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*)

o_orderkey	o_custkey	o_totalprice	o_clerk	…
o	*c*			

orders (≈ $SF \times 1.5\times10^6$ rows)

l_orderkey	l_linenumber	l_partkey	l_quantity	l_extendedprice	…
o					

lineitem (≈ $SF \times 6\times10^6$ rows)

c_custkey	c_name	c_acctbal	c_nationkey	…
c			*n*	

customer (≈ $SF \times 150000$ rows)

n_nationkey	n_name	n_regionkey	…
n		*r*	

nation (25 rows)

r_regionkey	r_name	…
r		

region (5 rows)

Q_{14}: Three-Way Join Against a TPC-H Instance

Price and quantity of parts orderd 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 ⋈ o) ⋈ c.
- Commutativity and associativity of ⋈ 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     𝔸, …, 𝔸                       -- 𝔸 ≡ base table or (query)
[ WHERE    p AND … AND p ]               -- p ≡ predicate in DNF
[ GROUP BY g, …, g                       -- ⎫ e, p, g, o ≡
 [ HAVING  p AND … AND p ] ]             -- ⎬   atomic expression or
[ ORDER BY o, …, o ]                     -- ⎭   scalar (subquery)
[ OFFSET   n ]                           -- ⎫ n, m ≡ integer literal
[ LIMIT    m ]                           -- ⎭
```

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

3 ⁞ Pre-Processing: Query Unnesting

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

```
SELECT c.c_name                                  ❶
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
```

```
SELECT o.o_orderkey          ❷
FROM   orders AS o
WHERE  o.o_custkey IN
     ⎧ (SELECT c.c_custkey
⍋    ⎨  FROM   customer AS c
     ⎩  WHERE  c.c_name = '…')
```

- 💡 If possible, **unnest** ⍋ queries and “inline” into parent query ⇒ ⍋ 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 ❷:

```
SELECT $e_1$                                        SELECT DISTINCT $e_1$
FROM   $q_1$,…,$q_i$                                FROM   $q_1$,…,$q_i$,$q_{i+1}$,…,$q_n$
WHERE  $p_1$                                   ⇛    WHERE  $p_1$
  AND  $e_2$ IN (SELECT $e_3$                         AND  $e_2$ = $e_3$
                 FROM   $q_{i+1}$,…,$q_n$             AND  $p_3$
                 WHERE  $p_3$)
```

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

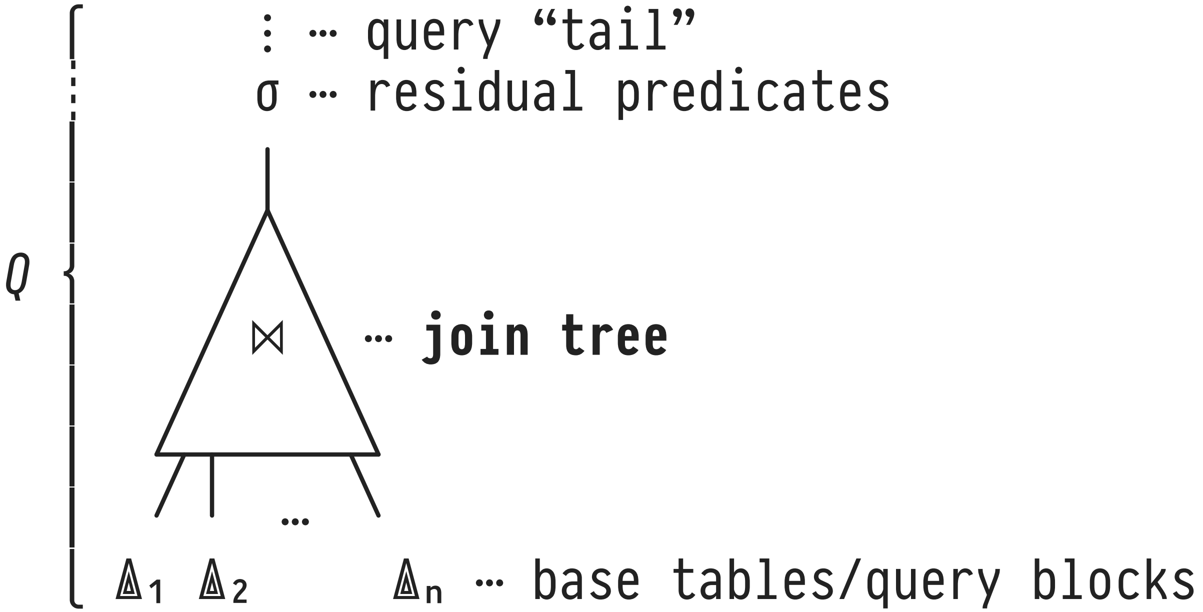
4 ¦ Join Tree Optimization

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

SQL

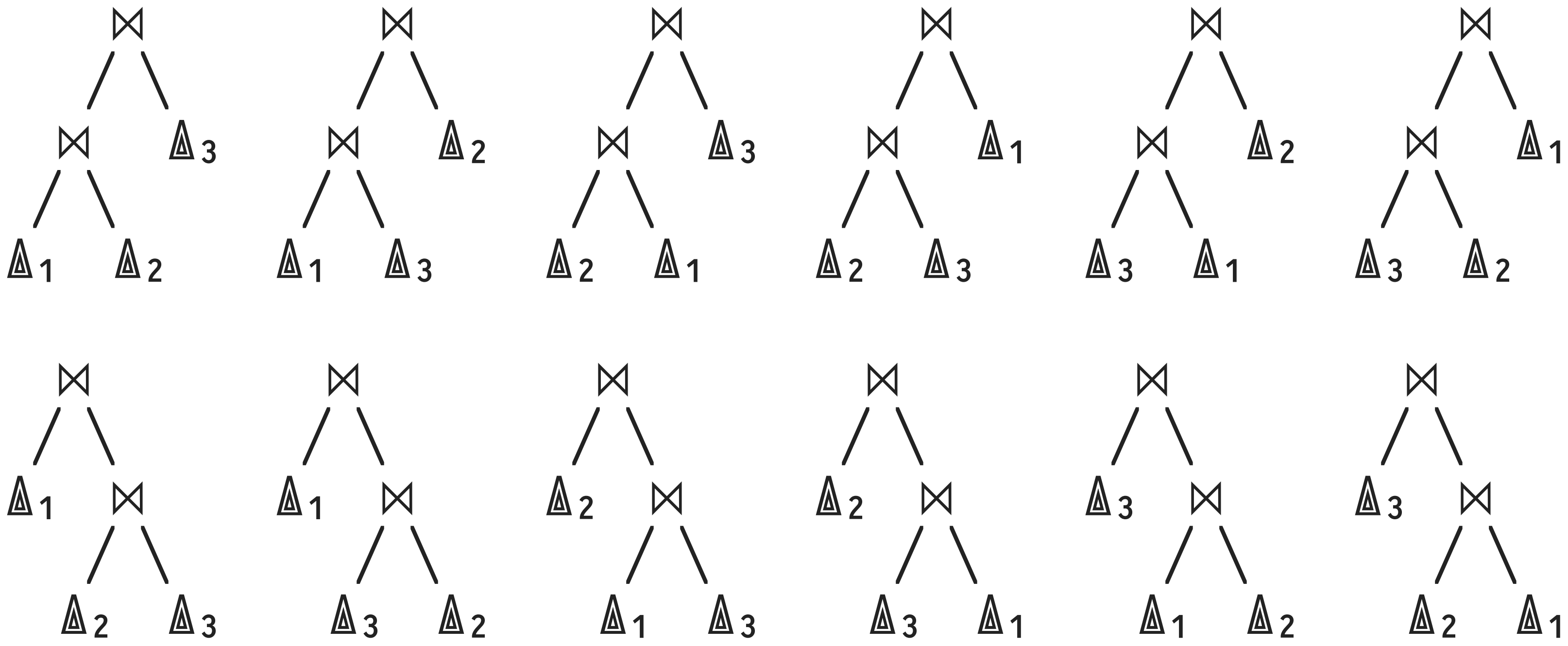
Q:
```
  ⋮
FROM  ⟁₁, ⟁₂, …, ⟁ₙ
WHERE p
  ⋮
```

Canonical Plan


⋮ ··· query “tail”
σ ··· residual predicates
Q
⋈
··· **join tree**
···
⟁$_1$ ⟁$_2$ ⟁$_n$ ··· base tables/query blocks

Join Tree Optimization

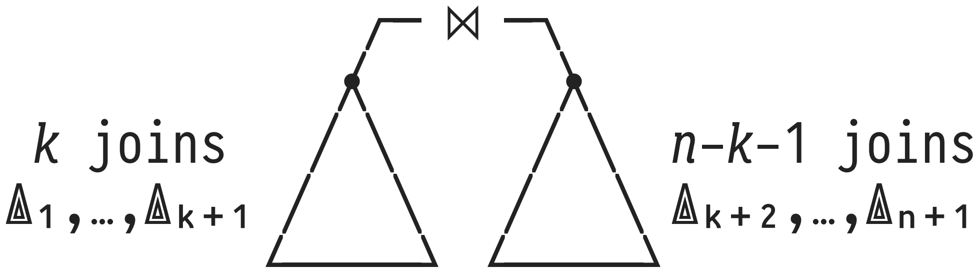
Given n join inputs, the number of possible **join tree shapes** is *huge*. Consider $n = 3$:



- Shapes based on associativity and commutativity of ⋈.

How Many Possible Join Trees are There?

1. A join of $n+1$ inputs △ requires n binary joins. The root ⋈ combines subtrees of k and $n-k-1$ joins $(0 \leqslant k \leqslant n-1)$:[3]



# of join tree shapes:

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

2. Orderings of the △ 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 ◬ $(n+1)$	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 $opt[\{⟁_1,\ldots,⟁_n\}]$ that joins n inputs $⟁_1,\ldots,⟁_n$.

1. **Iteration 1:** For each $⟁_j$, find and memorize **best 1-input plan** $opt[\{⟁_j\}]$ that accesses $⟁_j$ only.

2. **Iteration k > 1:** Find and memorize **best k-input plans** that join $k \leqslant n$ inputs by combining (for $1 \leqslant i < k$)
   - the best i-input plans and ⎫ simple lookups in
   - the best $(k-i)$-input plans. ⎭ $opt[\cdot]$ memo 👍

Bottom-Up Dynamic Programming (n = 3)

k	Possible k-input Access/Join Plans
1	$\mathit{opt}[\{\mathbb{A}_1\}] \leftarrow \mathit{prune}(\{\text{Seq Scan } \mathbb{A}_1, \text{ Index Scan } \mathbb{A}_1, \text{ Bitmap Scan } \mathbb{A}_1, \overbrace{\mathbb{A}_1}^{\text{if } \mathbb{A}_i \text{ is complex}}\})$ $\mathit{opt}[\{\mathbb{A}_2\}] \leftarrow \mathit{prune}(\{\text{Seq Scan } \mathbb{A}_2, \text{ Index Scan } \mathbb{A}_2, \text{ Bitmap Scan } \mathbb{A}_2, \mathbb{A}_2\})$ $\mathit{opt}[\{\mathbb{A}_3\}] \leftarrow \mathit{prune}(\{\text{Seq Scan } \mathbb{A}_3, \text{ Index Scan } \mathbb{A}_3, \text{ Bitmap Scan } \mathbb{A}_3, \mathbb{A}_3\})$
2	$\mathit{opt}[\{\mathbb{A}_1,\mathbb{A}_2\}] \leftarrow \mathit{prune}(\mathit{opt}[\{\mathbb{A}_1\}] \circledast \mathit{opt}[\{\mathbb{A}_2\}])$ $\mathit{opt}[\{\mathbb{A}_1,\mathbb{A}_3\}] \leftarrow \mathit{prune}(\mathit{opt}[\{\mathbb{A}_1\}] \circledast \mathit{opt}[\{\mathbb{A}_3\}])$ $\mathit{opt}[\{\mathbb{A}_2,\mathbb{A}_3\}] \leftarrow \mathit{prune}(\mathit{opt}[\{\mathbb{A}_2\}] \circledast \mathit{opt}[\{\mathbb{A}_3\}])$
3	$\mathit{opt}[\{\mathbb{A}_1,\mathbb{A}_2,\mathbb{A}_3\}] \leftarrow \mathit{prune}(\mathit{opt}[\{\mathbb{A}_1\}] \circledast \mathit{opt}[\{\mathbb{A}_2,\mathbb{A}_3\}] \cup$ $\mathit{opt}[\{\mathbb{A}_2\}] \circledast \mathit{opt}[\{\mathbb{A}_1,\mathbb{A}_3\}] \cup$ $\mathit{opt}[\{\mathbb{A}_3\}] \circledast \mathit{opt}[\{\mathbb{A}_1,\mathbb{A}_2\}]\ )$

$\mathit{prune}(P)$ ≡ best (= minimal cost + interestingly ordered) plans in set P

$l \circledast r \equiv \{l \Join^{\mathrm{nl}} r,\ r \Join^{\mathrm{nl}} l,\quad l \Join^{\mathrm{mj}} r,\ r \Join^{\mathrm{mj}} l,\quad l \Join^{\mathrm{hj}} r,\ r \Join^{\mathrm{hj}} l\}$

Join Plan Generation (Notes)

- **Access plan choices** (*access*(·)):
  - Consider sequential/index scans if $\mathbb{A}$ is a base table, otherwise simply consume $\mathbb{A}$'s rows.

- **Join plan choices** ($\_ \circledast \_$):
  - 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 $\{\mathbb{A}_1, \dots, \mathbb{A}_n\}$, memorize in *opt*[·]
    1. … its overall best plan and
    2. … its best plan satisfying each **interesting order**.

(Bushy) Join Plan Generation: Pseudo Code

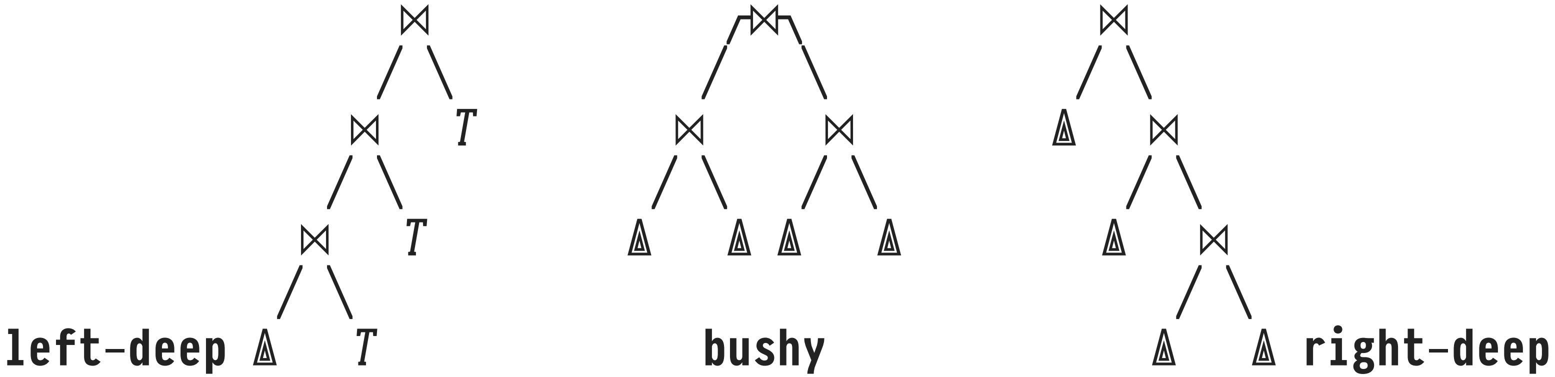
```
JoinPlan({$\mathbb{A}_1,\dots,\mathbb{A}_n$}):
  foreach $p \in \{\mathbb{A}_1,\dots,\mathbb{A}_n\}$                    } 1-input plans
    $opt[\{p\}] \leftarrow prune(access(p))$;

  for $k$ in $2,\dots,n$                                  } $k$-input plans
    foreach $S \subseteq \{\mathbb{A}_1,\dots,\mathbb{A}_n\}$ with $|S| = k$ } enumerate subsets
      $opt[S] \leftarrow \emptyset$;
      foreach $T \subset S$ with $T \neq \emptyset$
        $opt[S] \leftarrow opt[S] \cup \{ opt[T] \ulcorner\bowtie^{a}\urcorner opt[S \setminus T] \}$;
      $opt[S] \leftarrow prune(opt[S])$;

return $opt[\{\mathbb{A}_1,\dots,\mathbb{A}_n\}]$;
```

- *access*(·), *prune*(·) defined as above, $\ulcorner\bowtie^{a}\urcorner$ 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 ($\_\ \theta\ \_ = \mathtt{true}$).

- 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

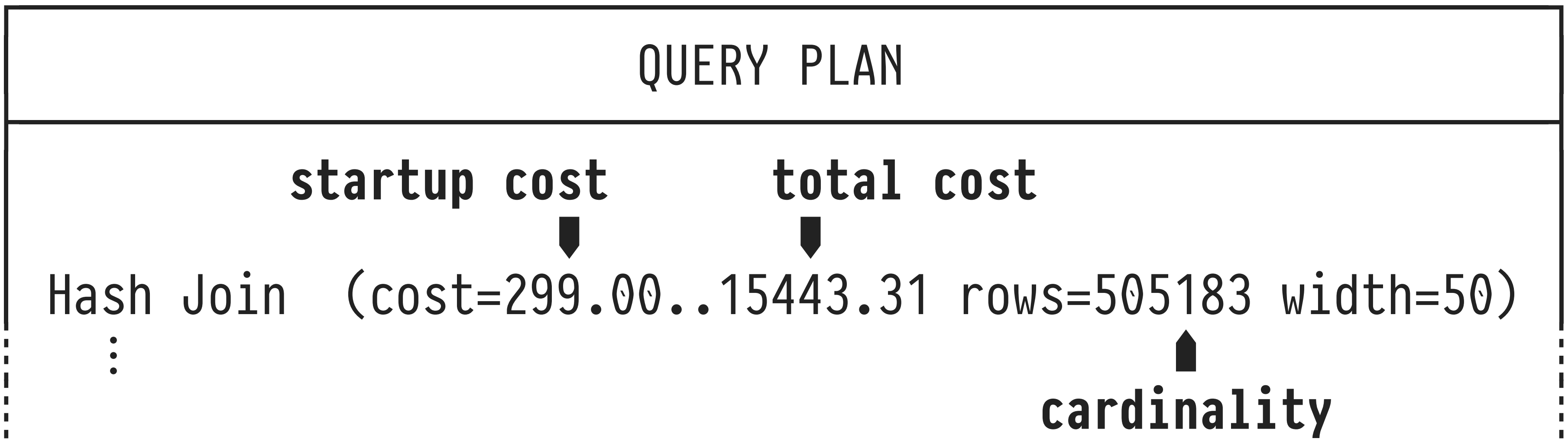
The query optimizer explores the vast plan search space to find the **optimal (“best”, “cheapest”) plan.**

- 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=c_1..c_2← …).
- 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

Hash Join  (cost=299.00..15443.31 rows=505183 width=50)
  ⋮
```

- **run cost** $\overset{\text{def}}{=}$ **total cost** − **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 (≡ ∞).

Cost Model Configuration

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

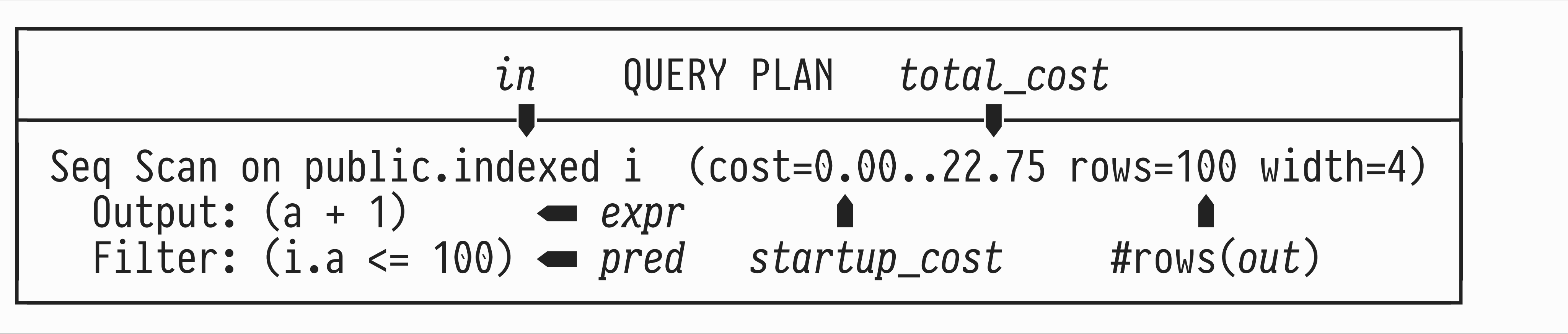
- Parameters are configurable:
  - Seek cost, thus random_page_cost » seq_page_cost. But…
  - … if DB fits in RAM, random_page_cost = seq_page_cost may be more appropriate.

Cost of Seq Scan ❶

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*?



```
                        QUERY PLAN
Seq Scan on public.indexed i  (cost=0.00..22.75 rows=100 width=4)
  Output: (a + 1)
  Filter: (i.a <= 100)
```

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…
#rows(*in*)	# rows (cardinality) of table *in*	pg_class.reltuples
#pages(*in*)	# pages in heap file of *in*	pg_class.relpages
sel(*pred*)	selectivity of filter *pred*[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*) ∊ {0,…,1} with sel(*pred*) = 0 ≡ no row satisfies filter *pred*.

Cost of Seq Scan ❸

$$\textbf{startup\_cost} \stackrel{\text{def}}{=} \overset{\text{typically} = 0}{\text{startup\_cost}(\textit{pred}) + \text{startup\_cost}(\textit{expr})}$$

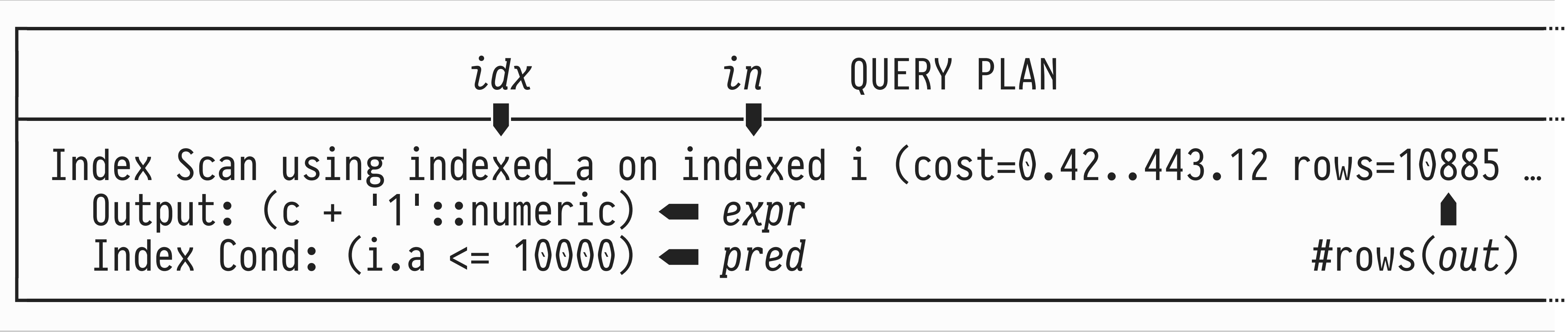
$$\begin{aligned}\text{cpu\_run\_cost} \stackrel{\text{def}}{=}\ & \text{\#rows}(\textit{in}) \times (\overbrace{\text{cpu\_tuple\_cost}}^{\text{decode heap row}} + \overbrace{\text{run\_cost}(\textit{pred})}^{\text{evaluate filter}}) \\ & + \underbrace{\text{\#rows}(\textit{in}) \times \text{sel}(\textit{pred})}_{= \text{\#rows}(\textit{out})} \times \underbrace{\text{run\_cost}(\textit{expr})}_{\text{evaluate SELECT clause}}\end{aligned}$$

$$\text{disk\_run\_cost} \stackrel{\text{def}}{=} \underbrace{\text{\#pages}(\textit{in}) \times \text{seq\_page\_cost}}_{\text{sequentially read entire input heap file}}$$

$$\textbf{total\_cost} \stackrel{\text{def}}{=} \text{startup\_cost} + \underbrace{\text{cpu\_run\_cost} + \text{disk\_run\_cost}}_{= \textbf{run\_cost}}$$

Cost of Index Scan ➊

Modeling the cost for an Index Scan has to reflect that *two* data structures (heap file & B⁺Tree) are involved:



```
                          QUERY PLAN
Index Scan using indexed_a on indexed i (cost=0.42..443.12 rows=10885 …
  Output: (c + '1'::numeric)
  Index Cond: (i.a <= 10000)
```

The model separately accounts for

1. the B⁺Tree descent (startup of the Index Scan),
2. the index leaf level scan, and
3. heap file access (clustered *vs.* non-clustered).

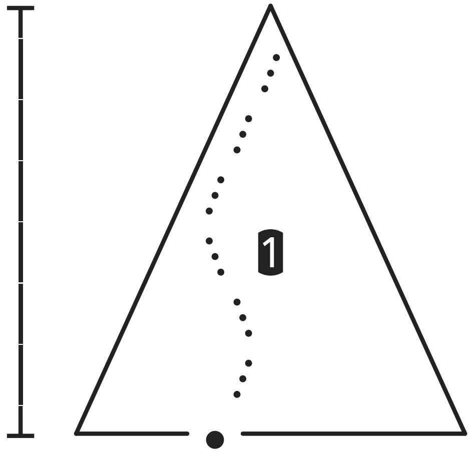
Cost of Index Scan ❷

Cost model parameters:

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

- corr(*idx*) ∊ {-1.0,…,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 ❸ (B⁺Tree Descent)



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

⇛ # of key comparisons during B⁺Tree descent ❶:

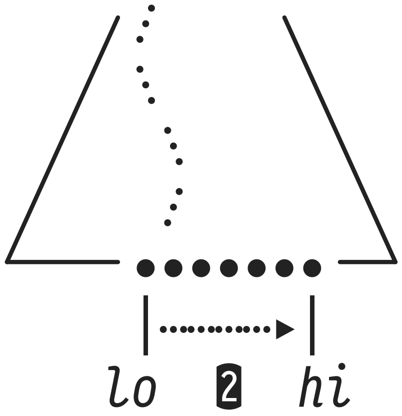
$$\underbrace{\lceil \log_2(2 \times o)}_{} \times h\rceil = \lceil \log_2(\#\mathrm{rows}(\mathit{idx}))\rceil$$

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

$$\textbf{startup\_cost} \overset{\text{def}}{=}$$
$$\quad \mathrm{startup\_cost}(\mathit{pred}) + \mathrm{startup\_cost}(\mathit{expr})$$
$$+ (\underbrace{\lceil \log_2(\#\mathrm{rows}(\mathit{idx}))\rceil}_{\text{B}^+\text{Tree descent}} + \underbrace{(h + 1)}_{\vdots\ +\ \bullet} \times \underbrace{50}_{\text{index node processing}}) \times \mathrm{cpu\_operator\_cost}$$

Cost of Index Scan ❹ (Leaf Level Scan)

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

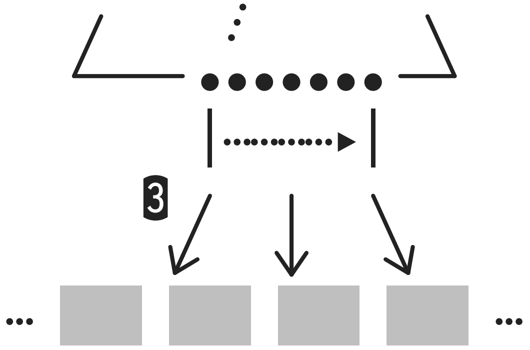


$$\text{index\_cpu\_cost} \overset{\text{def}}{=} \overbrace{\text{sel}(\textit{pred}) \times \#\text{rows}(\textit{idx})}^{\text{\# rows in scanned range } |{\cdots}\blacktriangleright|} \times (\underbrace{\text{cpu\_index\_tuple\_cost}}_{\text{decode index leaf entry}} + \underbrace{\text{run\_cost}(\textit{pred})}_{\text{evaluate } \leqslant \textit{ hi}})$$

$$\text{index\_IO\_cost} \overset{\text{def}}{=} \overbrace{\lceil \text{sel}(\textit{pred}) \times \#\text{pages}(\textit{idx}) \rceil}^{\text{\# of pages } \bullet \text{ in scanned range}} \times \overbrace{\text{random\_page\_cost}}^{\text{B}^{+}\text{Tree leaves not clustered}}$$

Cost of Index Scan ❺ (Heap File Access)

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

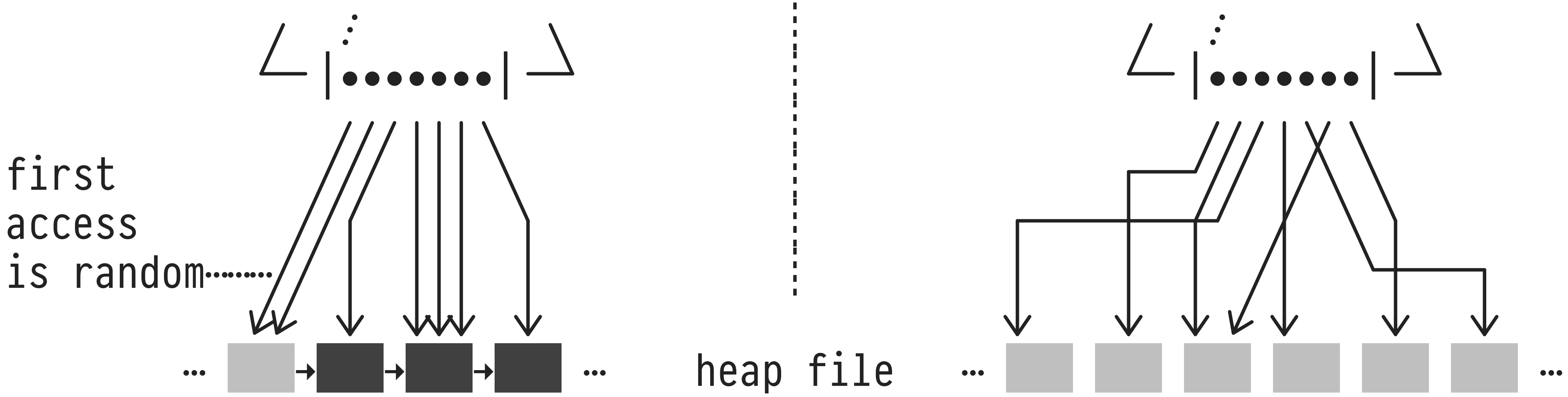


$$\text{heap\_cpu\_cost} \overset{\text{def}}{=} \overbrace{\text{sel}(\textit{pred}) \times \#\text{rows}(\textit{in})}^{\#\text{ rows accessed in heap file}} \times (\underbrace{\text{cpu\_tuple\_cost}}_{\text{decode row on } \blacksquare} + \underbrace{\text{run\_cost}(\textit{expr})}_{\text{evaluate Output}})$$

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

$$\begin{aligned}\text{heap\_IO\_cost} \overset{\text{def}}{=}\ & \text{unclustered\_IO\_cost} \\ & + \underset{\uparrow\ \approx\ \text{clustering factor} \in \{0,\dots,1\}}{\text{corr}(\textit{idx})^2} \times (\text{clustered\_IO\_cost} - \text{unclustered\_IO\_cost})\end{aligned}$$

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



clustered_IO_cost ≝ 1 × random_page_cost (▒)
+ (sel(*pred*) × #pages(*in*) − 1) × seq_page_cost (■)

unclustered_IO_cost ≝ #pages(*in*) × random_page_cost (▒)

total_cost ≝ **startup_cost** + index_cpu_cost + index_IO_cost
+ heap_cpu_cost + heap_IO_cost