

DB 2

14 – Query Optimization

Summer 2018

Torsten Grust
Universität Tübingen, Germany

1 : One Query — Millions of Plans

Q: Given a SQL query Q , what is *the optimal* (a reasonable)¹ plan to evaluate it? — **A:** It depends:

- Can we **simplify** (flatten, unnest) Q ?
- How can we **access the tables** referenced in Q ?
- How do **CPU and (sequential, random) I/O cost** compare?
- What is **selectivity of the predicates** used in Q ?
- Which plan **operator implementations** are applicable?
- Can we **regroup/reorder the joins** in Q ?

¹ Here: focus on reducing the overall query evaluation time. The optimum is, generally, not reached.

Excerpt of the TPC-H Benchmark (at Scale Factor SF)

<u>o_orderkey</u>	<u>o_custkey</u>	<u>o_totalprice</u>	<u>o_clerk</u>	...
o	c			

orders ($\approx SF \times 1.5 \times 10^6$ rows)

<u>l_orderkey</u>	<u>l_linenum</u>	<u>l_partkey</u>	<u>l_quantity</u>	<u>l_extendedprice</u>	...
o					

lineitem ($\approx SF \times 6 \times 10^6$ rows)

<u>c_custkey</u>	<u>c_name</u>	<u>c_acctbal</u>	<u>c_nationkey</u>	...
c			n	

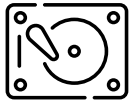
customer ($\approx SF \times 150000$ rows)

<u>n_nationkey</u>	<u>n_name</u>	<u>n_regionkey</u>	...
n		r	

nation (25 rows)

<u>r_regionkey</u>	<u>r_name</u>	...
r		

region (5 rows)



Q₁₄: Three-Way Join Against a TPC-H Instance

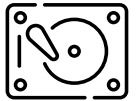
Price and quantity of parts ordered by customer #001:

```

SELECT l.l_partkey, l.l_quantity, l.l_extendedprice
FROM   lineitem AS l JOIN orders AS o      -- { l ⋈ o
      ON (l.l_orderkey = o.o_orderkey)    -- }
      JOIN customer AS c                  -- } ⋈ c
      ON (o.o_custkey = c.c_custkey)      -- }
WHERE  c.c_name = 'Customer#001';

```

- Above SQL syntax suggests the **join order** $(l \bowtie o) \bowtie c$.
- Commutativity and associativity of \bowtie enable the RDBMS to **reorder** the joins—based on *estimated evaluation costs*.
 - ... unless we insist on the syntactic order. 🤖



2 | Pre-Processing: Query Normalization

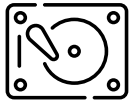
Transform the input SQL query such that it features **SELECT-FROM-WHERE** (SFW) blocks of the following shape:

```

SELECT [ DISTINCT ] e, ..., e
FROM     $\Delta$ , ...,  $\Delta$            --  $\Delta$   $\equiv$  base table or (query)
[ WHERE  p AND ... AND p ]   -- p  $\equiv$  predicate in DNF
[ GROUP BY g, ..., g           -- }
[ HAVING  p AND ... AND p ] ] -- } e, p, g, o  $\equiv$ 
[ ORDER BY o, ..., o ]       -- } atomic expression or
[ OFFSET  n ]                 -- } scalar (subquery)
[ LIMIT   m ]                 -- } n, m  $\equiv$  integer literal

```

- Query clauses in [...] may be missing.

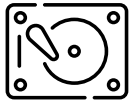


3 | Pre-Processing: Query Unnesting

Nested SQL queries suggest a (naïve, inefficient) nested-loop-style evaluation strategy. Consider:

<pre> SELECT c.c_name FROM customer AS c, ⚠ { (SELECT n.n_nationkey, n.n_name \ FROM nation AS n) AS t WHERE c.c_nationkey = t.n_nationkey AND strpos(c.c_address, t.n_name) > 0 </pre>	1 :	<pre> SELECT o.o_orderkey FROM orders AS o WHERE o.o_custkey IN ⚠ { (SELECT c.c_custkey FROM customer AS c WHERE c.c_name = '...') </pre>	2
---	-------	--	---

- 💡 If possible, **unnest** ⚠ queries and “inline” into parent query \Rightarrow ⚠ can participate in join reordering.



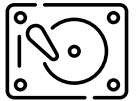
Pre-Processing: Query Unnesting

Perform **query unnesting** on the level of

- the operator-based plan representation of the query,² or
- the internal AST representation of SQL. Re ②:

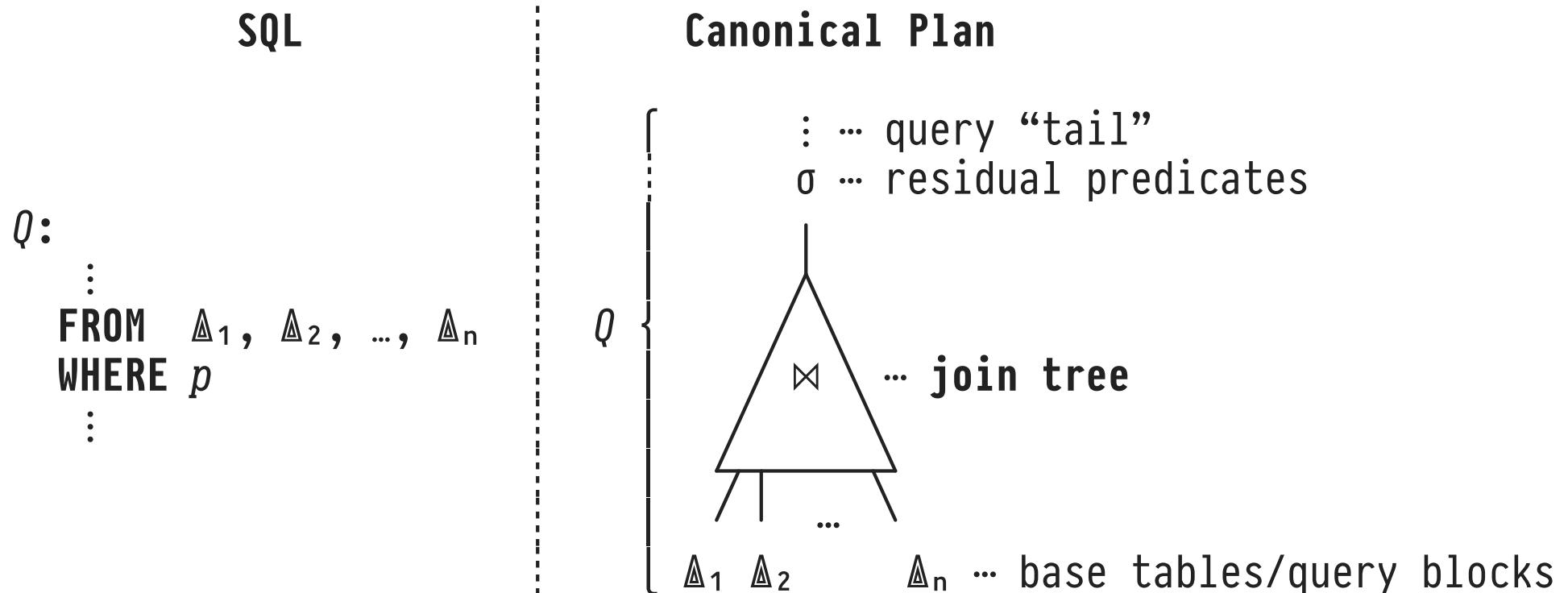
<pre> SELECT e₁ FROM q₁, ..., q_i WHERE p₁ AND e₂ IN (SELECT e₃ FROM q_{i+1}, ..., q_n WHERE p₃) </pre>	⇒	<pre> SELECT DISTINCT e₁ FROM q₁, ..., q_i, q_{i+1}, ..., q_n WHERE p₁ AND e₂ = e₃ AND p₃ </pre>
---	---	---

² See *Unnesting Arbitrary Queries*, Thomas Neumann, Alfons Kemper. BTW 2015, Hamburg, Germany.



4 : Join Tree Optimization

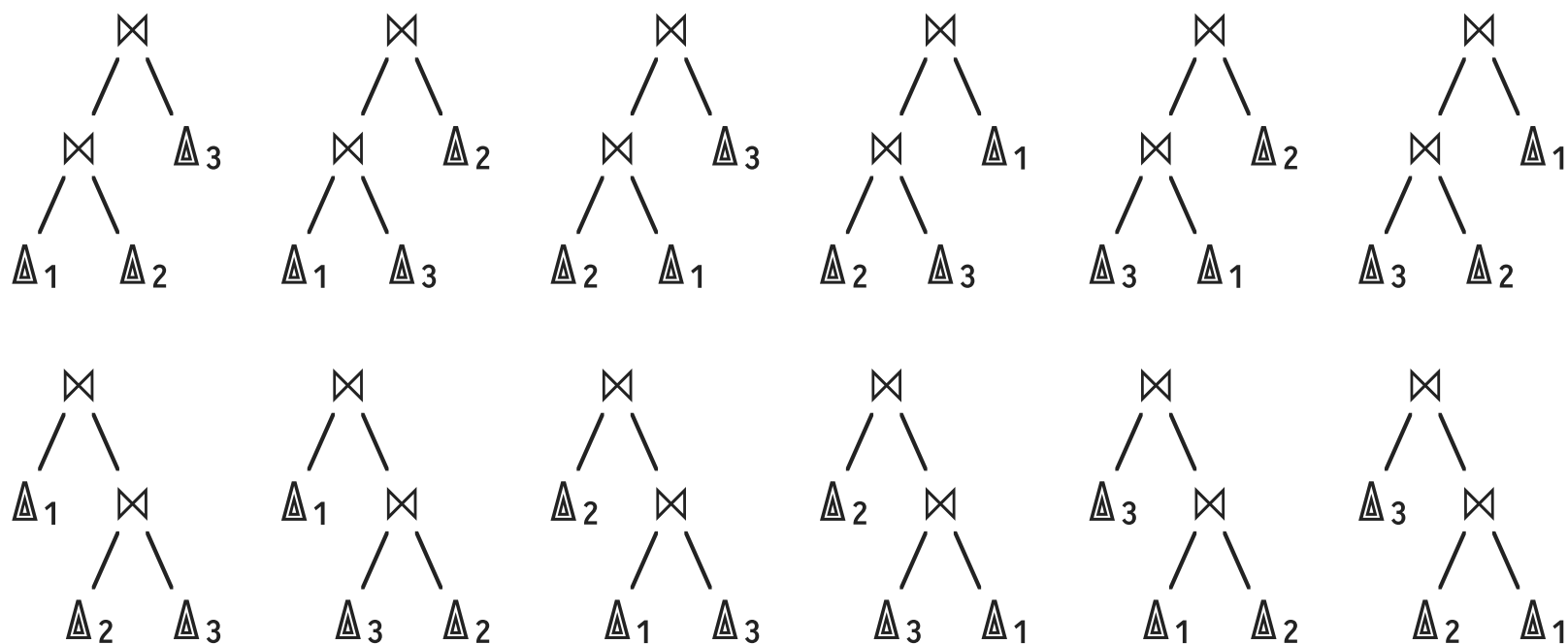
Processing a SQL query Q starts out with its **FROM** and **WHERE** clauses which describe a **join tree** over Q 's inputs:



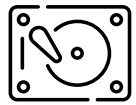


Join Tree Optimization

Given n join inputs, the number of possible **join tree shapes** is *huge*. Consider $n = 3$:

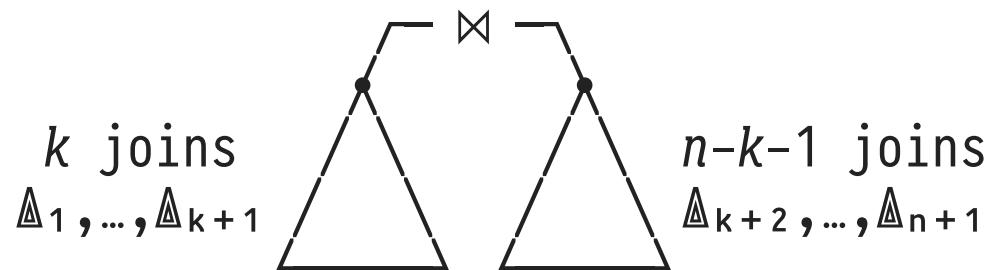


- Shapes based on associativity and commutativity of \bowtie .



How Many Possible Join Trees are There?

1. A join of $n+1$ inputs Δ requires n binary joins. The root \bowtie combines subtrees of k and $n-k-1$ joins ($0 \leq k \leq n-1$):³

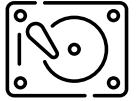


of join tree shapes:

$$C_n = \sum_{k=0}^{n-1} C_k \times C_{n-k-1}$$

2. Orderings of the Δ at the join tree leaf level: $(n+1)!$.
3. Join algorithm choices (a available algorithms): a^n .

³ C_n are the *Catalan numbers*, the number of ordered binary trees with $n+1$ leaves. $C_0 = 1$.

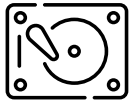


How Many Possible Join Trees are There?


Number of possible join trees given n binary joins with $a = 3$ implementation choices:

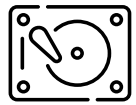
# of $\triangle (n+1)$	C_n	# of join trees
2	1	6
3	2	108
4	5	3240
5	14	136080
6	42	7384320
7	132	484989120
8	429	37829151360
9	1430	3404623622400
10	4862	347271609484800

- A search space of this size is impossible to fully explore for any query optimizer.



Join Plan Generation Through Dynamic Programming

- **Problem:** Find optimal query plan $opt[\{\Delta_1, \dots, \Delta_n\}]$ that joins n inputs $\Delta_1, \dots, \Delta_n$.
 1. **Iteration 1:** For each Δ_j , find and memorize **best 1-input plan** $opt[\{\Delta_j\}]$ that accesses Δ_j only.
 2. **Iteration $k > 1$:** Find and memorize **best k -input plans** that join $k \leq n$ inputs by combining (for $1 \leq i < k$)
 - the best i -input plans and \setminus simple lookups in
 - the best $(k-i)$ -input plans. $\int opt[\cdot]$ memo 



Bottom-Up Dynamic Programming ($n = 3$)

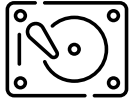
k **Possible k -input Access/Join Plans** if Δ_i is complex

1 $opt[\{\Delta_1\}] \leftarrow prune(\{Seq\ Scan\ \Delta_1, Index\ Scan\ \Delta_1, Bitmap\ Scan\ \Delta_1, \overbrace{\Delta_1}\})$
 $opt[\{\Delta_2\}] \leftarrow prune(\{Seq\ Scan\ \Delta_2, Index\ Scan\ \Delta_2, Bitmap\ Scan\ \Delta_2, \Delta_2\})$
 $opt[\{\Delta_3\}] \leftarrow prune(\{Seq\ Scan\ \Delta_3, Index\ Scan\ \Delta_3, Bitmap\ Scan\ \Delta_3, \Delta_3\})$

2 $opt[\{\Delta_1, \Delta_2\}] \leftarrow prune(opt[\{\Delta_1\}] \otimes opt[\{\Delta_2\}])$
 $opt[\{\Delta_1, \Delta_3\}] \leftarrow prune(opt[\{\Delta_1\}] \otimes opt[\{\Delta_3\}])$
 $opt[\{\Delta_2, \Delta_3\}] \leftarrow prune(opt[\{\Delta_2\}] \otimes opt[\{\Delta_3\}])$

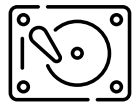
3 $opt[\{\Delta_1, \Delta_2, \Delta_3\}] \leftarrow prune($
 $\quad opt[\{\Delta_1\}] \otimes opt[\{\Delta_2, \Delta_3\}] \cup$
 $\quad opt[\{\Delta_2\}] \otimes opt[\{\Delta_1, \Delta_3\}] \cup$
 $\quad opt[\{\Delta_3\}] \otimes opt[\{\Delta_1, \Delta_2\}] \quad)$

$prune(P) \equiv$ best (= minimal cost + interestingly ordered) plans in set P
 $l \otimes r \equiv \{l \bowtie^{n1} r, r \bowtie^{n1} l, l \bowtie^{mj} r, r \bowtie^{mj} l, l \bowtie^{hj} r, r \bowtie^{hj} l\}$



Join Plan Generation (Notes)

- **Access plan choices** (*access*(·)):
 - Consider sequential/index scans if Δ is a base table, otherwise simply consume Δ 's rows.
- **Join plan choices** ($_ \otimes _$):
 - Considers all viable join algorithms (given θ , available indexes, ...) and left/right input orders.
- **Principle of Optimality** (*prune*(·)): A globally optimal plan is built from optimal subplans. Thus:
 - 💡 For each subset of $\{\Delta_1, \dots, \Delta_n\}$, memorize in *opt*[·]
 1. ... its overall best plan and
 2. ... its best plan satisfying each **interesting order**.



(Bushy) Join Plan Generation: Pseudo Code

```

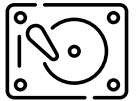
JoinPlan( $\{\Delta_1, \dots, \Delta_n\}$ ):
  foreach  $p \in \{\Delta_1, \dots, \Delta_n\}$  } 1-input plans
  [  $opt[\{p\}] \leftarrow prune(access(p))$ ;

  for  $k$  in  $2, \dots, n$  }  $k$ -input plans
  [ foreach  $S \subseteq \{\Delta_1, \dots, \Delta_n\}$  with  $|S| = k$  } enumerate subsets
  [ [  $opt[S] \leftarrow \phi$ ;
    [ foreach  $T \subset S$  with  $T \neq \phi$   $\Join^a \setminus$ 
    [  $opt[S] \leftarrow opt[S] \cup \{opt[T] \Join^a opt[S \setminus T]\}$ ;
    [  $opt[S] \leftarrow prune(opt[S])$ ;

  return  $opt[\{\Delta_1, \dots, \Delta_n\}]$ ;

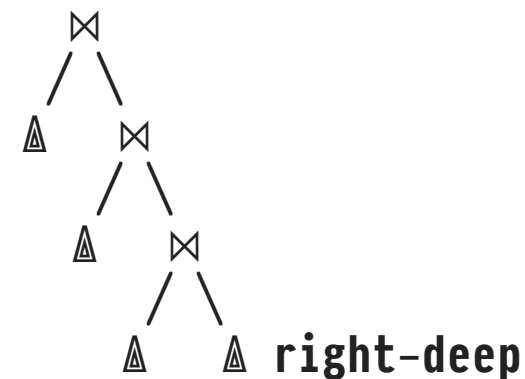
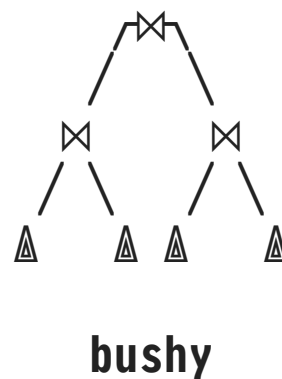
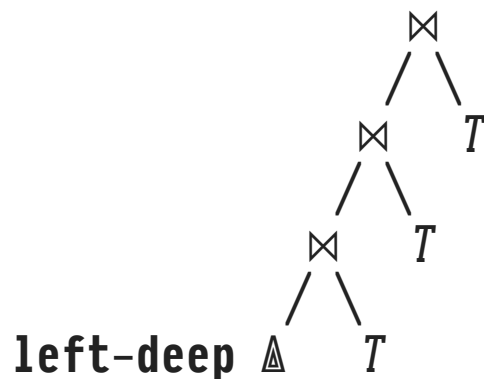
```

- $access(\cdot)$, $prune(\cdot)$ defined as above,
 $\Join^a \setminus$ builds all join algorithm choices ($a \in \{nl, mj, hj\}$).

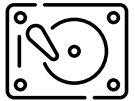


Reducing the Search Space

- Avoid generating costly **Cartesian products**: don't form joins between inputs w/o join predicate (`_ θ _ = true`).
- Generate **left-deep** join plans only: right join input (NL⋈: inner input) is a scan over base table *T*.
 - Admits use of Index Nested Loop Join.
 - Straightforward Volcano-style execution (reset inner).



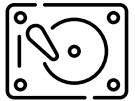
5 | Estimating Plan Cost



The query optimizer explores the vast plan search space to find the **optimal** (“best”, “cheapest”) plan.

- Typically, RDBMSs measure **plan cost** in terms of *total execution time* (time until last result row delivered).
- These total plan costs are **estimated before** plan execution begins (`EXPLAIN: ... cost=c1..c2← ...`).
- A **cost model**—measured in abstract “space\$”—reflects the true costs (measured in *ms*, CPU time, # I/O ops, ...) of plans p_1 , p_2 :

$$\text{space}\$(p_1) < \text{space}\$(p_2) \Rightarrow \text{true cost}(p_1) < \text{true cost}(p_2)$$



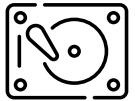
PostgreSQL: Plan Cost

EXPLAIN shows estimated costs (unit: space\$) and cardinalities (# of rows):

QUERY PLAN	
startup cost	total cost
Hash Join	(cost=299.00..15443.31 rows=505183 width=50)
⋮	↑ cardinality

- **run cost** $\stackrel{\text{def}}{=} \text{total cost} - \text{startup cost}$ ⁴ (not shown).
- Optimizer decisions are based on estimated **total cost**.

⁴ To implement `set enable_<op> = off`, PostgreSQL sets the operator's **startup cost** to 10^9 ($\equiv \infty$).



Cost Model Configuration

Model Configuration	Default	Description
<code>seq_page_cost</code>	1.0	I/O cost of one sequential page access
<code>random_page_cost</code>	4.0	I/O cost of one random page access
<code>cpu_tuple_cost</code>	0.01	CPU cost to process a heap file row
<code>cpu_index_tuple_cost</code>	0.005	CPU cost to process an index leaf entry
<code>cpu_operator_cost</code>	0.0025	CPU function/operator evaluation cost
<code>parallel_tuple_cost</code>	0.1	Cost of passing one row worker→leader
<code>parallel_setup_cost</code>	1000.0	Shared memory setup cost

- Parameters are configurable:
 - Seek cost, thus `random_page_cost` » `seq_page_cost`. But...
 - ... if DB fits in RAM, `random_page_cost` = `seq_page_cost` may be more appropriate.

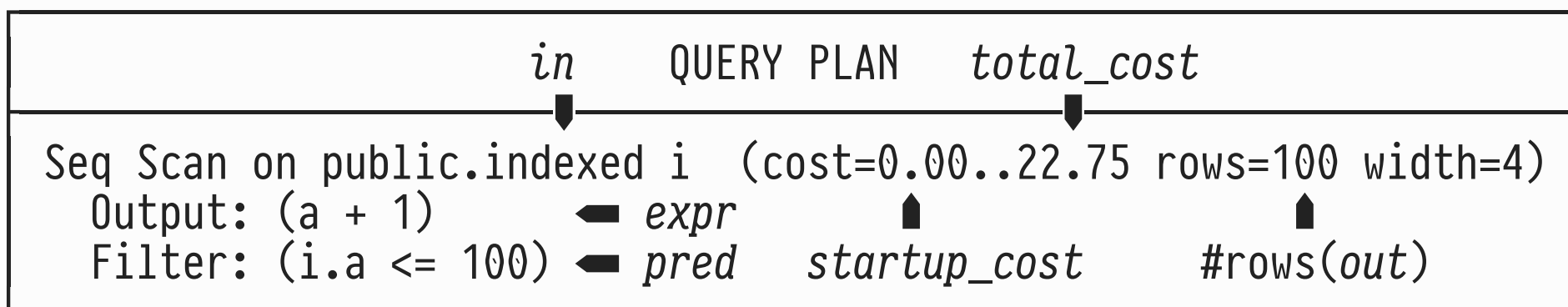


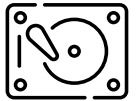
Cost of Seq Scan ①

Given an occurrence of **Seq Scan** with arguments

- *in*: input table,
- *pred*: (optional) filter predicate on *in*,
- *expr*: **SELECT** clause expression(s),

how does PostgreSQL derive *startup_cost* and *total_cost*?





Cost of Seq Scan ②

Cost calculation depends on the following parameters, mostly available in PostgreSQL's internal `pg_*` meta data tables:

Parameter	Description	Available as...
<code>#rows(<i>in</i>)</code>	# rows (cardinality) of table <i>in</i>	<code>pg_class.reltuples</code>
<code>#pages(<i>in</i>)</code>	# pages in heap file of <i>in</i>	<code>pg_class.relpages</code>
<code>sel(<i>pred</i>)</code>	selectivity of filter <i>pred</i> ⁵	see below

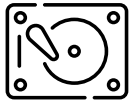
- Meta data like `#rows(in)`, `#pages(in)` and others are updated whenever the system performs an `ANALYZE` run on table *in*.
- Predicate selectivity `sel(pred)` is estimated based on sampled table data and the syntactic structure of *pred*.

⁵ `sel(pred)` \in $\{0, \dots, 1\}$ with `sel(pred)` = 0 \equiv no row satisfies filter *pred*.



Cost of Seq Scan ③

$$\begin{aligned}
 \text{startup_cost} &\stackrel{\text{def}}{=} \overset{\text{typically} = 0}{\text{startup_cost}(\text{pred})} + \text{startup_cost}(\text{expr}) \\
 \\
 \text{cpu_run_cost} &\stackrel{\text{def}}{=} \underbrace{\#rows(\text{in}) \times (\text{cpu_tuple_cost} + \text{run_cost}(\text{pred}))}_{\text{decode heap row}} + \underbrace{\#rows(\text{in}) \times \text{sel}(\text{pred}) \times \text{run_cost}(\text{expr})}_{\text{evaluate filter}} \\
 &= \underbrace{\#rows(\text{out})}_{\text{evaluate SELECT clause}} \\
 \\
 \text{disk_run_cost} &\stackrel{\text{def}}{=} \underbrace{\#pages(\text{in}) \times \text{seq_page_cost}}_{\text{sequentially read entire input heap file}} \\
 \\
 \text{total_cost} &\stackrel{\text{def}}{=} \text{startup_cost} + \underbrace{\text{cpu_run_cost} + \text{disk_run_cost}}_{= \text{run_cost}}
 \end{aligned}$$

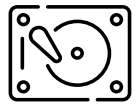


Cost of Index Scan ②

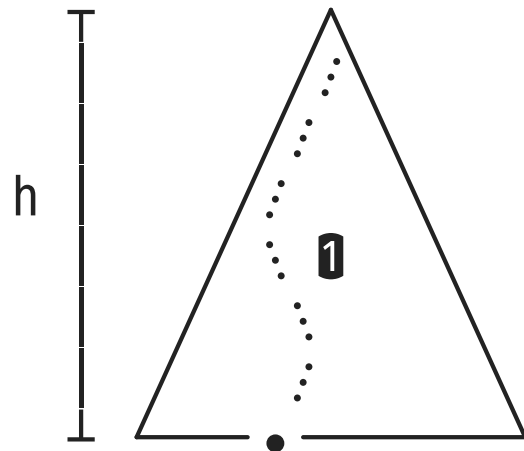
Cost model parameters:

Parameter	Description	Available as...
<code>#rows(<i>in</i>)</code>	# rows (cardinality) of table <i>in</i>	<code>pg_class.reltuples</code>
<code>#pages(<i>in</i>)</code>	# pages in heap file of <i>in</i>	<code>pg_class.relpages</code>
<code>sel(<i>pred</i>)</code>	selectivity of filter <i>pred</i>	see below
<code>h(<i>idx</i>)</code>	height of B+Tree <i>idx</i>	<code>bt_metap(.)</code>
<code>#rows(<i>idx</i>)</code>	# leaf entries in index <i>idx</i>	<code>pg_class.reltuples</code>
<code>#pages(<i>idx</i>)</code>	# pages in leaf level of <i>idx</i>	<code>pg_class.relpages</code>
<code>corr(<i>idx</i>)</code>	≈ clustering factor for index <i>idx</i>	<code>pg_stats.correlation</code>

- `corr(idx)` $\in \{-1.0, \dots, 1.0\}$ characterizes how much the physical orderings of index leaves and heap file deviate.
 - After `CLUSTER in ON idx`, we have `corr(idx) = 1.0`.



Cost of Index Scan ③ (B+Tree Descent)



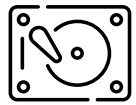
- B+Tree height $h = \log_{2 \times o}(\#rows(idx))$

⇒ # of key comparisons during B+Tree descent **1**:

$$\underbrace{\lceil \log_2(2 \times o) \times h \rceil}_{\text{binary search in inner B+Tree node with fan-out } F = 2 \times o} = \lceil \log_2(\#rows(idx)) \rceil$$

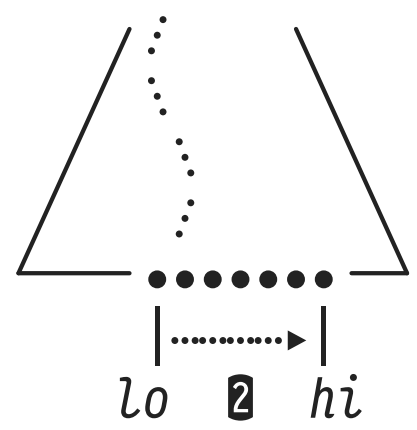
binary search in inner B+Tree
node with fan-out $F = 2 \times o$

$$\begin{aligned} \text{startup_cost} &\stackrel{\text{def}}{=} \text{startup_cost}(pred) + \text{startup_cost}(expr) \\ &+ \underbrace{\lceil \log_2(\#rows(idx)) \rceil}_{\text{B+Tree descent}} + \underbrace{(h + 1)}_{\text{: + •}} \times \underbrace{50}_{\text{index node processing}} \times \text{cpu_operator_cost} \end{aligned}$$



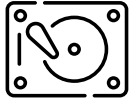
Cost of Index Scan ④ (Leaf Level Scan)

The index leaf level (sequence set) scan ② incurs CPU as well as I/O cost that contribute to the overall **run_cost**:



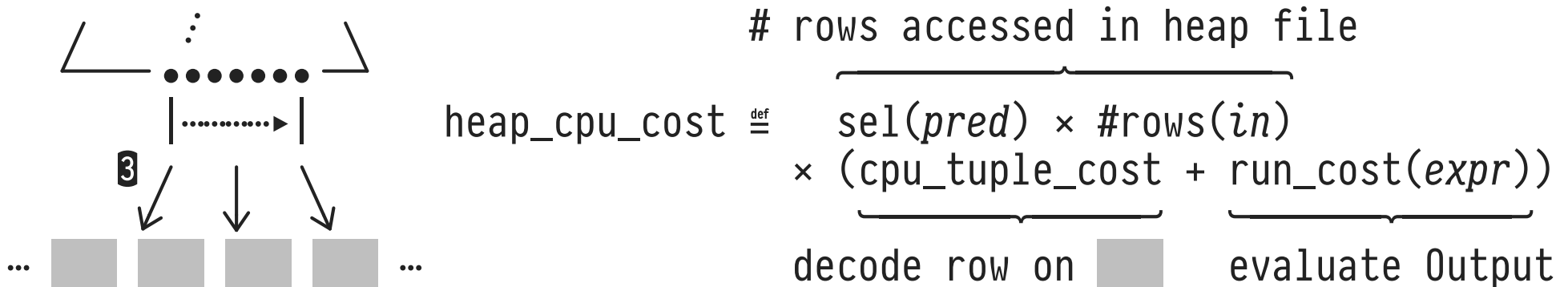
$$\text{index_cpu_cost} \stackrel{\text{def}}{=} \underbrace{\text{sel}(pred) \times \#rows(idx)}_{\text{\# rows in scanned range } |\dots\rangle|} \times \underbrace{(\text{cpu_index_tuple_cost} + \text{run_cost}(pred))}_{\substack{\text{decode index leaf entry} \quad \text{evaluate } \leq hi}}$$

$$\text{index_I0_cost} \stackrel{\text{def}}{=} \underbrace{[\text{sel}(pred) \times \#pages(idx)]}_{\text{\# of pages } \bullet \text{ in scanned range}} \times \underbrace{\text{random_page_cost}}_{\text{B+Tree leaves not clustered}}$$



Cost of Index Scan ⑤ (Heap File Access)

Unless we perform an **Index Only Scan**, we additionally pay CPU and I/O costs for heap file accesses ③:

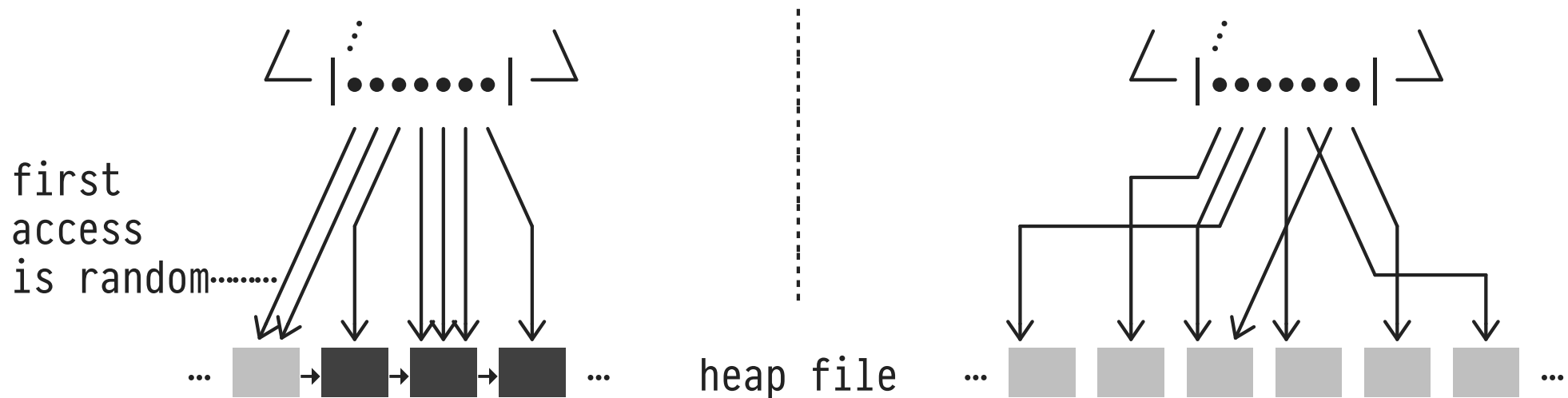


- The more **clustered** the index, the cheaper the heap I/O. Linearly interpolate between the clustered and non-clustered scenarios:

$$\text{heap_IO_cost} \stackrel{\text{def}}{=} \text{unclustered_IO_cost} + \text{corr}(idx)^2 \times (\text{clustered_IO_cost} - \text{unclustered_IO_cost})$$

$\uparrow \approx \text{clustering factor} \in \{0, \dots, 1\}$

Cost of Index Scan ⑥ ([Non-]Clustered Heap File Access)



$$\text{clustered_IO_cost} \stackrel{\text{def}}{=} 1 \times \text{random_page_cost} (\blacksquare) + (\text{sel}(\text{pred}) \times \#\text{pages}(\text{in}) - 1) \times \text{seq_page_cost} (\blacksquare)$$

$$\text{unclustered_IO_cost} \stackrel{\text{def}}{=} \#\text{pages}(\text{in}) \times \text{random_page_cost} (\blacksquare)$$

$$\text{total_cost} \stackrel{\text{def}}{=} \text{startup_cost} + \text{index_cpu_cost} + \text{index_IO_cost} + \text{heap_cpu_cost} + \text{heap_IO_cost}$$