookups into DEPT
DEPT	all 100	**100 lookups** into EMP

Start from the table that keeps the fewest rows.

The FROM list does not know which. The running query says FROM EMP, DEPT, JOB: it starts with EMP, the one table its WHERE clause does not filter at all.

Measure, then choose

Applying filters early is the rule that survives: strawman 1's fix.

Strawman 2's rules each held on one query and broke on the next: only the data says which.

1. **A cost measure:** one number per plan, in units we can count.
2. **Statistics:** enough about the data to predict that number.
3. **A search:** through the candidates, without enumerating them all.

Estimate the cost of each candidate. Pick the cheapest.

Cost and statistics

What the optimizer minimizes

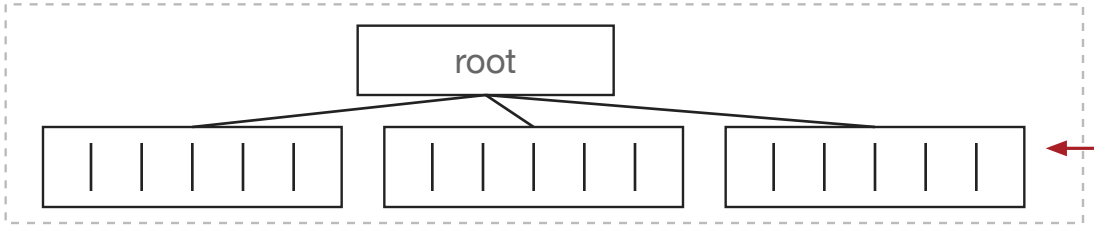
$$\texttt{COST} \approx \texttt{PAGE FETCHES}$$

A **page fetch** is one page read from disk, a data page or an index page.

- **Disk is slow:** a page read takes far longer than the CPU work on its rows.
- **Disk moves whole pages:** one row costs as much as the page it sits on.
- **So count pages:** the plan that reads the fewest pages is the cheapest.

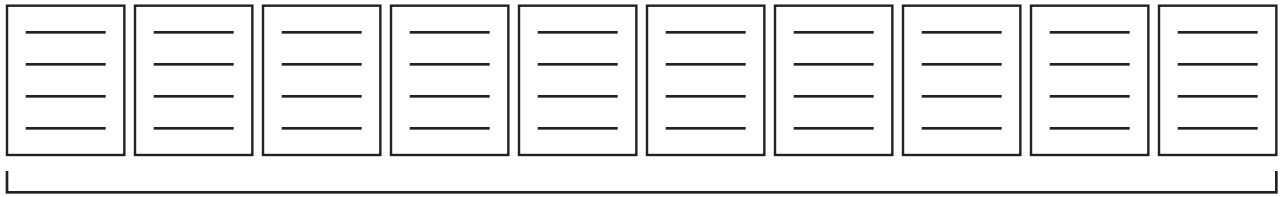
Four numbers in the catalog

index_pages: the index's own pages



distinct: distinct keys, kept per index,
so an un-indexed column has none

the table T: a scan reads all of its pages and nothing else



rows(T): rows of T

pages(T): the pages that hold them

Refreshed by a manual UPDATE STATISTICS, not on each write (a hot lock); Postgres ANALYZE likewise.

Our running statistics

T	rows	pages	Indexes (distinct, index_pages)
EMP	10,000	500	**DNO clustered** (100, 50) JOB (50, 50) · SAL (2,000, 40)
DEPT	100	5	**DNO clustered unique** (100, 2) LOC (10, 2)
JOB	50	2	**JOB clustered unique** (50, 1) TITLE unique (50, 1)

EMP: **20 rows per page** · SAL keys 5,000 to 25,000 · sorting Q rows: 2 × ceil(Q / 20)

Clustered = rows stored in that index's key order · **unique** = at most one row per key

Selectivity

Table 1: a factor per boolean factor

Predicate	F, the fraction of rows that pass
`column = value`	1 / distinct(column), else **1/10**
`column1 = column2`	1 / max(distinct1, distinct2), else **1/10**
`column > value`	(high – value) / (high – low), else **1/3**
`column BETWEEN a AND b`	(b – a) / (high – low), else **1/4**
`column IN (list)`	n × F(column = value), capped at 1/2

Bold: System R's guesses when there is no index, assuming a missing index means a small table.

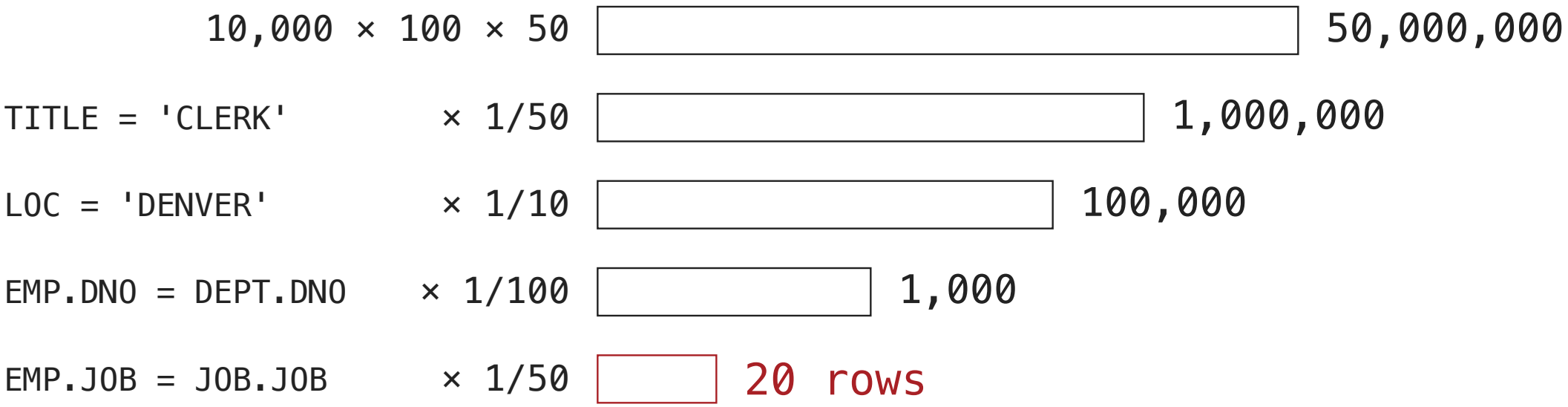
high, *low*: the index's largest and smallest key. AND: F1 × F2.

Our four selectivity factors

Boolean factor	Table 1 row used	F	Survivors
`TITLE = 'CLERK'`	`1 / distinct(TITLE) = 1/50`	`1/50`	1 of 50
`LOC = 'DENVER'`	`1 / distinct(LOC) = 1/10`	`1/10`	10 of 100
`EMP.DNO = DEPT.DNO`	`1 / max(100, 100)`	`1/100`	—
`EMP.JOB = JOB.JOB`	`1 / max(50, 50)`	`1/50`	—

Fifty million rows become twenty

result_rows = the block's answer size: the three tables' row counts times every factor's F.



result_rows = 50,000,000 / 2,500,000 = 20 rows.

Local factors first, then the two join factors. Bars on a log scale.

Read the assumptions

F(`LOC = 'DENVER'`) = 1/10 and F(`ZIP LIKE '802%'`) = 1/10. Multiply them?

DNO	LOC	ZIP
50	DENVER	80202
51	BOULDER	80301
52	DENVER	80216
57	DENVER	80237

Four of DEPT's 100 rows.

How many of the 100 pass both filters?

Read the assumptions

F(`LOC = 'DENVER'`) = 1/10 and F(`ZIP LIKE '802%'`) = 1/10. Multiply them?

DNO	LOC	ZIP
50	DENVER	80202
51	BOULDER	80301
52	DENVER	80216
57	DENVER	80237

Four of DEPT's 100 rows.

How many of the 100 pass both filters?

Independence: **1**

100 × 1/10 × 1/10

The data: **10**

every Denver department

One table, two tables, three

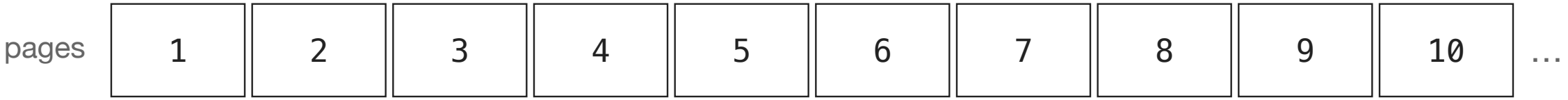
We can price a plan. Now choose one, building up a table at a time.

1. **One table:** what is the cheapest way to read it, a scan or an index?
2. **Two tables:** what is the cheapest way to join them?
3. **Three tables or more:** in what order should we join them?

One table: pick an access path

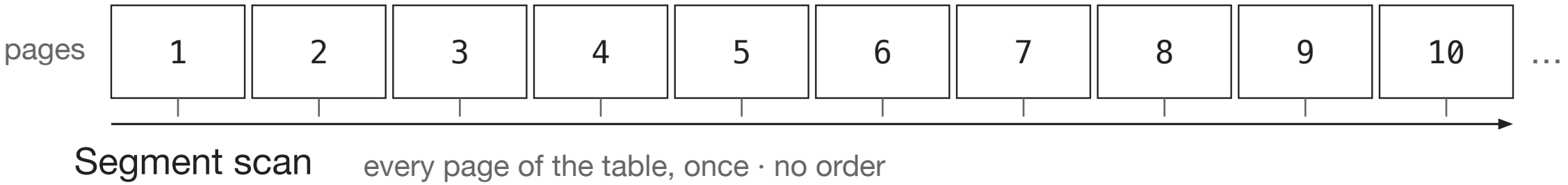
Two ways to read a table

Not the running query: rule 1's `SAL > 20000` on EMP, for this whole section.



Two ways to read a table

Not the running query: rule 1's `SAL > 20000` on EMP, for this whole section.



Segment scan — every page of the table, once · no order

Two ways to read a table

Not the running query: rule 1's `SAL > 20000` on EMP, for this whole section.

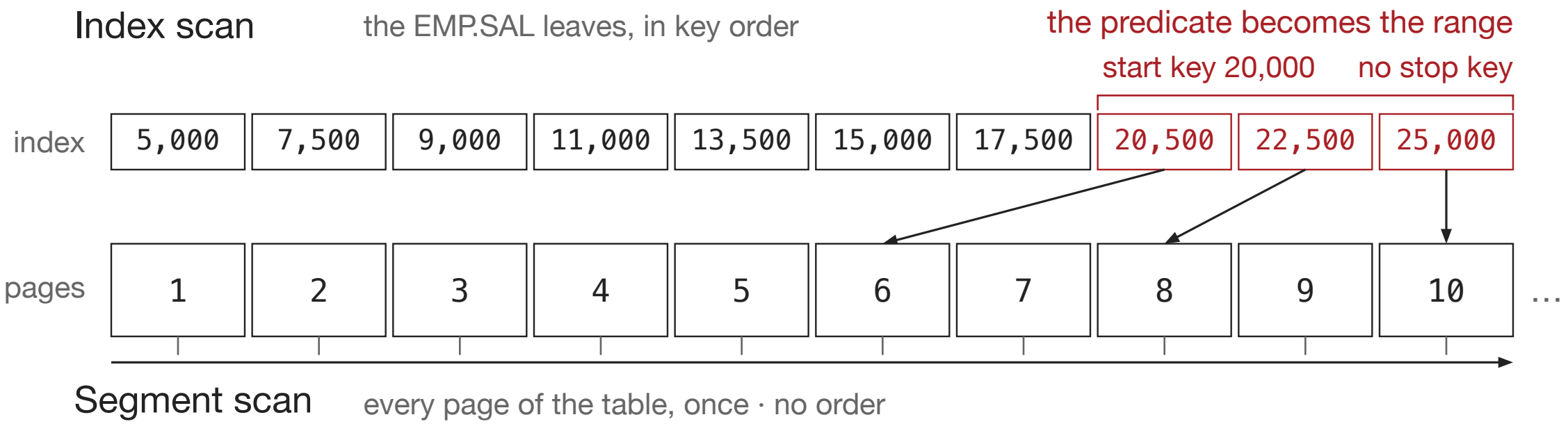


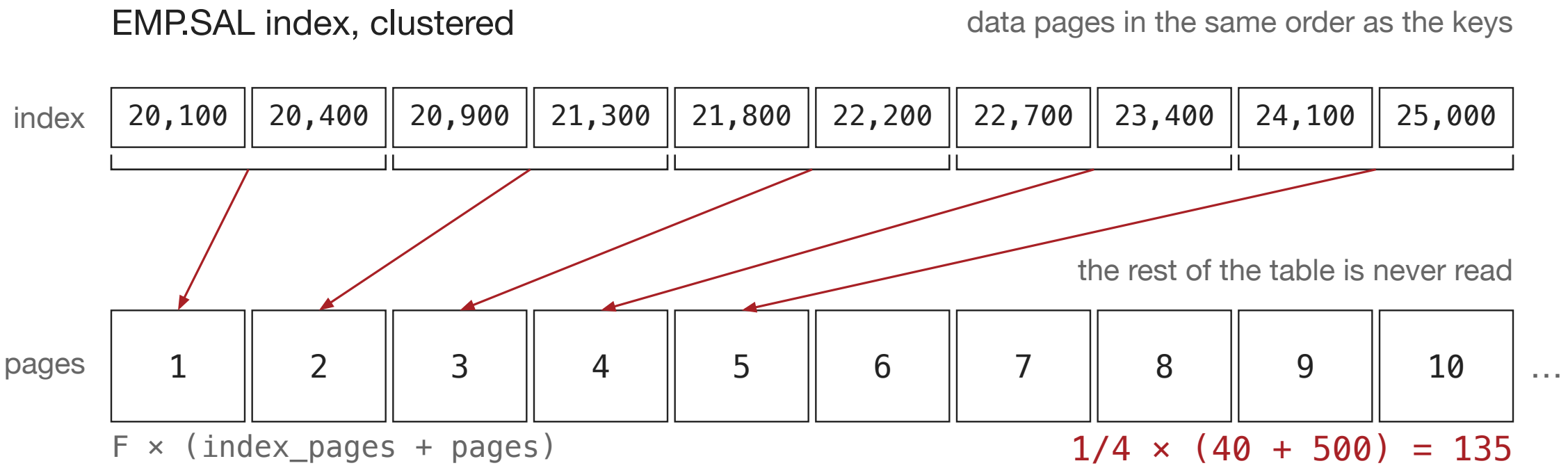
Table 2: six situations, eight formulas

Situation	Cost in pages
Unique index, equality	`1 + 1`
Clustered, matching	`F × (index_pages + pages)`
Non-clustered, matching	`F × (index_pages + rows)`
Clustered, matching nothing	`index_pages + pages`
Non-clustered, matching nothing	`index_pages + rows`
Segment scan	`pages`

Clustered = the table is stored in that index's key order, so a table has at most one.

The non-clustered rows have a second form, *pages* for *rows*, if the pages fit in the buffer.

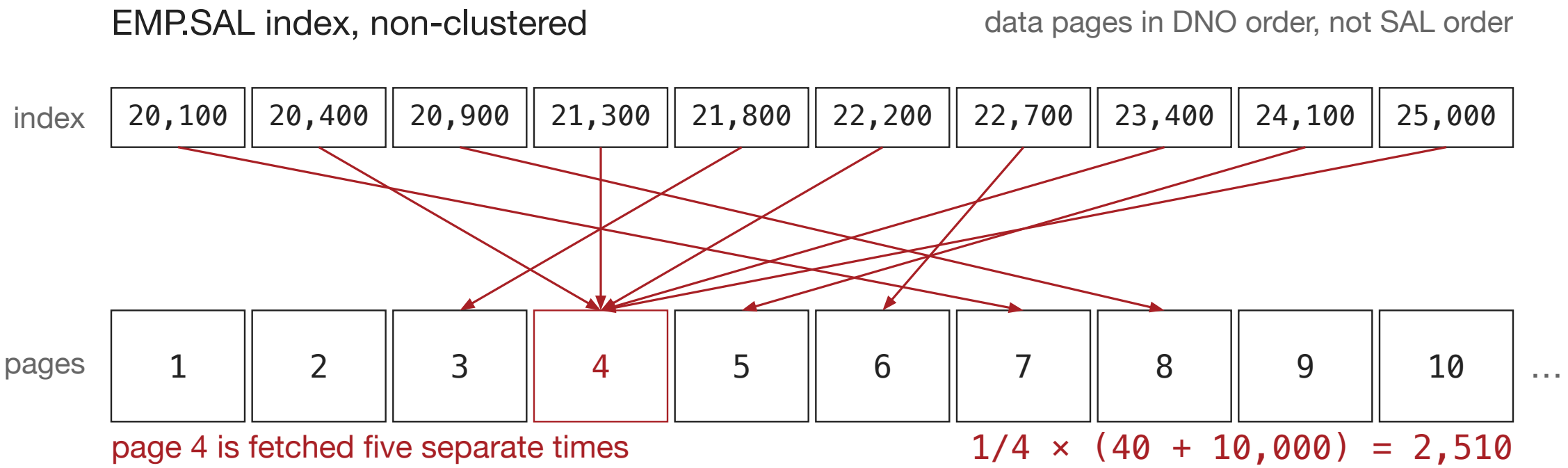
Clustered: one fetch per page



Were it clustered: 125 consecutive data pages + 10 index pages = 135.

EMP's rows sit in DNO order and only one index per table can be clustered, so this panel is a counterfactual. Layout: 20 rows per page, ten of 500 drawn.

Non-clustered: one fetch per row



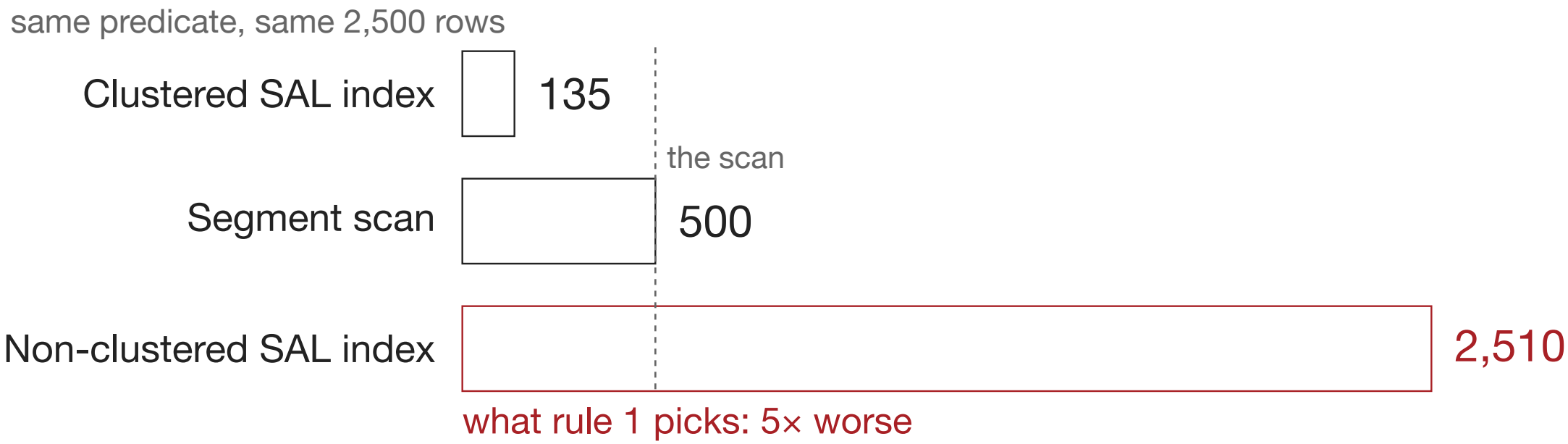
The same 2,500 rows, 2,500 scattered page visits: one fetch per row.

In the formula, *pages* became *rows*: strawman 2's "about a read per row".

Table 2 assumes EMP is far bigger than the buffer, so each visit to page 4 is a fresh fetch.

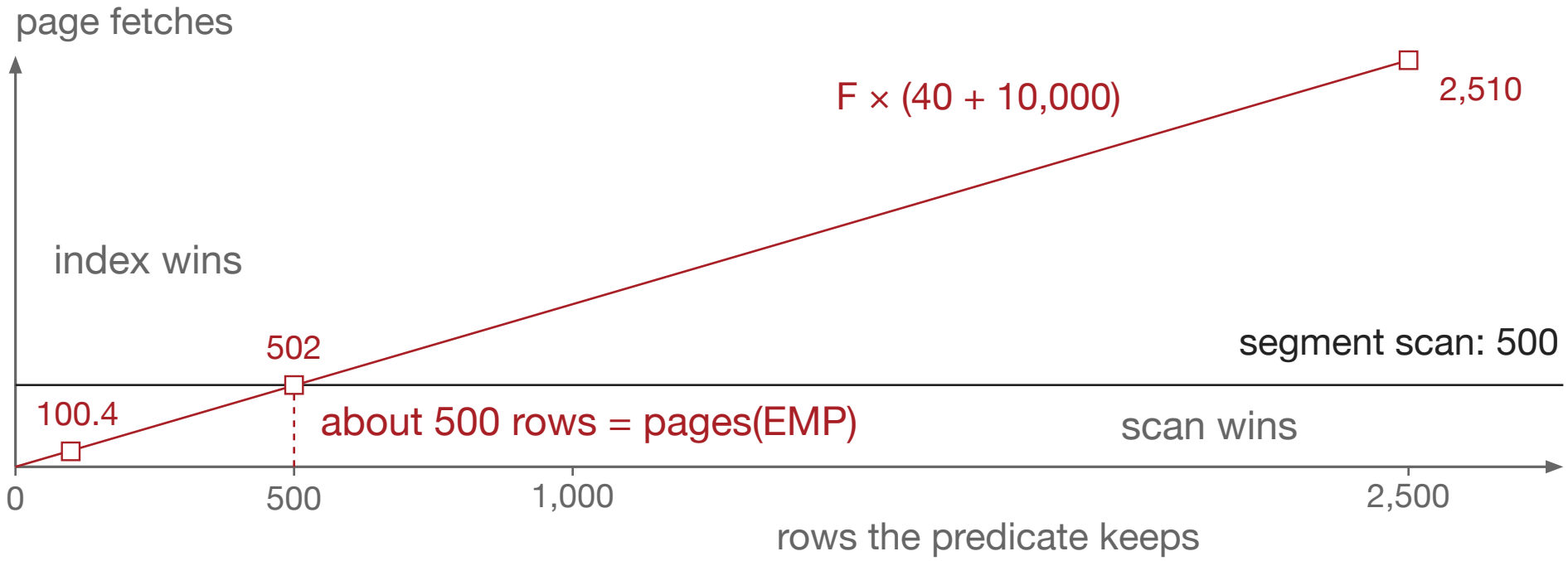
The same predicate, priced three ways

Rule 1's one-table query: SAL > 20000, F = 1/4, 2,500 rows out. Three Table 2 rows: the scan, EMP's real SAL index, and the clustered one it cannot have.



Break-even: five percent of the table

Vary F and the index cost scales with it: F × (40 + 10,000) against a flat 500.



The index loses once the qualifying rows outnumber the table's pages.

Break-even: F × (index_pages + rows) = pages, so F = 500 / 10,040, about 5% of the rows.

Two tables: join methods

A join, as two loops

```
for each d in DEPT:
    for each e in EMP:
        if e.DNO = d.DNO:
            emit (d, e)
```

Outer: DEPT

read once, row by row

Inner: EMP

read again for every outer row;

one lookup into it is a **probe**

Nested loops: rule 2's lookups

```
for each d in DEPT with LOC = 'DENVER':
    for each e in EMP with e.DNO = d.DNO:
        emit (d, e)
```

Nested loops: rule 2's lookups

```
for each d in DEPT with LOC = 'DENVER':
    for each e in EMP with e.DNO = d.DNO:
        emit (d, e)
```

Line two is not a scan of EMP. It is an access path, chosen by cost with d.DNO supplied as the constant, which is how EMP's index gets to match.

One probe is Table 2's clustered-matching row: 1/100 × (50 + 500) = 5.5 pages.

Nested loops: rule 2's lookups

```
for each d in DEPT with LOC = 'DENVER':
    for each e in EMP with e.DNO = d.DNO:
        emit (d, e)
```

Line two is not a scan of EMP. It is an access path, chosen by cost with d.DNO supplied as the constant, which is how EMP's index gets to match.

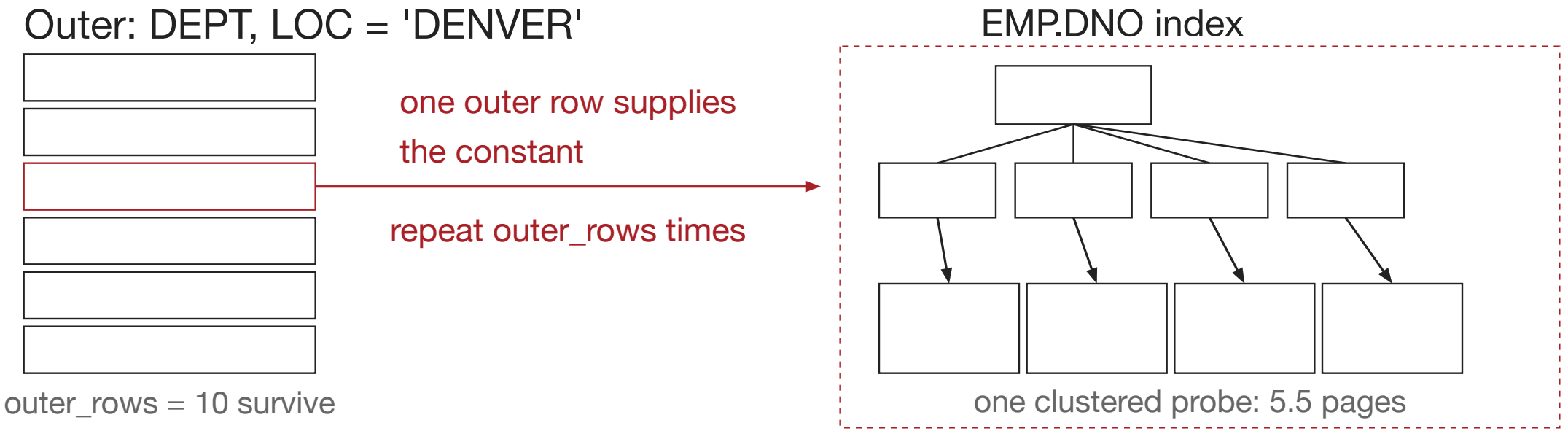
One probe is Table 2's clustered-matching row: 1/100 × (50 + 500) = 5.5 pages.

$$C = C_{outer} + outer_rows \times C_{inner}$$

Read the outer once. Look the inner up once per surviving outer row.

Here 5 + 10 × 5.5 = 60, with outer_rows = 10, not 100: DEPT's filter has already run.

Nested loops: outer and inner



$$C = C_{outer} + outer_rows \times C_{inner} = 5 + 10 \times 5.5 = 60$$

Without an index on EMP.DNO, each lookup scans all 500 pages: 10 × 500 = 5,000, not 55.

Read one outer row, find its matches in the inner; repeat outer_rows times.

What a merge join is

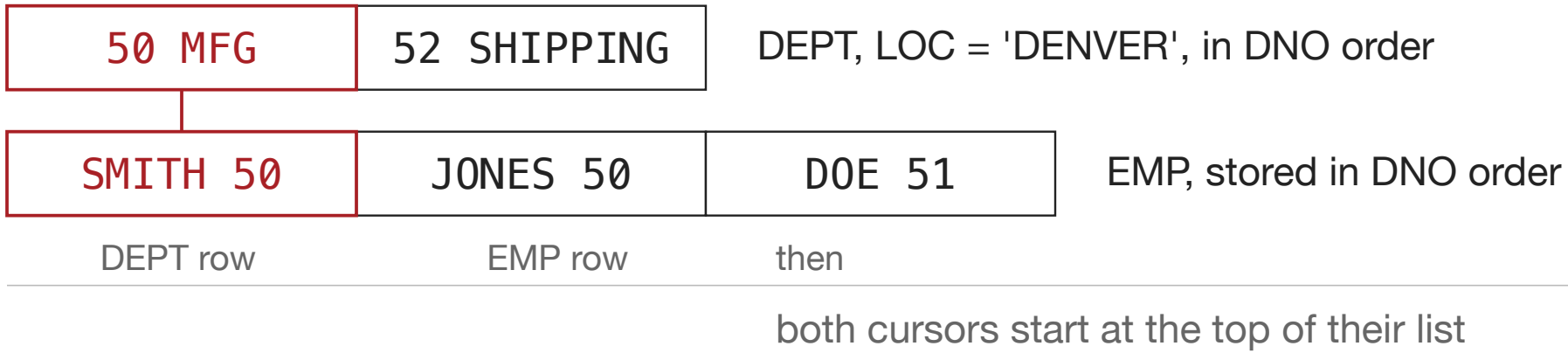
Nested loops pay a lookup per outer row. The other join method looks nothing up:

Merge join	walk both tables side by side in join-column order, the way you merge two sorted lists
It reads	each table once, start to finish, with no lookups
The price	both in join-column order first: sorted, or already stored that way

Worth it when the lookups would be many: 10,000, not 10.

Rule 2’s first query: EMP first needs 10,000 lookups into DEPT; DEPT first needs 10.

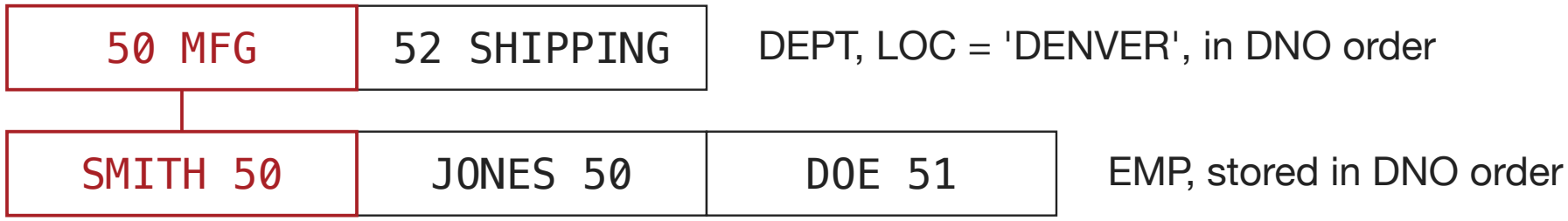
A merge, step by step



Merge join cost: read each input once, in join order.

Clustered index: index + table pages. Otherwise: table pages + a sort.

A merge, step by step

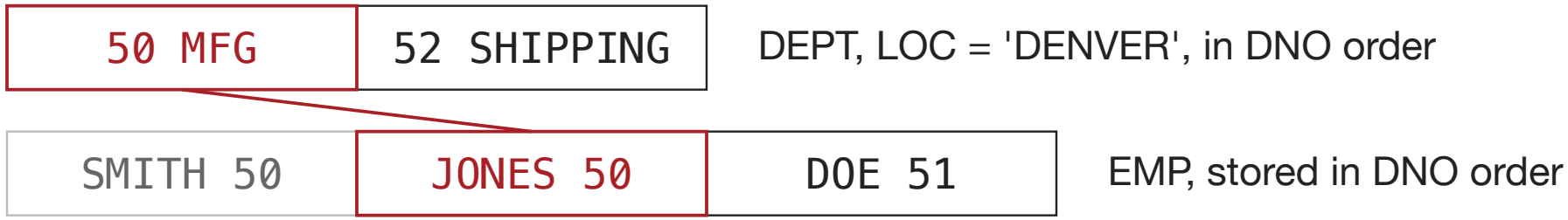


	DEPT row		EMP row	then
1	`50 MFG`	vs	`SMITH 50`	equal: output (MFG, SMITH), next EMP row

Merge join cost: read each input once, in join order.

Clustered index: index + table pages. Otherwise: table pages + a sort.

A merge, step by step



	DEPT row		EMP row	then
1	`50 MFG`	vs	`SMITH 50`	equal: output (MFG, SMITH), next EMP row
2	`50 MFG`	vs	`JONES 50`	equal: output (MFG, JONES), next EMP row

Merge join cost: read each input once, in join order.

Clustered index: index + table pages. Otherwise: table pages + a sort.

A merge, step by step

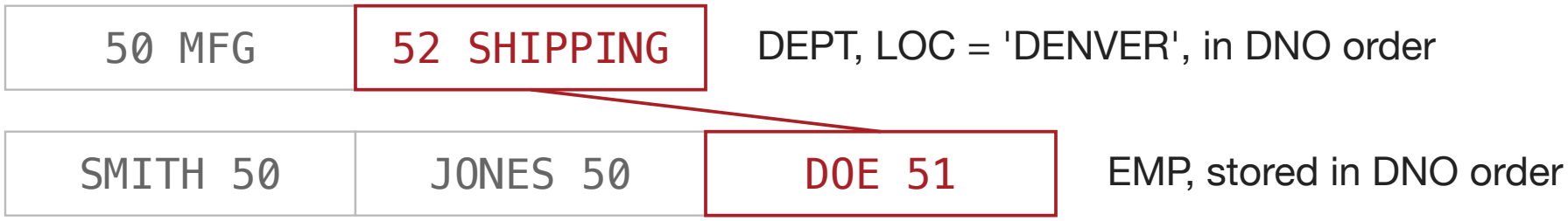


	DEPT row		EMP row	then
1	`50 MFG`	vs	`SMITH 50`	equal: output (MFG, SMITH), next EMP row
2	`50 MFG`	vs	`JONES 50`	equal: output (MFG, JONES), next EMP row
3	`50 MFG`	vs	`DOE 51`	50 < 51: next DEPT row

Merge join cost: read each input once, in join order.

Clustered index: index + table pages. Otherwise: table pages + a sort.

A merge, step by step



	DEPT row		EMP row	then
1	`50 MFG`	vs	`SMITH 50`	equal: output (MFG, SMITH), next EMP row
2	`50 MFG`	vs	`JONES 50`	equal: output (MFG, JONES), next EMP row
3	`50 MFG`	vs	`DOE 51`	50 < 51: next DEPT row
4	`52 SHIPPING`	vs	`DOE 51`	51 < 52: next EMP row: none left, done

Merge join cost: read each input once, in join order.

Clustered index: index + table pages. Otherwise: table pages + a sort.

Which table goes outside

Joining EMP and DEPT; 1,000 rows out either way. DEPT.DNO is **unique**: one probe is Table 2's row one, 1 + 1 = 2. All of DEPT in DNO order is its clustered index, 2 + 5 = 7.

Plan	Arithmetic	Cost
DEPT scan outer, probe EMP.DNO	`5 + 10 × 5.5`	**60**
DEPT.DNO index outer, probe EMP.DNO	`7 + 10 × 5.5`	62
EMP scan outer, probe DEPT.DNO	`500 + 10,000 × 2`	20,500
Merge, both via clustered DNO indexes	`50 + 500 + 7`	557

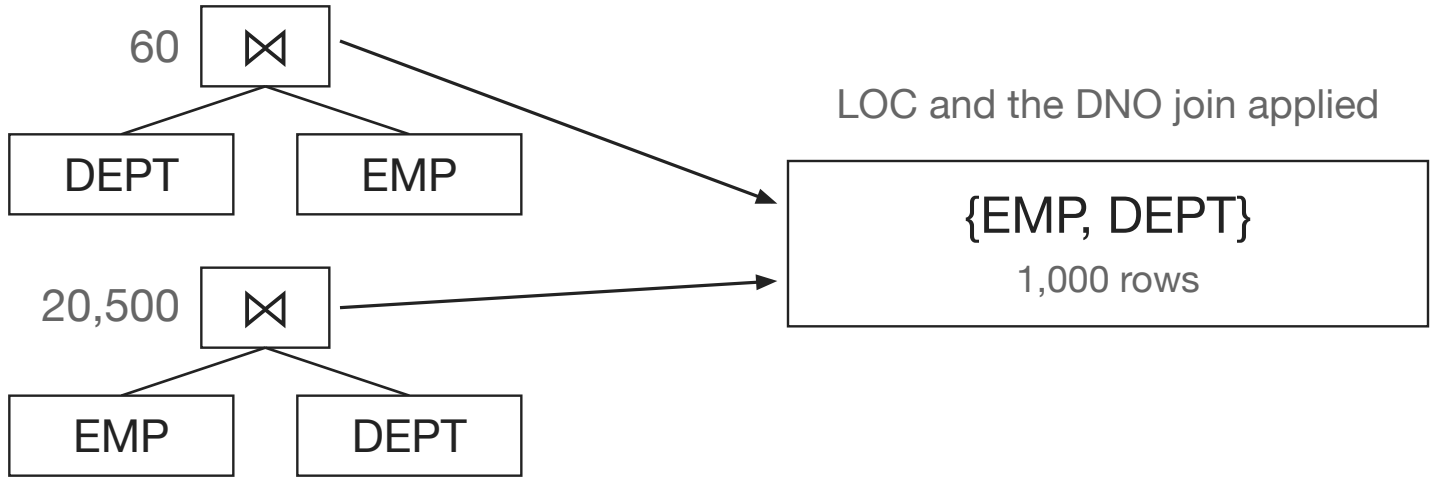
Many tables: the order problem

The order problem

Every join order returns the same rows, at different cost: 163 or 20,603 page fetches on ours. The naive search: price every order, keep the cheapest.

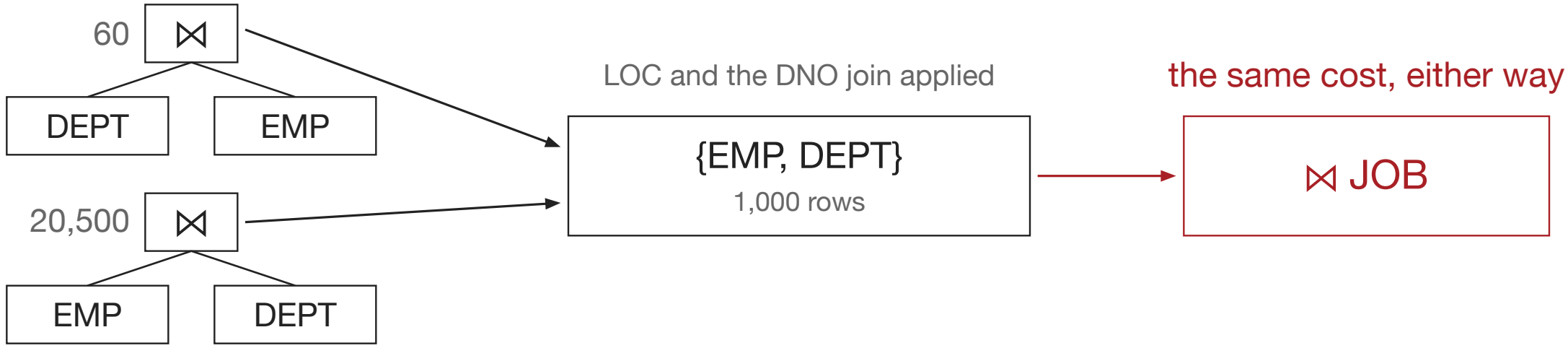
Tables (n)	Orders (n!)	Search time, 1 µs an order
3	6	6 µs
8	40,320	0.04 seconds
12	479,001,600	8 minutes
20	2.4×10^{18}	77,000 years

The observation that saves us



Two ways to join EMP and DEPT. Does the next join care which one you used?

The observation that saves us

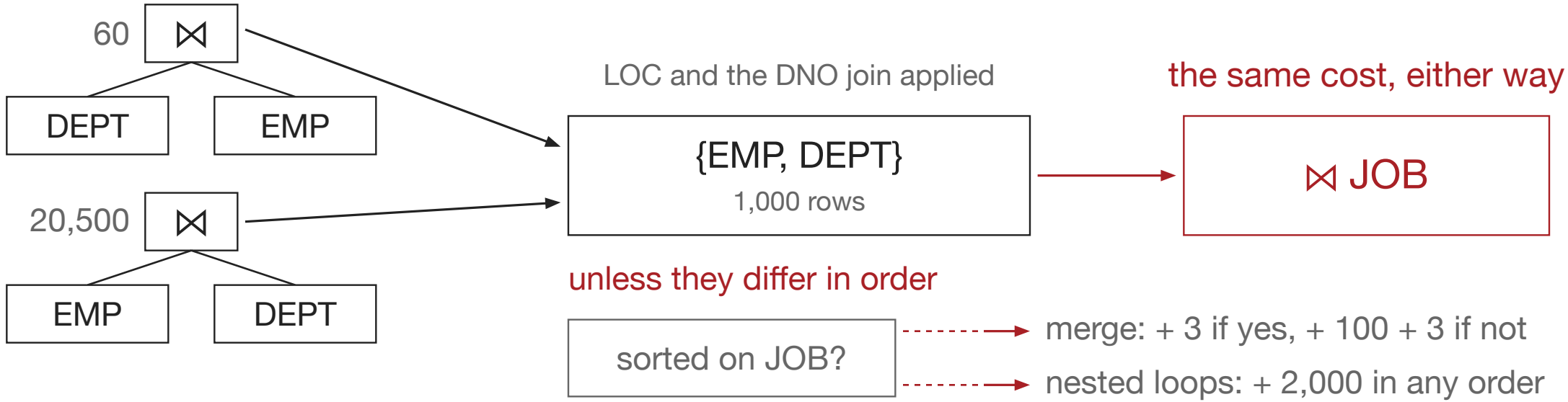


Two ways to join EMP and DEPT. Does the next join care which one you used?

Only through two things: how many rows came out, and in what order.

"independent of the order of joining the first k"

The observation that saves us



Two ways to join EMP and DEPT. Does the next join care which one you used?

Only through two things: how many rows came out, and in what order.

"independent of the order of joining the first k"

Why throwing a plan away is safe

Continuation	With 20,500	With 60
Nested loops into JOB.JOB: + 1,000 × 2	22,500	2,060
Sort 1,000 rows, merge JOB: + 100 + 3	20,603	**163**
Any continuation at all: + X	20,500 + X	60 + X

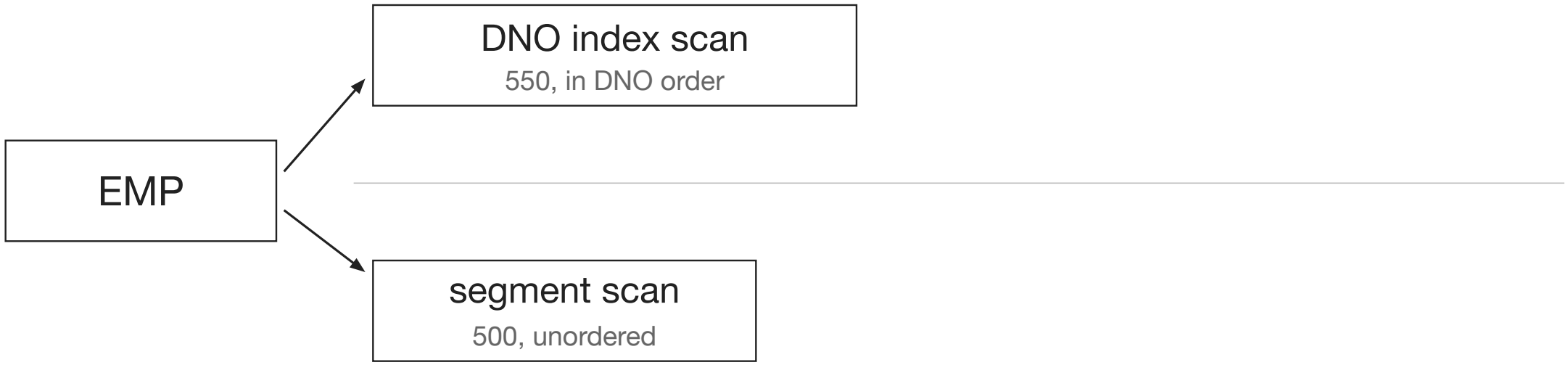
Every total falls by the same 20,440: the 20,500 plan is in no optimal plan.

The swap is valid: both subplans deliver the same 1,000 rows, predicates applied.

Why the state carries the order

For EMP, the scan is cheapest, but the DNO index returns rows already in DNO order.

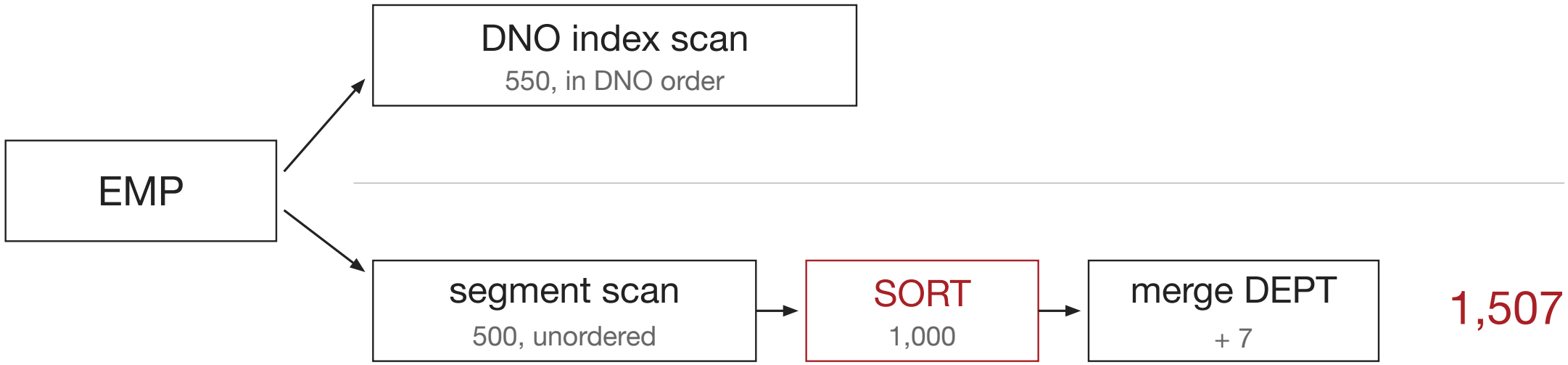
A later join may want that order, so keep both, not just the cheapest.



Why the state carries the order

For EMP, the scan is cheapest, but the DNO index returns rows already in DNO order.

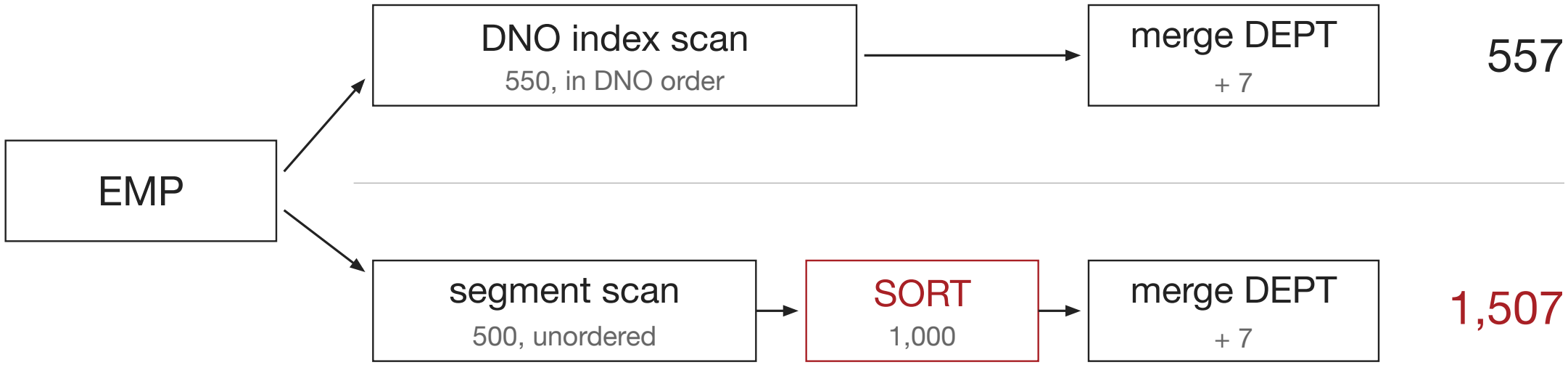
A later join may want that order, so keep both, not just the cheapest.



Why the state carries the order

For EMP, the scan is cheapest, but the DNO index returns rows already in DNO order.

A later join may want that order, so keep both, not just the cheapest.



Spend 50 on the plan that lost, save 950 at the next merge.

Dynamic programming over subsets

```
for each single table:
    keep the best plan, plus the best in each useful order
```

Dynamic programming over subsets

```
for each single table:
    keep the best plan, plus the best in each useful order
for each pair {A, B}:
    every kept plan for {A} ⋈ B and for {B} ⋈ A, both methods; best per order
```

Dynamic programming over subsets

```
for each single table:
    keep the best plan, plus the best in each useful order
for each pair {A, B}:
    every kept plan for {A} ⋈ B and for {B} ⋈ A, both methods; best per order
for each triple:
    every kept plan for each pair in it ⋈ the remaining table; best per order
...
```

Dynamic programming over subsets

```
for each single table:
    keep the best plan, plus the best in each useful order
for each pair {A, B}:
    every kept plan for {A} ⋈ B and for {B} ⋈ A, both methods; best per order
for each triple:
    every kept plan for each pair in it ⋈ the remaining table; best per order
...
```

Each level reads only the level below: how a subset was built no longer matters.

The added table is always the inner, and always a base table.

Keep the best plan per (subset, order), not per join order.

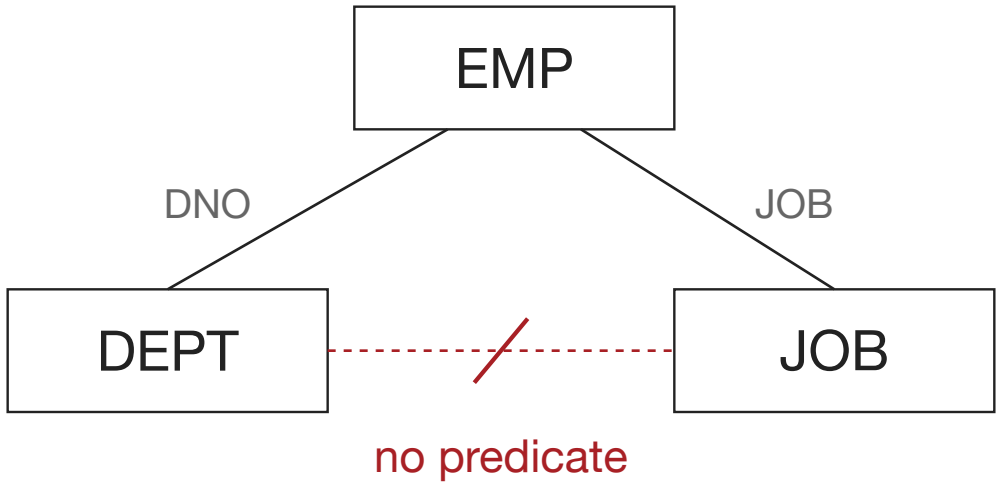
Our query has 3 singles, 2 pairs and 1 triple.

Subsets instead of orders

n	Orders (n!)	Subsets (2^n)	Ratio
3	6	8	0.75× — a loss
5	120	32	3.8×
8	40,320	256	**158×**
10	3,628,800	1,024	3,544×
12	479,001,600	4,096	117,000×

No cross products

Join predicates



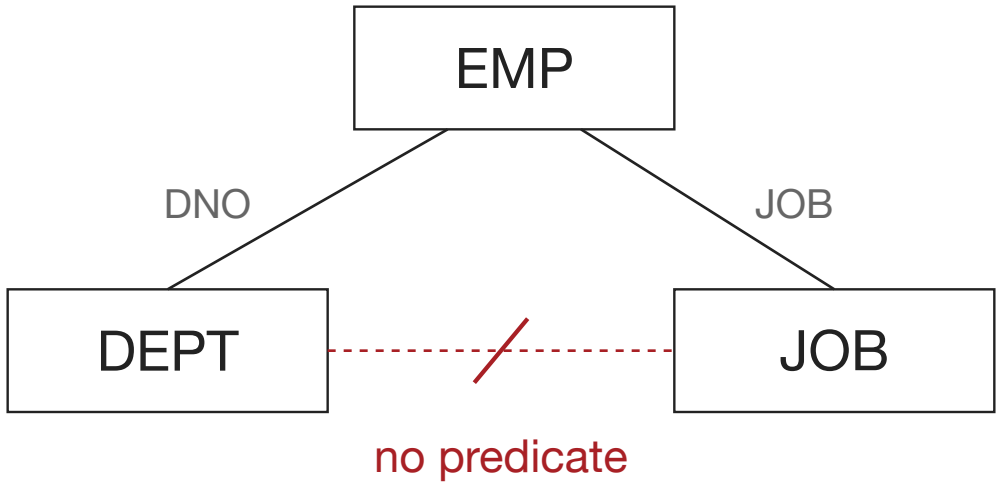
The six orders

```
DEPT, EMP, JOB
EMP, DEPT, JOB
EMP, JOB, DEPT
JOB, EMP, DEPT
DEPT, JOB, EMP
JOB, DEPT, EMP
```

Add a table only if it shares a join predicate with the tables joined so far.

No cross products

Join predicates



The six orders

```
DEPT, EMP, JOB
EMP, DEPT, JOB
EMP, JOB, DEPT
JOB, EMP, DEPT
```

~~DEPT, JOB, EMP~~ cross product first

~~JOB, DEPT, EMP~~ cross product first

Add a table only if it shares a join predicate with the tables joined so far.

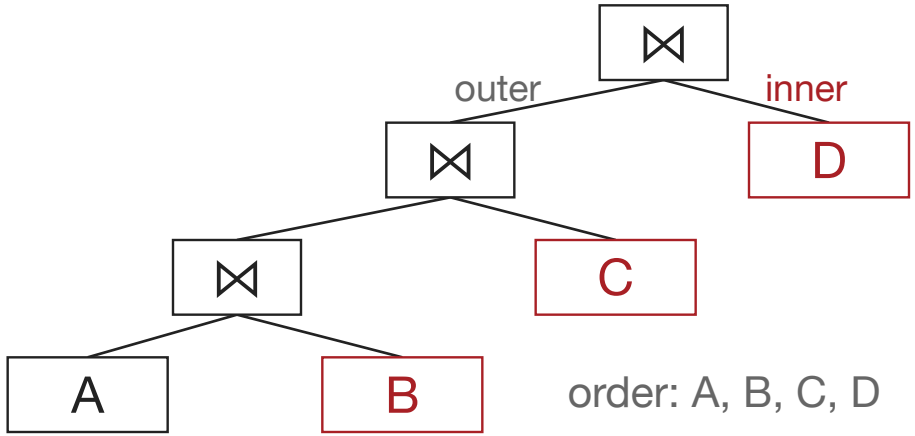
{DEPT, JOB} has no join predicate, so that subset is never entered.

Cross products are needed only for a disconnected join graph, and then as late as possible.

Left-deep only

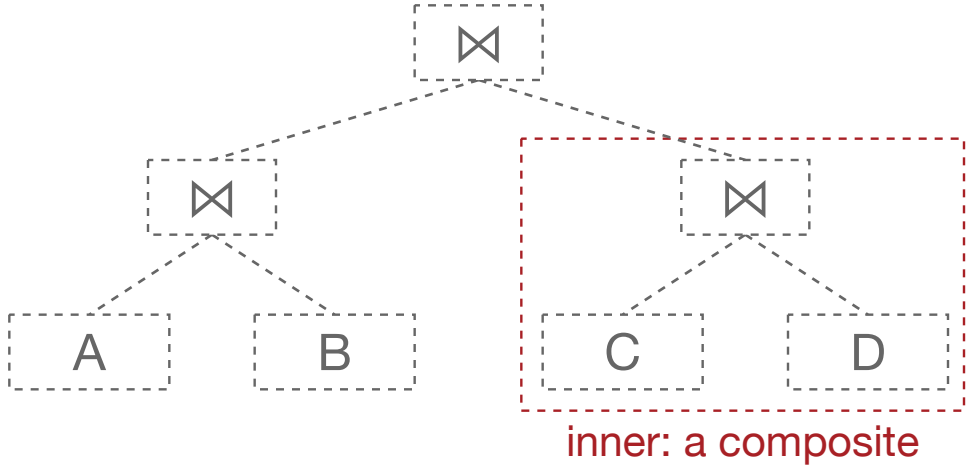
Each join adds one base table, the **inner**, to the composite built so far, the **outer**.

Left-deep: considered



order: A, B, C, D

Bushy: not considered

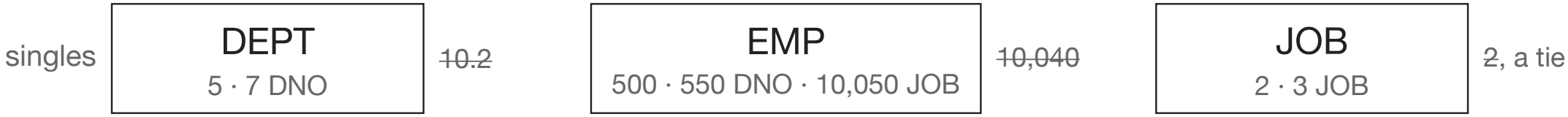


Why: a base-table inner can be probed through its index, as nested loops do;
a composite is only ever an outer, read once, so it streams on unless a merge must sort it.

The inner is never a composite, so a plan's shape is just an order of tables.

With three tables there is no bushy tree; the rule still excludes JOB ⋈ (DEPT ⋈ EMP).

The search: singles and pairs



kept: the cheapest, plus the cheapest in each useful order

The search: singles and pairs

singles

DEPT	EMP	JOB
5 · 7 DNO	500 · 550 DNO · 10,050 JOB	2 · 3 JOB
~~10.2~~	~~10,040~~	~~2~~, a tie

kept: the cheapest, plus the cheapest in each useful order

5	segment scan: all 5 pages, LOC = 'DENVER' tested as rows are read · the cheapest
7 DNO	DNO index, clustered, read end to end: 2 + 5 · the cheapest in DNO order
~~10.2~~	LOC index, non-clustered, matching LOC: 1/10 × (2 + 100) · dearer than 5, and no join wants LOC order

The search: singles and pairs

singles

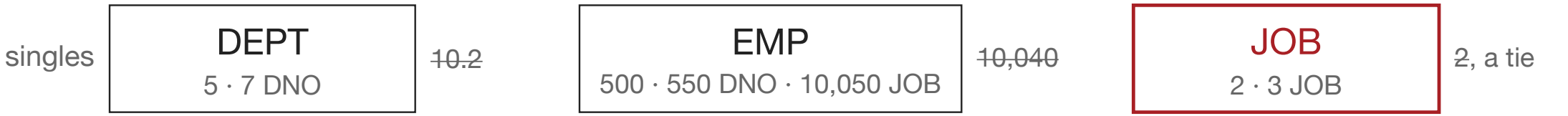
DEPT		EMP		JOB
5 · 7 DNO	~~10.2~~	500 · 550 DNO · 10,050 JOB	~~10,040~~	2 · 3 JOB

~~2~~, a tie

kept: the cheapest, plus the cheapest in each useful order

500	segment scan: all 500 pages · the cheapest
550 DNO	DNO index, clustered, read end to end: 50 + 500 · the cheapest in DNO order
10,050 JOB	JOB index, non-clustered, read end to end: 50 + 10,000 · the only path in JOB order
~~10,040~~	SAL index, non-clustered, read end to end: 40 + 10,000 · dearer than 500, and no join wants SAL order

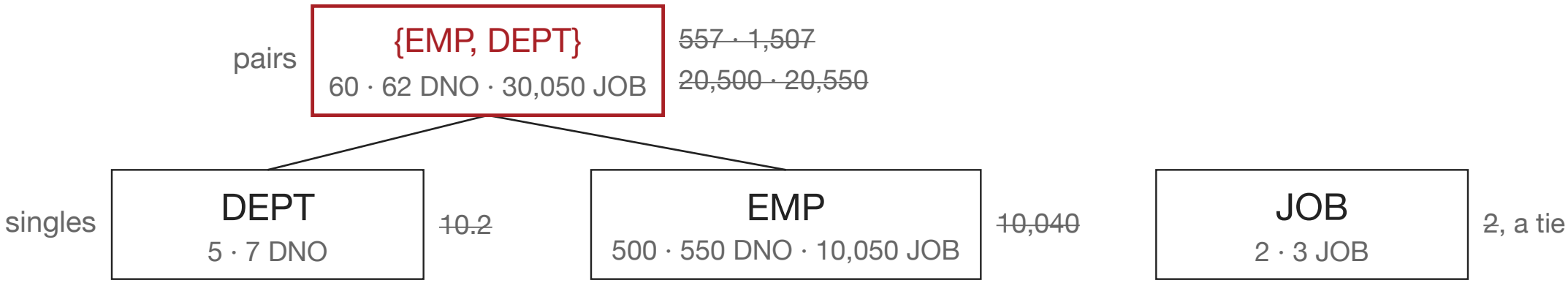
The search: singles and pairs



kept: the cheapest, plus the cheapest in each useful order

2	TITLE index, unique, TITLE = 'CLERK': 1 index page + 1 data page · the cheapest
3 JOB	JOB index, clustered, read end to end: 1 + 2 · the cheapest in JOB order
~~2~~	segment scan: both pages · ties the TITLE index; one unordered plan is kept

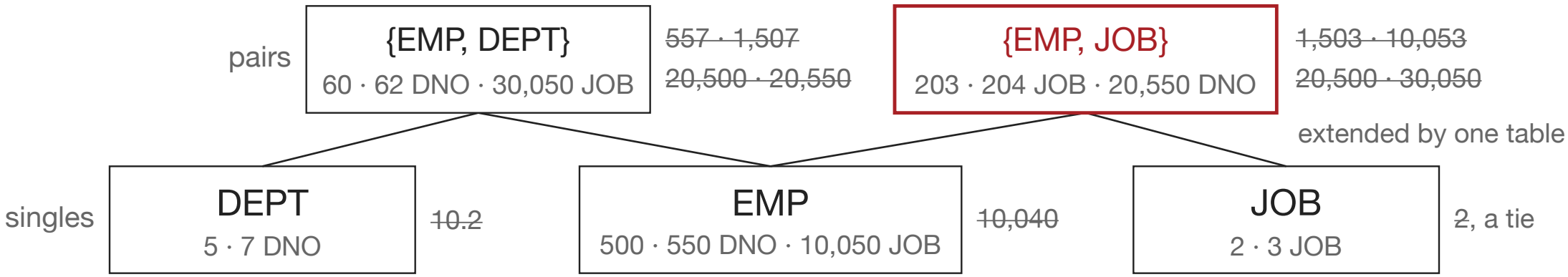
The search: singles and pairs



kept: the cheapest, plus the cheapest in each useful order

60	DEPT's scan 5 + a probe of EMP's DNO index per Denver row: 10 × 5.5 · the cheapest
62 DNO	DEPT's DNO index 7 + the same 10 probes, 55 · the cheapest in DNO order
30,050 JOB	EMP's JOB index 10,050 + a lookup in DEPT per row: 10,000 × 2 · the only plan in JOB order
~~557 · 1,507~~	merges: EMP in DNO order (index 550, or scan + sort 1,500) + DEPT's 7 · dearer than 62
~~20,500 · 20,550~~	EMP outside (scan 500 or DNO index 550) + 10,000 × 2 · dearer than 60 and 62

The search: singles and pairs



{DEPT, JOB}: no join predicate, never entered

kept: the cheapest, plus the cheapest in each useful order

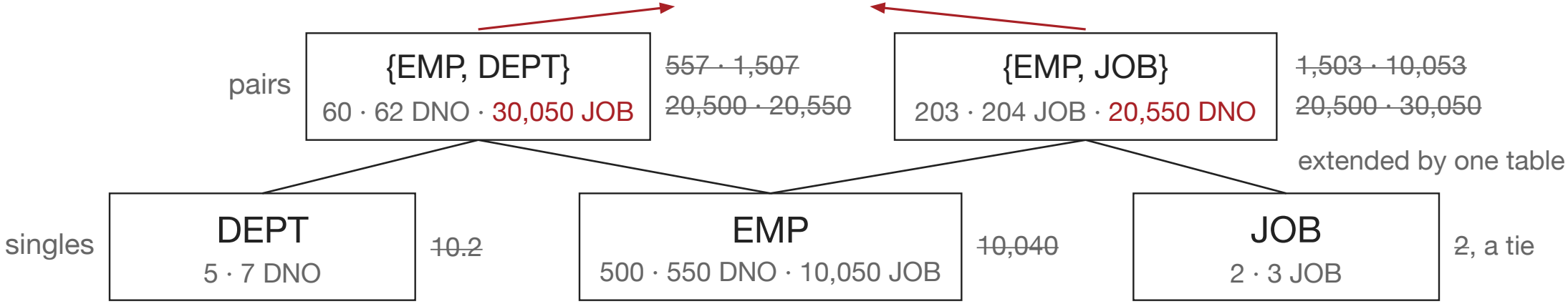
203	JOB's TITLE index 2 + a probe of EMP's JOB index for its 1 row: 201 · the cheapest
204 JOB	JOB's JOB index 3 + the same probe, 201 · the cheapest in JOB order
20,550 DNO	EMP's DNO index 550 + a lookup in JOB per row: 10,000 × 2 · the only plan in DNO order
~~1,503 · 10,053~~	merges: EMP in JOB order (scan + sort 1,500, or index 10,050) + JOB's 3 · dearer than 204
~~20,500 · 30,050~~	EMP outside (scan 500 or JOB index 10,050) + 10,000 × 2 · dearer than 203 and 204

The search: singles and pairs

30,050 and 20,550 are kept only for their order

the same rows as the 60 and the 203, already in JOB and DNO order

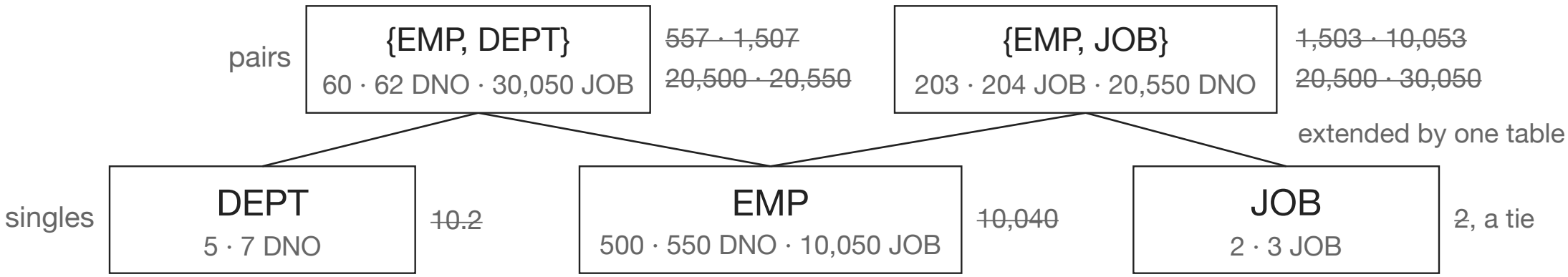
Will the triple use either?



{DEPT, JOB}: no join predicate, never entered

kept: the cheapest, plus the cheapest in each useful order

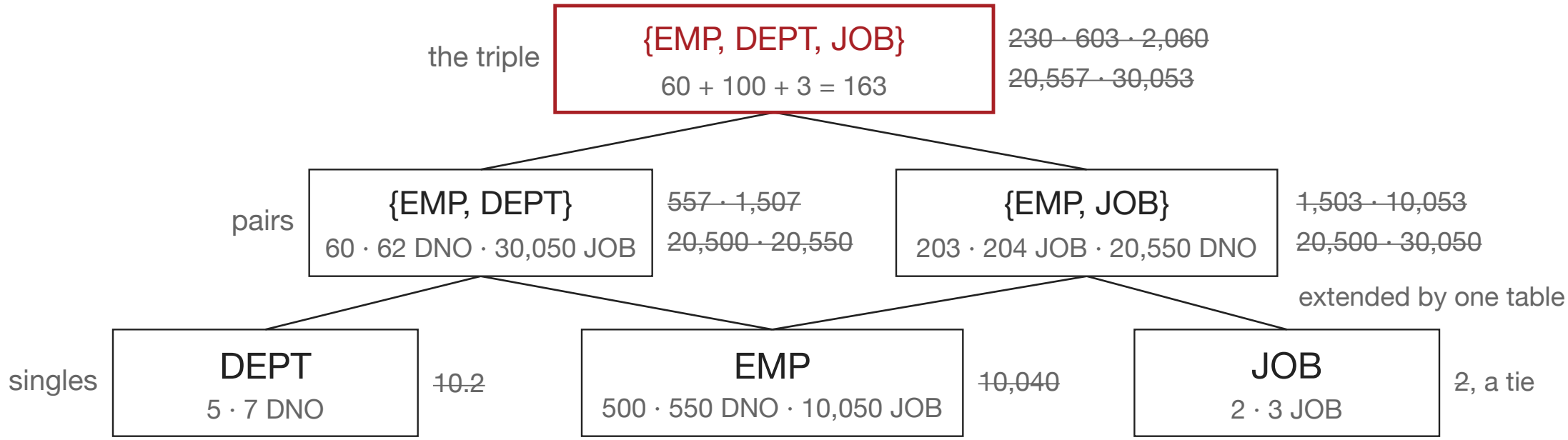
The search: the triple and the winner



{DEPT, JOB}: no join predicate, never entered

kept: the cheapest, plus the cheapest in each useful order

The search: the triple and the winner

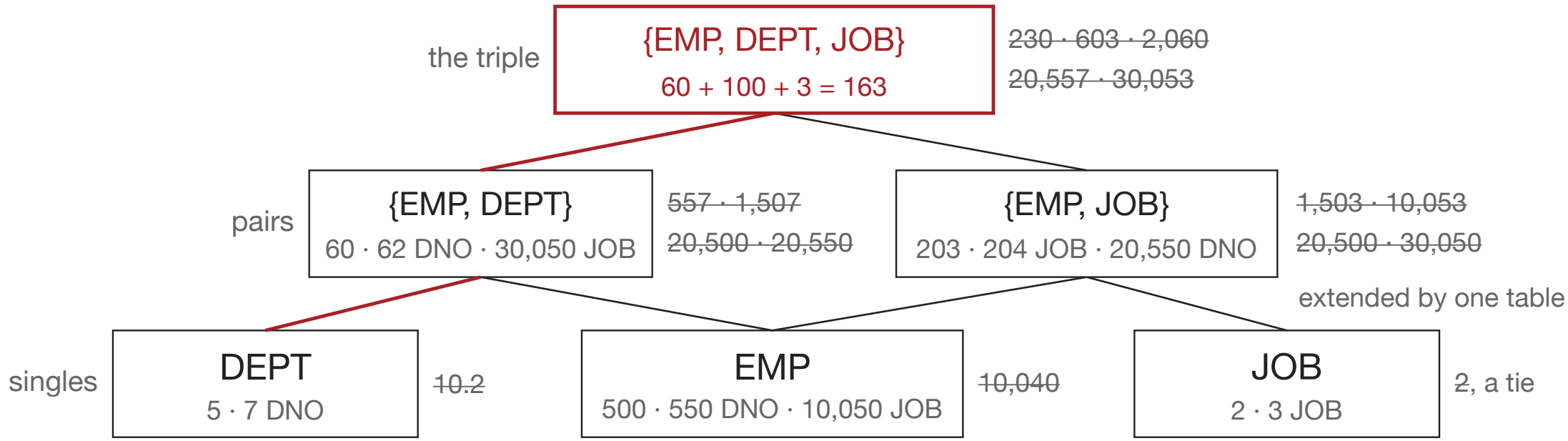


{DEPT, JOB}: no join predicate, never entered

kept: the cheapest, plus the cheapest in each useful order

163	{EMP, DEPT}'s 60 + 100 to sort its 1,000 rows on JOB + 3 to merge with JOB · the cheapest; with no ORDER BY, the only plan kept
~~230 · 603~~	{EMP, JOB}'s 203, then DEPT by sort 20 + merge 7, or by 200 × 2 lookups · dearer than 163
~~2,060~~	{EMP, DEPT}'s 60 + a lookup in JOB per row: 1,000 × 2 · dearer than 163
~~20,557 · 30,053~~	the ordered entries need no sort: 20,550 + 7 and 30,050 + 3 · the sort they save costs 20 or 100

The search: the triple and the winner

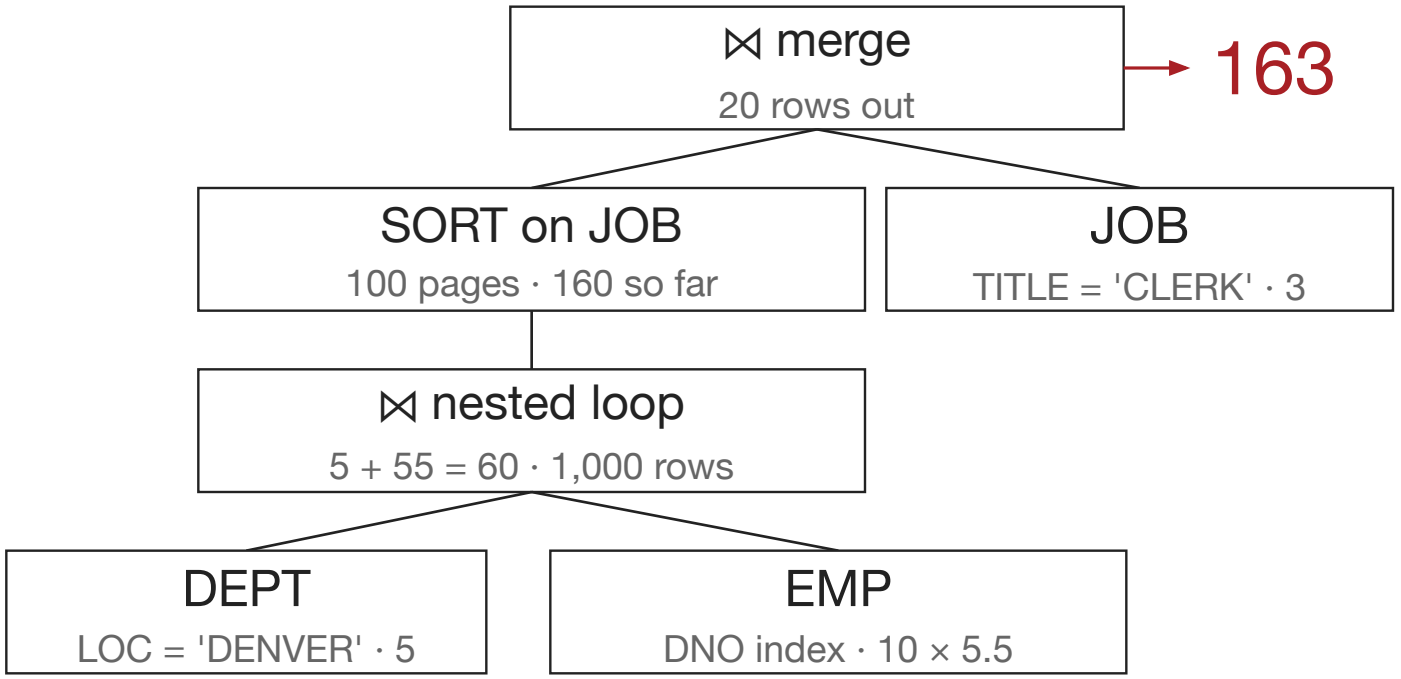


{DEPT, JOB}: no join predicate, never entered

kept: the cheapest, plus the cheapest in each useful order

163	{EMP, DEPT}'s 60 + 100 to sort its 1,000 rows on JOB + 3 to merge with JOB · the cheapest; with no ORDER BY, the only plan kept
~~230 · 603~~	{EMP, JOB}'s 203, then DEPT by sort 20 + merge 7, or by 200 × 2 lookups · dearer than 163
~~2,060~~	{EMP, DEPT}'s 60 + a lookup in JOB per row: 1,000 × 2 · dearer than 163
~~20,557 · 30,053~~	the ordered entries need no sort: 20,550 + 7 and 30,050 + 3 · the sort they save costs 20 or 100

The plan it picks



the same 20 rows, priced three other ways

the rule book

22,500 138×

the FROM order as written

20,603 126×

the tree as parsed

≈ 5,000,000 ≈ 31,000×

What the search itself cost

Search cost	System R, 1979
Storage for a typical optimization	A few thousand bytes
CPU on an IBM 370/158	A few tenths of a second
An eight-relation join	Optimized in a few seconds

A plan is compiled once and run many times, so the search amortizes.

Recap

1. **Rules:** each held on one query and broke on the next.
2. **Cost:** page fetches, the one number every plan is ranked by.
3. **Estimates:** four catalog numbers and a factor per predicate, multiplied.
4. **One table:** the cheapest access path, plus the cheapest per useful order.
5. **Two tables:** nested loops or merge scan, whichever costs less.
6. **Many tables:** DP over subsets, one plan per (subset, order).

163 page fetches for the 20 rows, against 22,500 for the rule book.

Next class, Spanner: the cost measure grows a network term and a shard fan-out term.

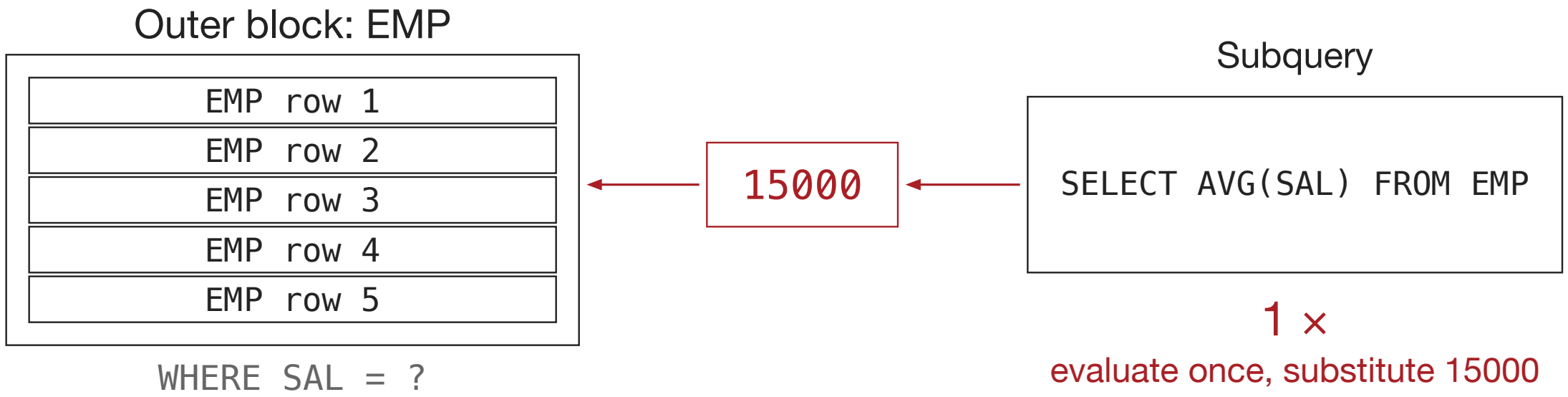
Appendix

Nested queries

Uncorrelated: evaluate once

```
SELECT NAME FROM EMP
WHERE  SAL = (SELECT AVG(SAL) FROM EMP)
```

Uncorrelated: the subquery names no outer column — one answer for the whole query.

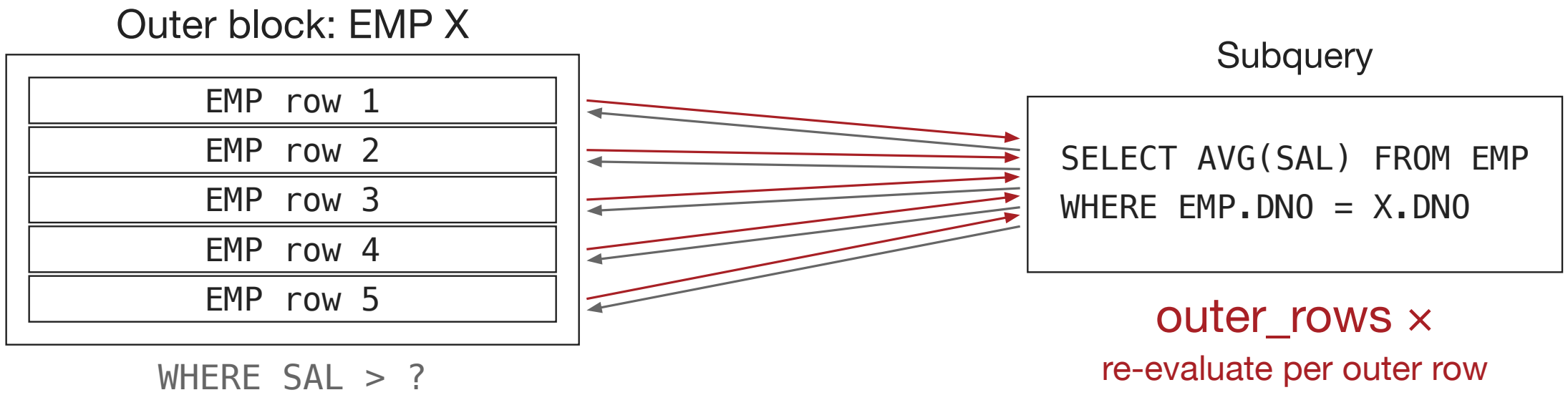


AVG(SAL) returns 15000, so the outer block optimizes SAL = 15000.

Correlated: a nested loop in disguise

```
SELECT NAME FROM EMP X
WHERE  SAL > (SELECT AVG(SAL) FROM EMP WHERE EMP.DNO = X.DNO)
```

Correlated: the subquery names X.DNO, the current outer row's value.



One inner evaluation per outer row. That is a nested-loop join.

What held up, what did not

The 1979 claim, and a test of it

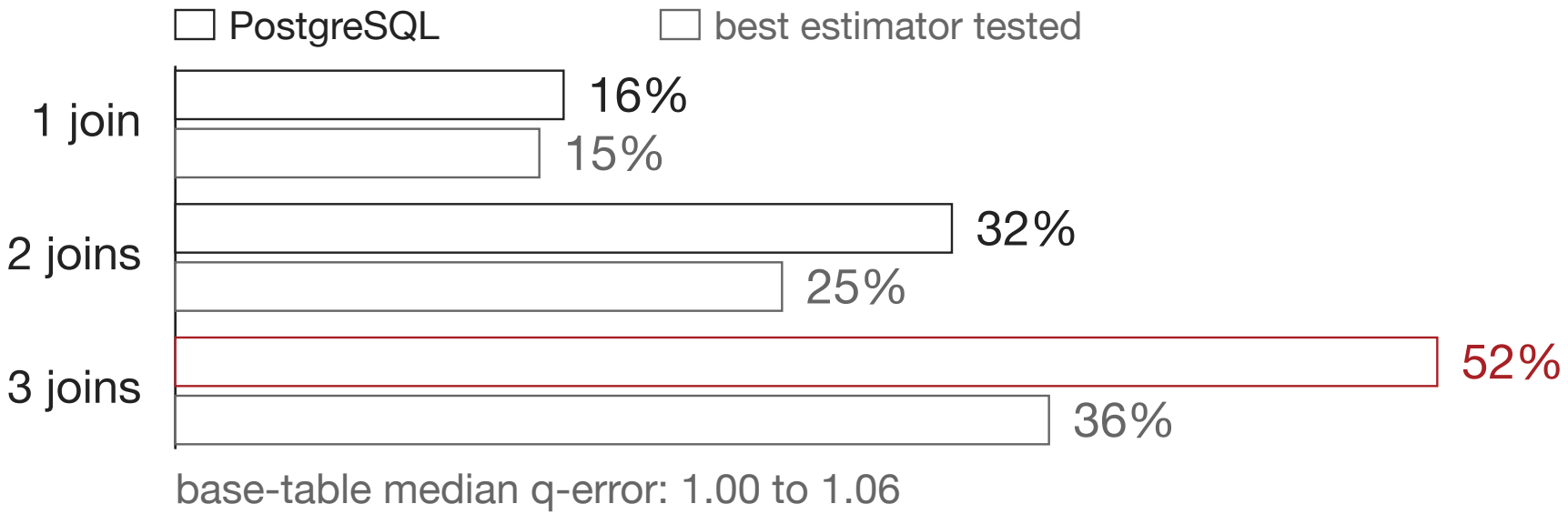
What they checked	What they found
Absolute costs	“often not accurate in absolute value”
Which plan wins	“the true optimal path is selected in a large majority of cases”
The full ordering	“in many cases, the ordering … is precisely the same as that among the actual measured costs”

The test: Leis et al., 2015 and 2018 — 113 queries on the real IMDB database, 3.6 GB, 21 tables, 3 to 16 joins, average 8.

The 1979 findings in the table were stated as preliminary, from work “in progress”.

Base tables are fine; joins are not

q-error: the factor by which an estimate differs from the truth.



Each join multiplies two estimates as if the columns were independent, once per join.

The failure is not in the statistics. It is in combining them.

Fraction of estimates wrong by 10× or more, on the 113 IMDB queries; here mostly underestimates.

The cost model matters less

True cardinalities for every model, so only the cost model varies.

Cost model	Median error vs. run time
PostgreSQL’s default, about 4,000 lines of C	38%
The same model, tuned for main memory	30%

Plans picked by a model that just counts rows through operators run 34% faster than the default’s; plans picked by the tuned model, 41% faster.

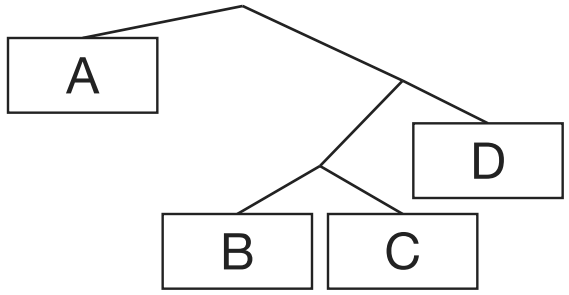
Thirty per cent from the cost model. Orders of magnitude from cardinalities.

A main-memory result. On disk, the journal version finds it matters more.

The search space had to widen

Each tree's cost against the best **bushy** plan — median above, **95th / max** below. Bushy joins two composites, {A, B} ⋈ {C, D}; left-deep forbids that. True cardinalities, PK+FK indexes.

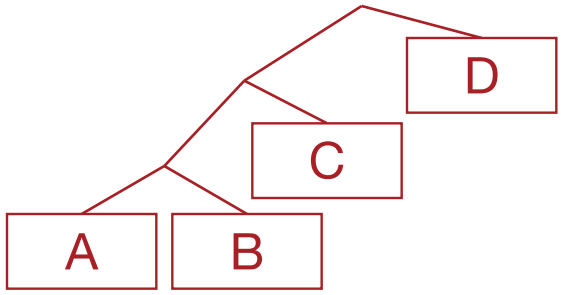
Zig-zag



1.00

1.16 / 1.88

Left-deep

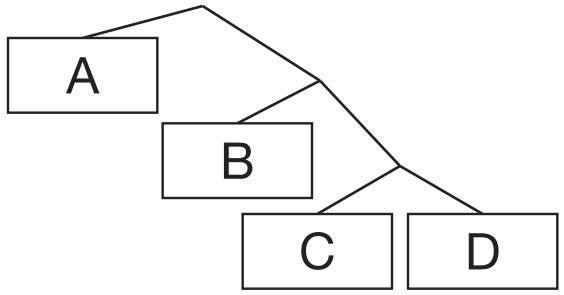


1.94

48.2 / 1,252

System R's space

Right-deep



6.46

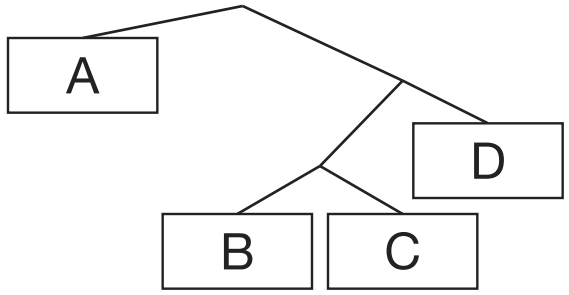
140 / 6,108

So did the 1979 search survive?

The search space had to widen

Each tree's cost against the best **bushy** plan — median above, **95th / max** below. Bushy joins two composites, {A, B} ⋈ {C, D}; left-deep forbids that. True cardinalities, PK+FK indexes.

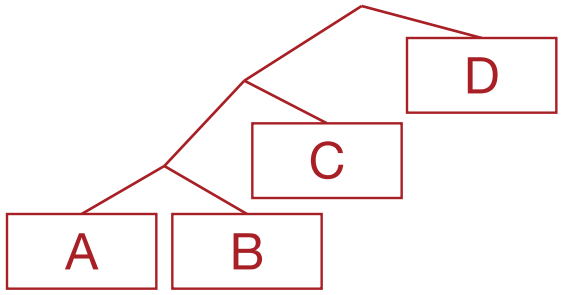
Zig-zag



1.00

1.16 / 1.88

Left-deep

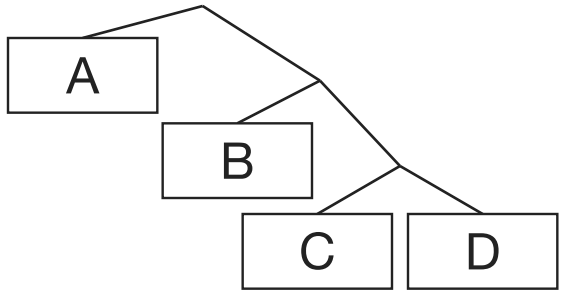


1.94

48.2 / 1,252

System R's space

Right-deep



6.46

140 / 6,108

So did the 1979 search survive?

The enumeration did. The search space did not.

The line to remember

The 1979 enumeration survived.

The 1979 search space had to be widened.

The 1979 statistics were replaced.

And the 1979 assumptions, which were the real problem, are still with us.

The design, assembled

1. **Rewrite:** parse into blocks, convert to CNF, push each filter to its table.
2. **Cost:** page fetches (System R adds a CPU term we leave out).
3. **Statistics:** kept in the catalog, refreshed by hand.
4. **Selectivity:** one factor per predicate from Table 1; multiply.
5. **Access paths:** per table, the cheapest overall and per useful order.
6. **Join methods:** nested loops or merge scan, decided by cost.
7. **Join order:** DP over subsets, one plan per (subset, order); left-deep, no cross products.
8. **Nested queries:** once if uncorrelated, per outer row if correlated.

Appendix: words for the choice cards

Page fetch — one data or index page read from disk. The unit of every cost.

Access path — one way to read one table: a scan, or one of its indexes.

Segment scan — System R's name for a scan: every page of the table, once.

Clustered — table stored in that index's key order. Only one index can be.

Matching — the predicate filters that index's own key, so a scan can narrow.

Legal order — each table after the first shares a join column with an earlier one.

Nested loops — read one table once; look each of its rows up in the other.

Merge scan — put both in join-column order, then walk them together once.

None of these words is in the SQL. Every one is the system's choice.

Each word is earned, with arithmetic under it, in the core lecture.

Appendix: which table first

Read the first table, look the second up once per surviving row, then sort and merge the third. DEPT: scan 5, 10 Denver rows. JOB: TITLE index 2, 1 clerk. EMP: scan 500, all 10,000 rows.

First, second, third	Read + survivors × lookup + sort + merge	Cost
DEPT, EMP, JOB	`5 + 10 × 5.5 + 100 + 3`	**163**
JOB, EMP, DEPT	`2 + 1 × 201 + 20 + 7`	230
EMP, JOB, DEPT	`500 + 10,000 × 2 + 20 + 7`	20,527
EMP, DEPT, JOB	`500 + 10,000 × 2 + 100 + 3`	20,603

Four legal orders, a 126-fold spread; the FROM list names the worst.

DEPT and JOB share no column: the two orders that pair them first are cross products.

Appendix: where 163 comes from

The winning order, DEPT then EMP then JOB, one term at a time.

Term	What it pays for	Rows out
`5`	read DEPT once: 5 pages, and the Denver rows are kept	10
`10 × 5.5`	one EMP lookup per Denver row: `1/100 × (50 + 500)`	1,000
`100`	sort those 1,000 pairs on JOB: `2 × ceil(1,000/20)`	1,000
`3`	one pass of JOB in JOB order: 1 index + 2 data pages	20

5 + 55 + 100 + 3 = 163 page fetches, for twenty rows of answer.

The 1,000 pairs: 10,000 EMP × 10 Denver rows, one in a hundred agreeing on DNO.

Appendix: what one lookup costs

Every join cost in this deck is some number of lookups. Here is what one costs, and why.

What one lookup reads	Table 2	Pages
EMP by DNO, clustered: 100 rows on 5 pages	`1/100 × (50 + 500)`	**5.5**
EMP by JOB, scattered: 200 rows on 200 pages	`1/50 × (50 + 10,000)`	201
either unique index: 1 row on 1 page	`1 + 1`	2

Clustered, rows share pages. Scattered, each row costs its own fetch.

Clustered F × (index_pages + pages), non-clustered F × (index_pages + rows), with F = 1/100 for DNO and 1/50 for JOB. The outer row's value lets the inner index match.

Appendix: why the last two terms differ

Of the four legal orders, the two that join DEPT to EMP first end in `+ 100 + 3`, and the two that join JOB to EMP first end in `+ 20 + 7`. All four add the third table the same way: sort the pair you have, then walk it and the third table once each.

First pair built	Pairs	Sort them	Then one pass of
DEPT and EMP, JOB left	1,000	`2 × ceil(1,000/20) = 100`	JOB: `1 + 2 = 3`
JOB and EMP, DEPT left	200	`2 × ceil(200/20) = 20`	DEPT: `2 + 5 = 7`

The tail follows the pair you built, not the table you read first.

1,000 = 10,000 × 10 × 1/100 and 200 = 10,000 × 1 × 1/50. Looking the third table up instead would cost `1,000 × 2` or `200 × 2`, so the sort wins both times.

Appendix: index or scan

The one-table query `... FROM EMP WHERE SAL > 20000`; the running query never filters EMP. EMP: 10,000 rows on 500 pages, 2,500 of them match. Four paths, one answer.

Path on EMP	Arithmetic	Cost
Segment scan: read every page	`500`	**500**
DNO clustered index, no match	`50 + 500`	550
SAL index, matches the predicate	`1/4 × (40 + 10,000)`	2,510
JOB index, no match	`50 + 10,000`	10,050

The matching index costs 5× the scan: it pays per row, not per page.

On the running query nothing matches: the four cost 500, 550, 10,040, 10,050.

Appendix: which join method

Two join methods, three plans for the pair. Each returns the same 1,000 EMP-DEPT pairs.

Plan for the pair	Arithmetic	Cost
Nested loops: read DEPT, look up EMP	`5 + 10 × 5.5`	**60**
Nested loops: read EMP, look up DEPT	`500 + 10,000 × 2`	20,500
Merge scan: walk both in DNO order	`550 + 7`	557

Merge loses by 497 on this pair, wins by 1,897 once JOB is added.

Merging is cheap. Getting EMP into DNO order costs 550 before it starts; one pass of DEPT, 7.

Once JOB is added, sorting the 1,000 pairs and merging costs 100 + 3, against 1,000 × 2 to look JOB up.

Appendix: sort, or never sort

Sorting Q rows costs 2 × ceil(Q / 20) page fetches, out and back at 20 rows per page.

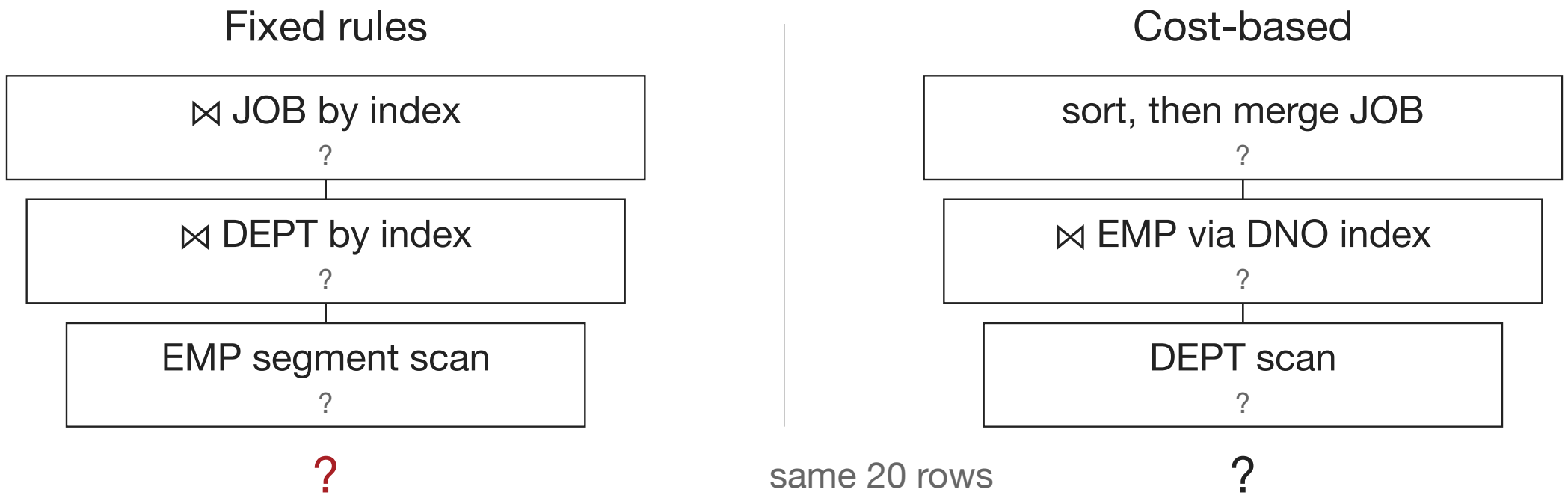
What is sorted	With a sort	Instead of	Saves
1,000 EMP-DEPT rows, on JOB	100 + 3	1,000 × 2 to look up JOB	**1,897**
200 EMP-JOB rows, on DNO	20 + 7	200 × 2 to look up DEPT	373
10,000 EMP rows, on DNO	500 + 1,000	550 via the DNO index	–950

The same sort saves 1,897 on 1,000 rows and loses 950 on 10,000.

Each saving is the subtraction beside it: 2,000 – 103, 400 – 27, 550 – 1,500.

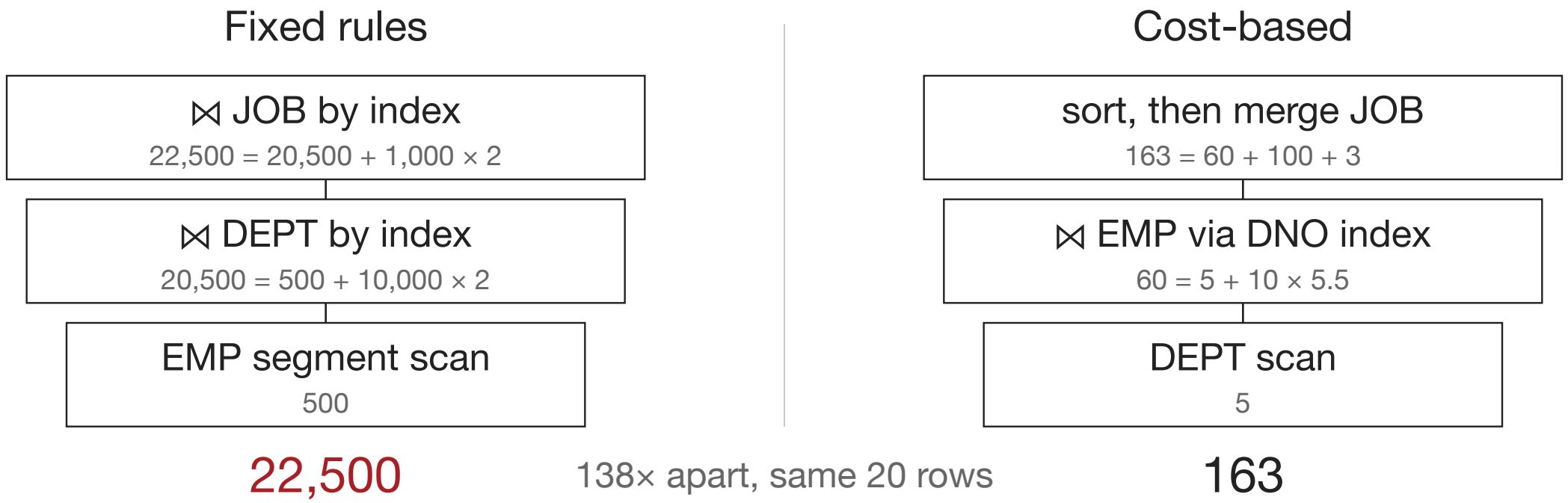
Row three must scan EMP for 500 before it can sort.

Appendix: the bill for the rule book



What does the rule book cost, against the best plan?

Appendix: the bill for the rule book



What does the rule book cost, against the best plan?

22,500 page fetches against 163. A factor of 138.

Appendix: where 22,500 comes from

The rule book's plan, one term at a time. Each node totals everything beneath it.

Term	What it pays for	Rows out
`500`	read EMP first, by a scan: no predicate for an index	10,000
`10,000 × 2`	one DEPT lookup per EMP row, `1 + 1` on a unique index	1,000
`1,000 × 2`	one JOB lookup per pair, `1 + 1` again, not a sort	20

500 + 20,000 + 2,000 = 22,500 page fetches, for the same 20 rows.

All 10,000 lookups are paid for, because they match on DNO, not LOC; only 1,000 find a Denver row. The FROM order's first pair adds 20,440 (20,500 – 60); looking JOB up instead of sorting adds 1,897. Together 22,337 = 22,500 – 163.

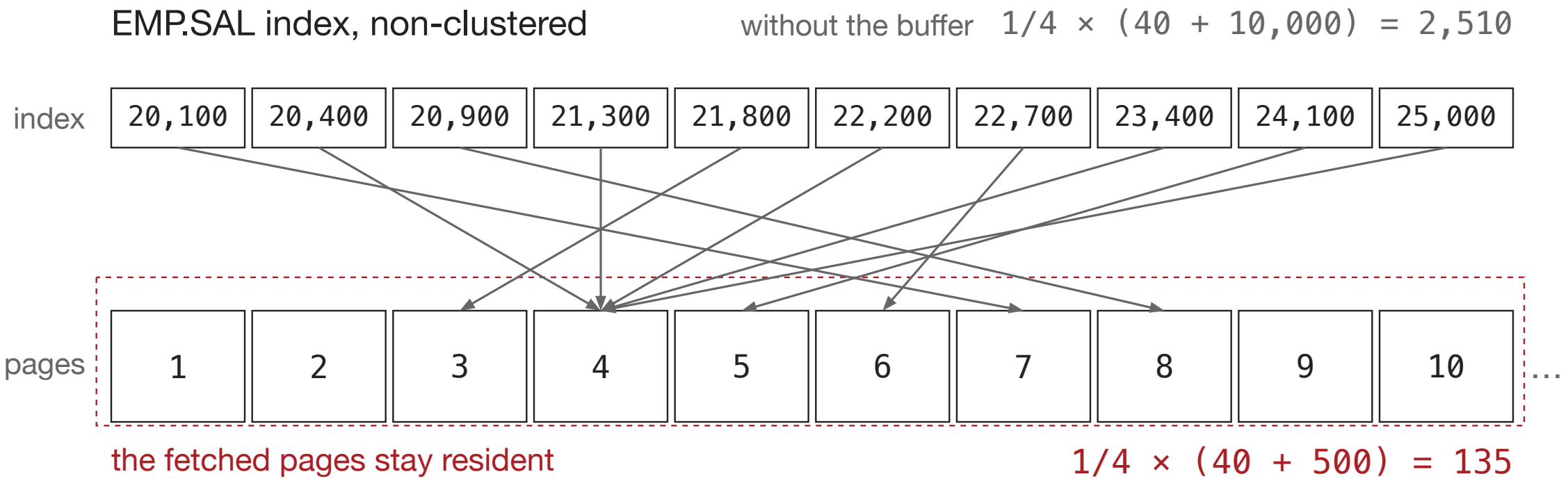
Appendix: Table 1 in full

Predicate	Selectivity factor F
`col = v`	1 / distinct(col) with an index on col, else **1/10**
`col1 = col2`	1 / the larger distinct; one index: 1 / its distinct; else **1/10**
`col > v`	(high – v) / (high – low) if numeric, v known; else **1/3**
`col BETWEEN a AND b`	(b – a) / (high – low) on the same condition, else **1/4**
`col IN (list)`	n × F(col = v), capped at 1/2
`col IN subquery`	the subquery's result_rows / the product of its tables' rows

AND is F1 × F2 · OR is F1 + F2 – F1 × F2 · NOT is 1 – F

Appendix: the buffer-fit alternative

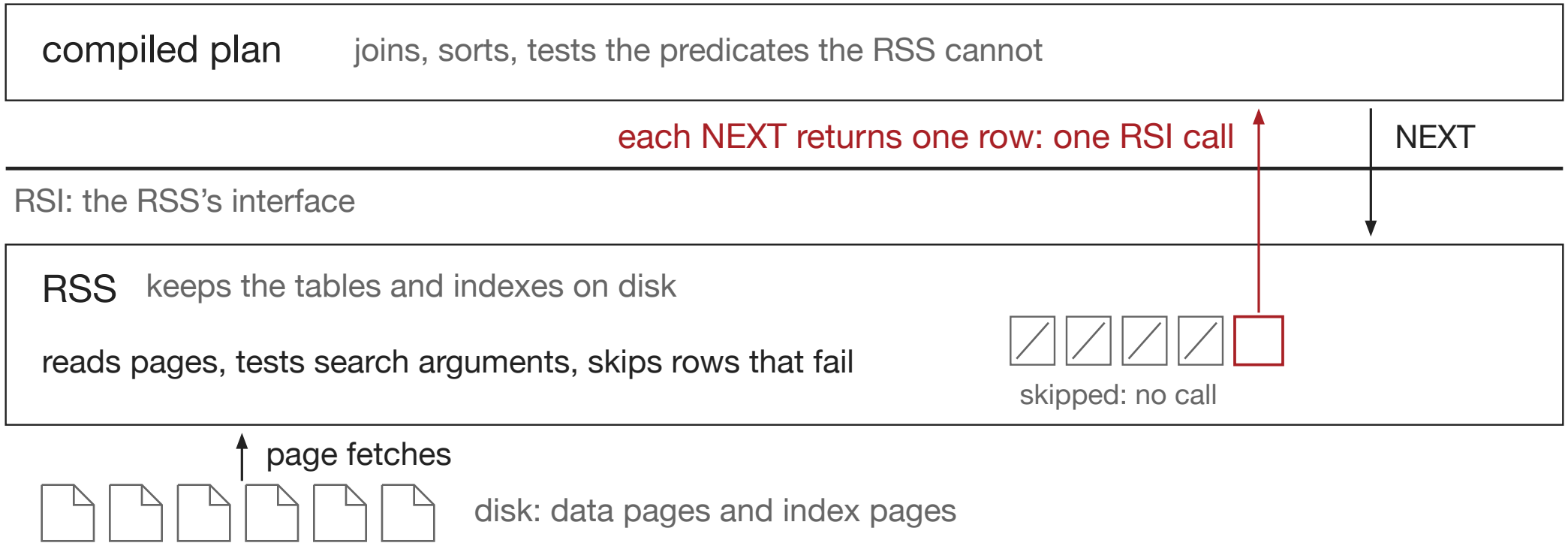
Pages that fit in the buffer are fetched once, so Table 2 charges pages, not rows: 135 beats 500.



The buffer-fit formula is the clustered formula: one rectangle changes it.

Appendix: the CPU term

System R adds a CPU term: `COST = PAGE FETCHES + W × RSI CALLS`, W a machine weight.



W prices a call in page fetches (say 10 ms a fetch, 0.1 ms a call: W = 0.01); no real value is published.

On one table every path passes up the same rows, so the term is the same for all. We count pages only.

Appendix: every single-table path

T	Path	Arithmetic	Pages
EMP	scan · DNO clustered · JOB · SAL	`500` · `50+500` · `50+10,000` · `40+10,000`	**500** · **550** · **10,050** · 10,040
DEPT	scan · DNO clustered unique · LOC matching	`5` · `2+5` · `1/10 × (2+100)`	**5** · **7** · 10.2
JOB	TITLE unique equality · JOB clustered · scan	`1+1` · `1+2` · `2`	**2** · **3** · 2, a tie

Bold costs are kept: the cheapest path, and the cheapest in each useful order.

Ten paths, seven kept. No EMP index matches: EMP has no predicate.

JOB’s scan ties its TITLE index at 2, both unordered; the search keeps one, the index.

Appendix: every join plan priced

Set	Kept, one per order	Priced, then pruned
{E,D}	**60** · **62** DNO · **30,050** JOB	557 · 1,507 · 20,500 · 20,550
{E,J}	**203** · **204** JOB · **20,550** DNO	1,503 · 10,053 · 20,500 · 30,050
{E,D,J}	**163** = 60 + 100 + 3	230 · 603 · 2,060 · 20,557 · 30,053

Fourteen pair plans and six triple plans, each `C = C_outer + outer_rows × C_inner` over the single-table paths or a sorted input; `outer_rows` is 10 for DEPT, 1 for JOB, 10,000 for EMP.

Lookups: EMP by DNO `1/100 × (50+500) = 5.5` · unique index `1 + 1 = 2` ·
EMP by JOB `1/50 × (50+10,000) = 201`.

A sort of Q rows adds 2 × ceil(Q / 20). Twenty costs, three subsets, one plan. All illustrative.

Appendix: DP against the heuristics

Every algorithm plans from PostgreSQL's estimates, then runs for real, so even exhaustive search lands at 1.42×: it optimizes the wrong objective more thoroughly. PK and FK indexes.

Algorithm	Median	95th	Max
Dynamic programming	**1.42**	**15.3**	**35.3**
Quickpick: best of 1,000 orders	2.10	77.6	3,303
Greedy, minimum cardinality	1.84	34.1	766
Greedy, minimum cost	2.04	20.3	136

Exhaustive search wins in the tail, which is where the timeouts live.

VLDB Journal 2018, Table 4. With primary keys only, a greedy median of 1.01 beats DP's 1.03.

Appendix: order equivalence classes



one order equivalence class

a plan sorted on any one of them is sorted on all three

E.DNO = D.DNO and D.DNO = F.DNO put all three columns in one class.

Exact, not a heuristic: equality is transitive. F is a third table; our query has two classes.

Appendix: what 1979 did not have

Missing in 1979	When it arrived
Hash join	GRACE hash join, 1983; hybrid hash, DeWitt et al. 1984
Histograms	Equi-width, Kooi 1980; equi-depth, Piatetsky-Shapiro 1984
Bushy plans	Postgres by 9.4; one major commercial engine only in 2014
Parallel plans	Volcano’s exchange operator, 1990; Postgres 9.6, 2016
Re-optimization	Kabra and DeWitt 1998; LEO 2001; shipped only per operator
Stale-plan checks	Postgres 8.3, 2008: re-plans a cached query on new statistics

Research dates are first publication. Postgres 8.3 and 9.6 are the releases that shipped the feature; 9.4 is only the release Leis tested, not the one that added bushy plans.

Appendix: questions to review

1. Give a query where “use the index” is wrong. Which two statistics say so?
2. Clustered or not: which statistic switches from pages to rows, and why?
3. Why can the best plan for {EMP, DEPT, JOB} be built from the best plans for its 2-subsets? Where does that need “per useful order”?
4. Correlated LOC and ZIP, factors multiplied: over- or underestimate?
5. Leis 2015: if the cost model is crude and the estimates are bad, why does exhaustive DP still help?

Five of the thirteen review questions at the end of the lecture notes.