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 the **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 ($\approx SF \times 1.5{\times}10^6$ rows)

l_orderkey	l_linenumber	l_partkey	l_quantity	l_extendedprice	…
o					

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

c_custkey	c_name	c_acctbal	c_nationkey	…
c			*n*	

customer ($\approx SF \times 150000$ 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** $(l \bowtie o) \bowtie c$.
- Commutativity and associativity of $\bowtie$ enable the RDBMS to **reorder** the joins—based on *estimated evaluation costs*.
  - … unless we insist on the syntactic order.

Demonstrate the impact of query optimization (force join reordering off) and show a (simple) example of unnesting in the FROM clause.

```
-- ❶ Check input tables
--   (this relies on a small TPC-H instance with SF = 0.01, see Week14/live/TPC-H)

  \d lineitem
```

Table "public.lineitem"

Column	Type	Collation	Nullable	Default
l_orderkey	integer		not null	
l_partkey	integer		not null	
l_suppkey	integer		not null	
l_linenumber	integer		not null	
l_quantity	numeric(15,2)		not null	
l_extendedprice	numeric(15,2)		not null	
l_discount	numeric(15,2)		not null	
l_tax	numeric(15,2)		not null	
l_returnflag	character(1)		not null	
l_linestatus	character(1)		not null	
l_shipdate	date		not null	
l_commitdate	date		not null	
l_receiptdate	date		not null	
l_shipinstruct	character(25)		not null	
l_shipmode	character(10)		not null	
l_comment	character varying(44)		not null	

```
Indexes:
    "lineitem_pkey" PRIMARY KEY, btree (l_orderkey, l_linenumber)
Foreign-key constraints:
    "lineitem_l_orderkey_fkey" FOREIGN KEY (l_orderkey) REFERENCES orders(o_orderkey)
    "lineitem_l_partkey_fkey" FOREIGN KEY (l_partkey, l_suppkey) REFERENCES partsupp(ps_partkey, ps_suppkey)

  \d orders
```

Table "public.orders"

Column	Type	Collation	Nullable	Default
o_orderkey	integer		not null	
o_custkey	integer		not null	
o_orderstatus	character(1)		not null	
o_totalprice	numeric(15,2)		not null	
o_orderdate	date		not null	
o_orderpriority	character(15)		not null	
o_clerk	character(15)		not null	
o_shippriority	integer		not null	

```
 o_comment        | character varying(79) |            | not null |
Indexes:
    "orders_pkey" PRIMARY KEY, btree (o_orderkey)
Referenced by:
    TABLE "lineitem" CONSTRAINT "lineitem_l_orderkey_fkey" FOREIGN KEY (l_orderkey) REFERENCES orders(o_orderkey)

  \d customer
                         Table "public.customer"
    Column     |          Type          | Collation | Nullable | Default
---------------+------------------------+-----------+----------+---------
 c_custkey     | integer                |           | not null |
 c_name        | character varying(25)  |           | not null |
 c_address     | character varying(40)  |           | not null |
 c_nationkey   | integer                |           | not null |
 c_phone       | character(15)          |           | not null |
 c_acctbal     | numeric(15,2)          |           | not null |
 c_mktsegment  | character(10)          |           | not null |
 c_comment     | character varying(117) |           | not null |
Indexes:
    "customer_pkey" PRIMARY KEY, btree (c_custkey)
Foreign-key constraints:
    "customer_c_nationkey_fkey" FOREIGN KEY (c_nationkey) REFERENCES nation(n_nationkey)


-- ❷ Evaluate Q₁₄, force join reordering/unnesting OFF

  set join_collapse_limit = 1;
  set from_collapse_limit = 1;

  -- (a) Explicit join order via JOIN … ON (join_collapse_limit)
  EXPLAIN (ANALYZE)
    SELECT l.l_partkey, l.l_quantity, l.l_extendedprice
    FROM   lineitem AS l JOIN orders AS o
             ON (l.l_orderkey = o.o_orderkey)
           JOIN customer AS c
             ON (o.o_custkey = c.c_custkey)
    WHERE  c.c_name = 'Customer#000000001';
                              QUERY PLAN (c ⋈ (l ⋈ o))
---------------------------------------------------------------------------------------------
 Nested Loop  (cost=598.50..3894.86 rows=40 width=17) (actual time=23.063..65.694 rows=35 loops=1)
   Join Filter: (o.o_custkey = c.c_custkey)
   Rows Removed by Join Filter: 60140
   ->  Seq Scan on customer c  (cost=0.00..54.75 rows=1 width=4) (actual time=0.022..0.468 rows=1 loops=1)
```

```
        Filter: ((c_name)::text = 'Customer#000000001'::text)
        Rows Removed by Filter: 1499
  ->  Hash Join  (cost=598.50..3087.92 rows=60175 width=21) (actual time=13.785..56.956 rows=60175 loops=1)
        Hash Cond: (l.l_orderkey = o.o_orderkey)
        ->  Seq Scan on lineitem l  (cost=0.00..1729.75 rows=60175 width=21) (actual time=0.011..10.405 rows=60175 loops=1)
        ->  Hash  (cost=411.00..411.00 rows=15000 width=8) (actual time=13.746..13.746 rows=15000 loops=1)
              Buckets: 16384  Batches: 1  Memory Usage: 714kB
              ->  Seq Scan on orders o  (cost=0.00..411.00 rows=15000 width=8) (actual time=0.013..6.647 rows=15000 loops=1)
Planning time: 1.115 ms
Execution time: 65.792 ms ⬅ slow
```

```
-- (b) Prescribed join order via subquery nesting in the FROM clause (from_collapse_limit)
EXPLAIN (ANALYZE)
  SELECT lo.l_partkey, lo.l_quantity, lo.l_extendedprice
  FROM   (SELECT l.l_partkey, l.l_quantity, l.l_extendedprice, o.o_custkey
          FROM   lineitem AS l, orders AS o
          WHERE  l.l_orderkey = o.o_orderkey) AS lo,
         customer AS c
  WHERE  c.c_name = 'Customer#000000001'
    AND  lo.o_custkey = c.c_custkey;
```

```
                                  QUERY PLAN (c ⋈ (l ⋈ o))
Nested Loop  (cost=598.50..3894.86 rows=40 width=17) (actual time=18.125..54.639 rows=35 loops=1)
  Join Filter: (o.o_custkey = c.c_custkey)
  Rows Removed by Join Filter: 60140
  ->  Seq Scan on customer c  (cost=0.00..54.75 rows=1 width=4) (actual time=0.021..0.319 rows=1 loops=1)
        Filter: ((c_name)::text = 'Customer#000000001'::text)
        Rows Removed by Filter: 1499
  ->  Hash Join  (cost=598.50..3087.92 rows=60175 width=21) (actual time=10.879..48.201 rows=60175 loops=1)
        Hash Cond: (l.l_orderkey = o.o_orderkey)
        ->  Seq Scan on lineitem l  (cost=0.00..1729.75 rows=60175 width=21) (actual time=0.011..8.804 rows=60175 loops=1)
        ->  Hash  (cost=411.00..411.00 rows=15000 width=8) (actual time=10.842..10.842 rows=15000 loops=1)
              Buckets: 16384  Batches: 1  Memory Usage: 714kB
              ->  Seq Scan on orders o  (cost=0.00..411.00 rows=15000 width=8) (actual time=0.011..5.293 rows=15000 loops=1)
Planning time: 0.926 ms
Execution time: 54.733 ms ⬅ slow
```

```
-- Plan shape and cardinalities:
--
--            | 35
--            ⋈
--       1 /    \ 60175
--        σ      ⋈
--         |   /   \
--    1500 c   l   o 15000
--           60175


-- ❷ Re-evaluate Q₁₄, with join reordering/unnesting enabled

  reset join_collapse_limit;
  reset from_collapse_limit;


  EXPLAIN (ANALYZE)
    SELECT l.l_partkey, l.l_quantity, l.l_extendedprice
    FROM   lineitem AS l JOIN orders AS o
             ON (l.l_orderkey = o.o_orderkey)
           JOIN customer AS c
             ON (o.o_custkey = c.c_custkey)
    WHERE  c.c_name = 'Customer#000000001';
```

```
                                   QUERY PLAN ((c ⋈ o) ⋈ⁱᵈˣ l)
Nested Loop  (cost=0.29..660.70 rows=40 width=17) (actual time=1.660..7.058 rows=35 loops=1)
  ->  Nested Loop  (cost=0.00..653.25 rows=10 width=4) (actual time=1.633..6.905 rows=9 loops=1)
        Join Filter: (o.o_custkey = c.c_custkey)
        Rows Removed by Join Filter: 14991
        ->  Seq Scan on customer c  (cost=0.00..54.75 rows=1 width=4) (actual time=0.023..0.367 rows=1 loops=1)
              Filter: ((c_name)::text = 'Customer#000000001'::text)
              Rows Removed by Filter: 1499
        ->  Seq Scan on orders o  (cost=0.00..411.00 rows=15000 width=8) (actual time=0.011..3.429 rows=15000 loops=1)
  ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.29..0.71 rows=4 width=21) (actual time=0.009..0.012 rows=4 loops=9)
        Index Cond: (l_orderkey = o.o_orderkey)
Planning time: 1.185 ms
Execution time: 7.145 ms ⬅ fast
```

```
-- Plan shape and cardinalities:
--               | 35
--               ⋈
--           9 /   \
--            ⋈     |
--        1 /   \   |
--         σ     |  |
--         |     |  |
--    1500 c     o  l 60175
--           15000
```

```
EXPLAIN (ANALYZE)
  SELECT lo.l_partkey, lo.l_quantity, lo.l_extendedprice
  FROM   (SELECT l.l_partkey, l.l_quantity, l.l_extendedprice, o.o_custkey
          FROM   lineitem AS l, orders AS o
          WHERE  l.l_orderkey = o.o_orderkey) AS lo,
         customer AS c
  WHERE  c.c_name = 'Customer#000000001'
    AND  lo.o_custkey = c.c_custkey;
```

```
                              QUERY PLAN ((c ⋈ o) ⋈ᶦᵈˣ l)

Nested Loop  (cost=0.29..660.70 rows=40 width=17) (actual time=1.616..6.295 rows=35 loops=1)
  ->  Nested Loop  (cost=0.00..653.25 rows=10 width=4) (actual time=1.582..6.177 rows=9 loops=1)
        Join Filter: (o.o_custkey = c.c_custkey)
        Rows Removed by Join Filter: 14991
        ->  Seq Scan on customer c  (cost=0.00..54.75 rows=1 width=4) (actual time=0.023..0.324 rows=1 loops=1)
              Filter: ((c_name)::text = 'Customer#000000001'::text)
              Rows Removed by Filter: 1499
        ->  Seq Scan on orders o  (cost=0.00..411.00 rows=15000 width=8) (actual time=0.011..2.891 rows=15000 loops=1)
  ->  Index Scan using lineitem_pkey on lineitem l  (cost=0.29..0.71 rows=4 width=21) (actual time=0.009..0.010 rows=4 loops=9)
        Index Cond: (l_orderkey = o.o_orderkey)
Planning time: 1.214 ms
Execution time: 6.389 ms ⬅ fast
```

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     ⟁, …, ⟁                 -- ⟁ ≡ 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.

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  $p_1$                                  $\overset{*}{\equiv}$   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  $p_3$)
```

* Precondition: e_1 is key in the left-hand side query

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

Demonstrate subquery unnesting for nesting in the ➊ FROM clause and nesting in the ➋ WHERE clause.

- ➊ Unnesting
  - N.B.: Rename t → n as required:

```
SELECT c.c_name
FROM   customer AS c, nation AS n
WHERE  c.c_nationkey = n.n_nationkey
AND    strpos(c.c_address, n.n_name) > 0;
```

- ➋ Unnesting (IN → join + DISTINCT)
  - ⚠️ DISTINCT actually *not* needed in this special case since for each o there will be at most one join partner in customer (c_custkey is key).

```
SELECT DISTINCT o.o_orderkey
FROM   orders AS o, customer AS c
WHERE  c.c_name = 'Customer#000000001'
AND    o.o_custkey = c.c_custkey;
```

```
-- ➊ Nesting in the FROM clause: inlining

  EXPLAIN (COSTS false)
    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;
```

```
                           QUERY PLAN
─────────────────────────────────────────────────────────────────────
 Hash Join
   Hash Cond: (c.c_nationkey = n.n_nationkey)
   Join Filter: (strpos((c.c_address)::text, (n.n_name)::text) > 0)
   ->  Seq Scan on customer c
   ->  Hash
         ->  Seq Scan on nation n
```

```
  -- Manual unnesting: identical plan

  EXPLAIN (COSTS false)
```

```
    SELECT c.c_name
    FROM   customer AS c, nation AS n
    WHERE  c.c_nationkey = n.n_nationkey
    AND    strpos(c.c_address, n.n_name) > 0;
```

```
                        QUERY PLAN
──────────────────────────────────────────────────────────────────────
 Hash Join
   Hash Cond: (c.c_nationkey = n.n_nationkey)
   Join Filter: (strpos((c.c_address)::text, (n.n_name)::text) > 0)
   ->  Seq Scan on customer c
   ->  Hash
         ->  Seq Scan on nation n
```

```
-- ❷ Nesting in the WHERE clause: IN ⇒ semijoin

  EXPLAIN (COSTS false)
    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 = 'Customer#000000001');
```

```
                        QUERY PLAN
─────────────────────────────────────────────────────────────────
 Nested Loop
   Join Filter: (o.o_custkey = c.c_custkey) ◀ will find at most one match for o ⇒ no Unique required
   ->  Seq Scan on customer c
         Filter: ((c_name)::text = 'Customer#000000001'::text)
   ->  Seq Scan on orders o
```

```
  -- Variant of ❷: do not compare with key `c_custkey` ⇒ optimizer uses hash-based duplicate elimination
     to ensure that each o will find at most one join partner

  EXPLAIN (COSTS false)
    SELECT o.o_orderkey
    FROM   orders AS o
    WHERE  o.o_clerk IN
      (SELECT c.c_name
       FROM   customer AS c);
```

```
                    QUERY PLAN
 ────────────────────────────────────────────────
 Hash Join
   Hash Cond: (o.o_clerk = (c.c_name)::bpchar)  ◀ will find at most one match for o ⇒ no Unique required
   ->  Seq Scan on orders o
   ->  Hash
         ->  HashAggregate                        ⎫ build table of
               Group Key: (c.c_name)::bpchar      ⎬ unique customer
               ->  Seq Scan on customer c         ⎭ names

-- Variant of ❷: switch off hash join ⇒ optimizer uses (Merge) Semi Join to ensure
   that each o is matched with at most one join partner

set enable_hashjoin = off;

EXPLAIN (COSTS false)
  SELECT o.o_orderkey
  FROM   orders AS o
  WHERE  o.o_clerk IN
    (SELECT c.c_name
     FROM   customer AS c);

                    QUERY PLAN
 ────────────────────────────────────────────────
 Merge Semi Join
   Merge Cond: (o.o_clerk = (c.c_name)::bpchar)
   ->  Sort
         Sort Key: o.o_clerk
         ->  Seq Scan on orders o
   ->  Sort
         Sort Key: c.c_name USING <
         ->  Seq Scan on customer c

set enable_hashjoin = on;
```

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

SQL

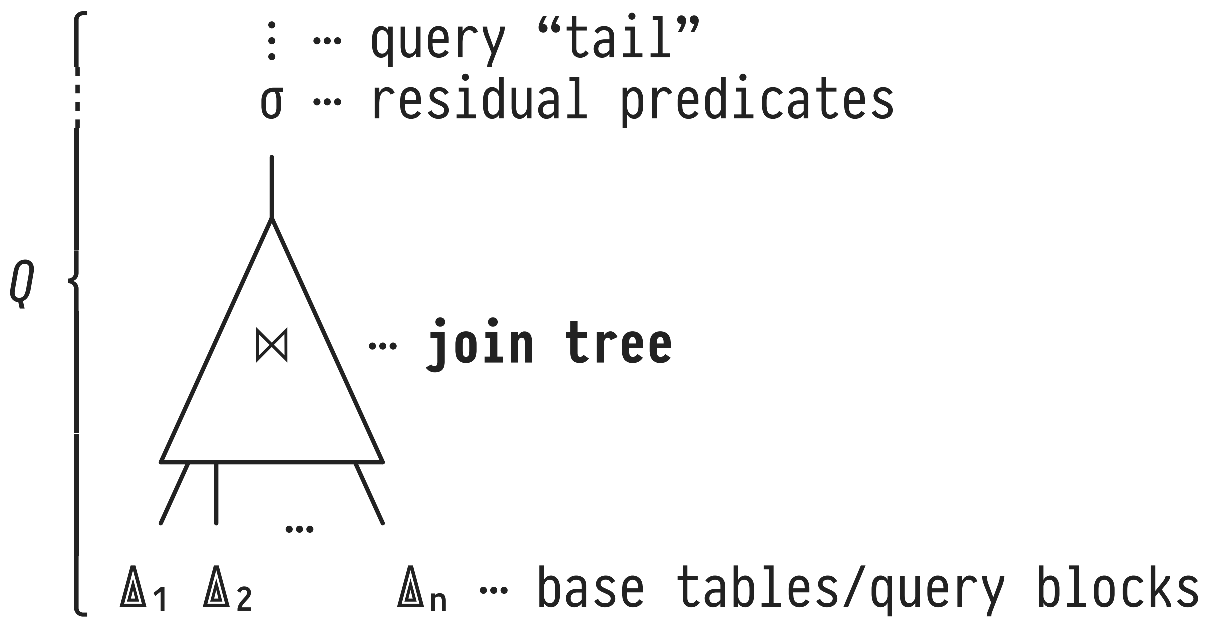
Q:

⋮

FROM $\triangle_1, \triangle_2, \ldots, \triangle_n$
WHERE p

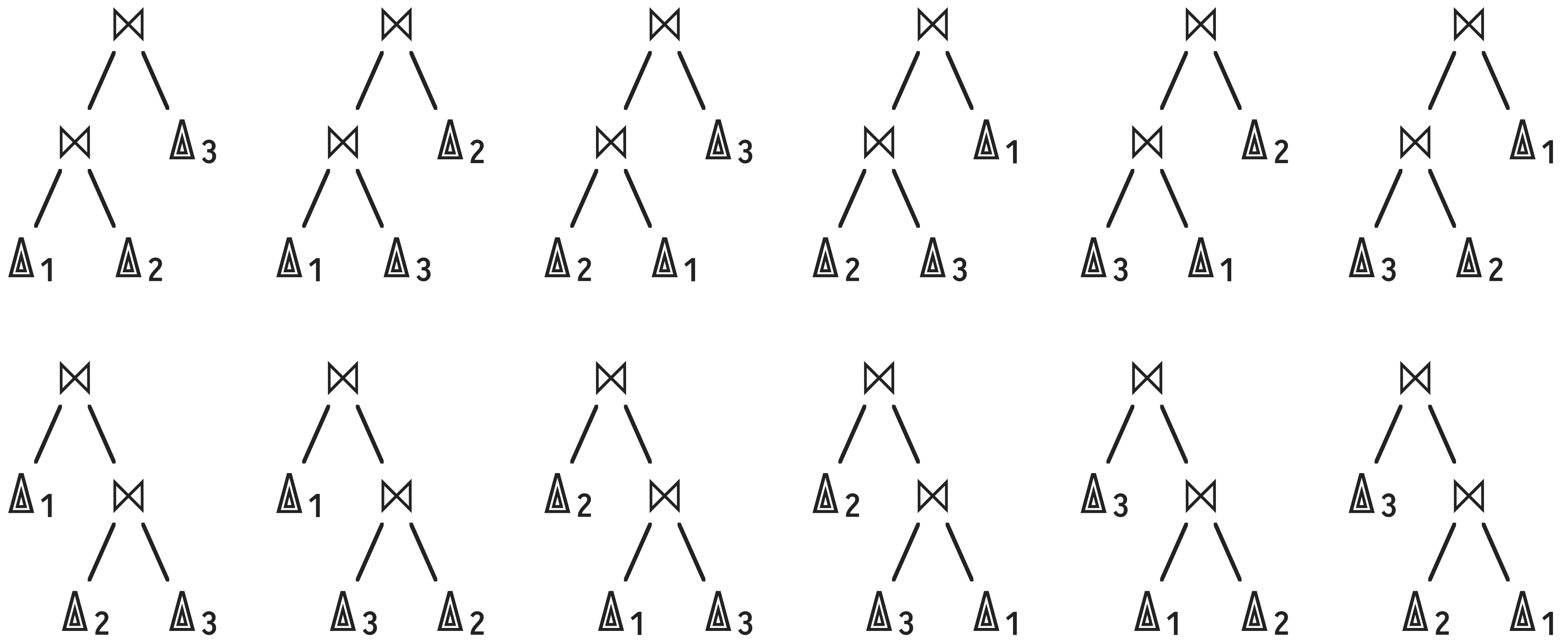
⋮

Canonical Plan


⋮ ⋯ query “tail”
σ ⋯ residual predicates
Q
⋈
⋯ join tree
⋯
$\triangle_1$ $\triangle_2$ $\triangle_n$ ⋯ base tables/query blocks

Join Tree Optimization

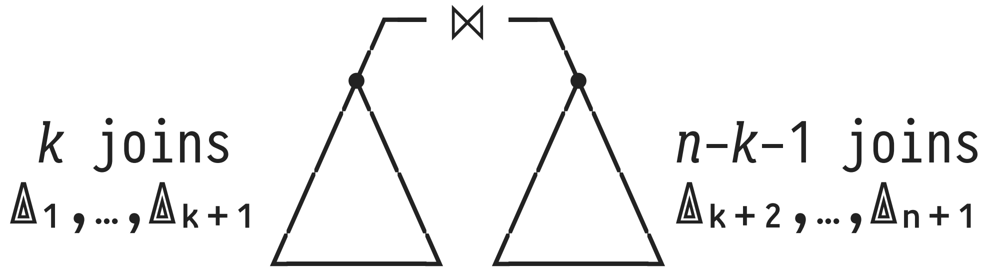
Given n join inputs, the number of possible **join tree shapes** is *huge*. Consider $n = 3$:



- 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. $C_0 = 1$.

Catalan numbers ≡ possible tree shapes:

- C_0 = 1 (single leaf node, no inner nodes)
- C_1 = 1 (single inner node)
- $C_2 = C_0 \times C_1 + C_1 \times C_0 = 1 + 1 = 2$

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.

- 347271609484800 ≡ 347 trillion 271 billion 609 million 484 thousand 800

Join Plan Generation Through Dynamic Programming

- **Problem:** Find optimal query plan $\mathit{opt}[\{\mathbb{A}_1,\ldots,\mathbb{A}_n\}]$ that joins n inputs $\mathbb{A}_1,\ldots,\mathbb{A}_n$.

  1. **Iteration 1:** For each $\mathbb{A}_j$, find and memorize **best 1-input plan** $\mathit{opt}[\{\mathbb{A}_j\}]$ that accesses $\mathbb{A}_j$ 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. ⎭ $\mathit{opt}[\cdot]$ memo 👍

Bottom-Up Dynamic Programming (n = 3)

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

$prune(P) \equiv$ best (= minimal cost + interestingly ordered) plans in set P

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

Join Plan Generation (Notes)

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

- **Join plan choices** (_ ⊛ _):
  - 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 $\{𝔸_1, …, 𝔸_n\}$, memorize in *opt*[·]
    1. … its overall best plan and
    2. … its best plan satisfying each **interesting order**.

(Bushy) Join Plan Generation: Pseudo Code

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

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

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

- *access*(·), *prune*(·) defined as above,
  $\ulcorner\!\bowtie^a\!\urcorner$ builds all join algorithm choices ($a \in$ {nl,mj,hj}).

- Enumerate all non-empty true subsets of S (bit set representation in C):

```
T = S & -S;
do {
  /* perform operation on T */
  T = S & (T - S);
} while (T != S);
```

Building the *opt*[·] and pruning plans early leads to a compact representation of the explored search space. Example for $n = 4$, all $\mathbb{A}_i$ base tables (i.e., access($\mathbb{A}_i$) considers Seq Scan, Index Scan, Bitmap Scan):

$$\left.\begin{aligned}
&\mathrm{opt}[\{\mathbb{A}_1\}] = \mathrm{prune}(\mathrm{access}(\mathbb{A}_1)) \leftarrow 3 \text{ plans considered}\\
&\mathrm{opt}[\{\mathbb{A}_2\}] = \dots\\
&\mathrm{opt}[\{\mathbb{A}_3\}] = \dots\\
&\mathrm{opt}[\{\mathbb{A}_4\}] = \dots
\end{aligned}\right\} \begin{array}{l} 4 \times 3 = 12 \text{ plans considered,}\\ \text{only 4 plans memorized} \end{array}$$

$$\left.\begin{aligned}
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_2\}] = \mathrm{prune}(\mathrm{opt}[\{\mathbb{A}_1\}] \circledast \mathrm{opt}[\{\mathbb{A}_2\}]) \leftarrow 6 \text{ plans considered}\\
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_3\}] = \dots\\
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_4\}] = \dots\\
&\mathrm{opt}[\{\mathbb{A}_2,\mathbb{A}_3\}] = \dots\\
&\mathrm{opt}[\{\mathbb{A}_2,\mathbb{A}_4\}] = \dots\\
&\mathrm{opt}[\{\mathbb{A}_3,\mathbb{A}_4\}] = \dots
\end{aligned}\right\} \begin{array}{l} 6 \times 6 = 36 \text{ plans considered,}\\ \text{only 6 plans memorized} \end{array}$$

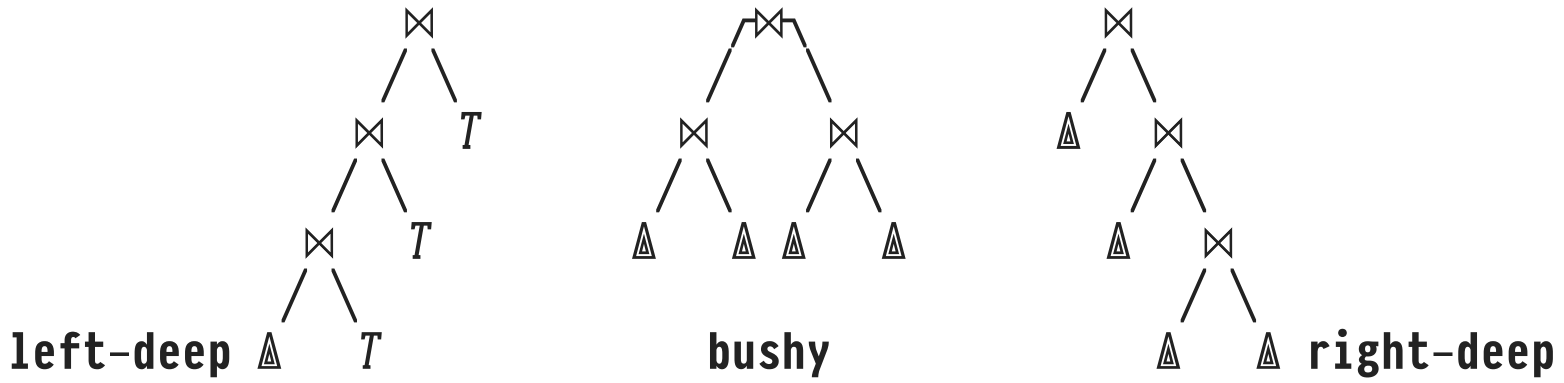
$$\left.\begin{aligned}
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_2,\mathbb{A}_3\}] = \left.\begin{aligned}&\mathrm{opt}[\{\mathbb{A}_1\}] \circledast \mathrm{opt}[\{\mathbb{A}_2,\mathbb{A}_3\}] \cup\\ &\mathrm{opt}[\{\mathbb{A}_2\}] \circledast \mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_3\}] \cup\\ &\mathrm{opt}[\{\mathbb{A}_3\}] \circledast \mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_2\}]\end{aligned}\right\} 18 \text{ plans considered}\\
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_3,\mathbb{A}_4\}] = \dots\\
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_2,\mathbb{A}_4\}] = \dots\\
&\mathrm{opt}[\{\mathbb{A}_2,\mathbb{A}_3,\mathbb{A}_4\}] = \dots
\end{aligned}\right\} \begin{array}{l} 4 \times 18 = 72 \text{ plans considered,}\\ \text{only 4 plans memorized} \end{array}$$

$$\left.\begin{aligned}
\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_2,\mathbb{A}_3,\mathbb{A}_4\}] = {}&\mathrm{opt}[\{\mathbb{A}_1\}] \circledast \mathrm{opt}[\{\mathbb{A}_2,\mathbb{A}_3,\mathbb{A}_4\}] \cup\\
&\mathrm{opt}[\{\mathbb{A}_2\}] \circledast \mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_3,\mathbb{A}_4\}] \cup\\
&\mathrm{opt}[\{\mathbb{A}_3\}] \circledast \mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_2,\mathbb{A}_4\}] \cup\\
&\mathrm{opt}[\{\mathbb{A}_4\}] \circledast \mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_2,\mathbb{A}_3\}] \cup\\
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_2\}] \circledast \mathrm{opt}[\{\mathbb{A}_3,\mathbb{A}_4\}] \cup\\
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_3\}] \circledast \mathrm{opt}[\{\mathbb{A}_2,\mathbb{A}_4\}] \cup\\
&\mathrm{opt}[\{\mathbb{A}_1,\mathbb{A}_4\}] \circledast \mathrm{opt}[\{\mathbb{A}_2,\mathbb{A}_3\}]
\end{aligned}\right\} \begin{array}{l} 42 \text{ plans considered,}\\ \text{only 1 plan memorized} \end{array}$$

Σ 162 plans considered
15 plans memorized

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).



Estimating Plan Cost

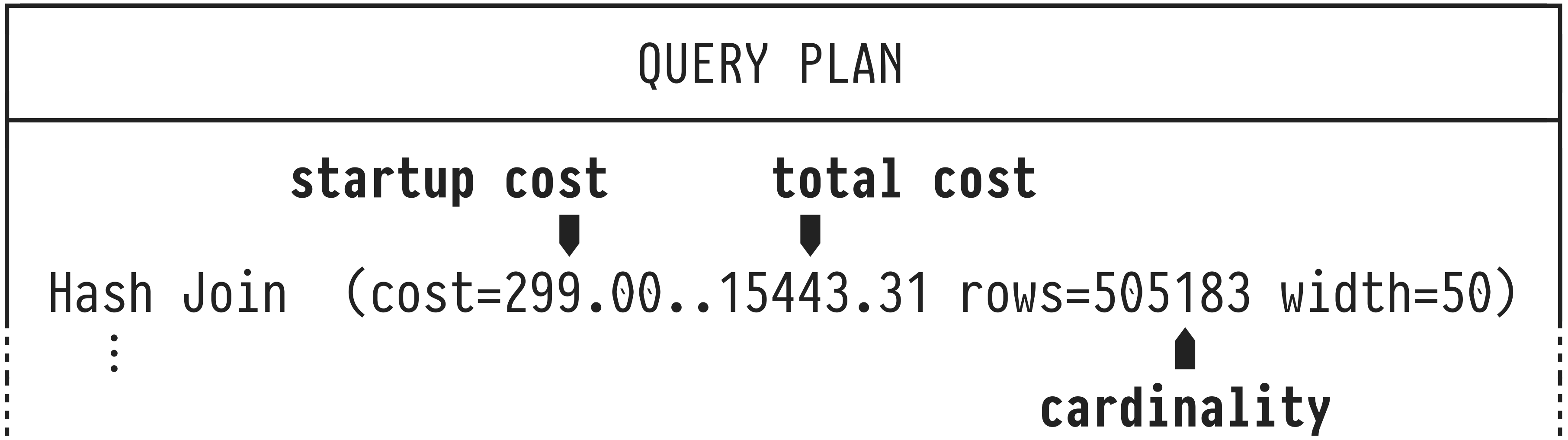
The query optimizer explores the vast plan search space to find the **optimal (“best”, “cheapest”) plan.**

- 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 ($\equiv \infty$).

Demonstrate that the PostgreSQL space$-based cost model does not try to estimate the true cost of plan evaluation:

```
-- ❶ Check input table

  \d ternary
                  Table "public.ternary"
```

Column	Type	Collation	Nullable	Default
a	integer		not null	
b	text		not null	
c	double precision			

```
-- ❷ Plan/evaluate two queries with same space$ cost but *wildly*
--   different true cost

   EXPLAIN (VERBOSE, ANALYZE)
     SELECT t.a::bigint + 1    -- same data type as used by _!
     FROM   ternary AS t;
```

```
                                   QUERY PLAN
Seq Scan on public.ternary t  (cost=0.00..25.00 rows=1000 width=8) (actual time=0.060..0.554 rows=1000 loops=1)
  Output: ((a)::bigint + 1)
Planning time: 0.063 ms
Execution time: 0.756 ms ⬅ fast
```

```
   EXPLAIN (VERBOSE, ANALYZE)
     SELECT t.a!
     FROM   ternary AS t;
```

```
                                   QUERY PLAN
Seq Scan on public.ternary t  (cost=0.00..25.00 rows=1000 width=32) (actual time=0.016..1535.795 rows=1000 loops=1)
  Output: ((a)::bigint !)
Planning time: 0.039 ms
Execution time: 1536.038 ms ⬅ slow
```

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

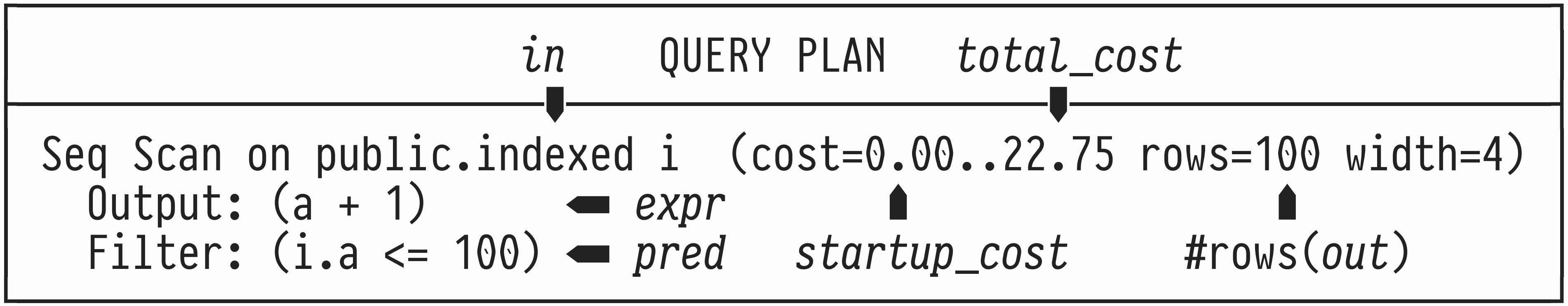
- Parameters are configurable:
  - Seek cost, thus random_page_cost » seq_page_cost. But…
  - … if DB fits in RAM, random_page_cost = seq_page_cost may be more appropriate.

Cost of Seq Scan 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*?



```
                              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** ③

$$\textbf{startup\_cost} \overset{\text{def}}{=} \overbrace{\text{startup\_cost}(\mathit{pred}) + \text{startup\_cost}(\mathit{expr})}^{\text{typically} = 0}$$

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

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

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

Demonstrate the cost derivation for Seq Scan on table indexed.

```
-- ❶ Set up input table

  DROP TABLE IF EXISTS indexed;
  CREATE TABLE indexed (a int PRIMARY KEY, b text, c numeric(3,2));

  INSERT INTO indexed(a,b,c)
    SELECT i, md5(i::text), sin(i)
    FROM   generate_series(1,1000000) AS i;

  ALTER INDEX indexed_pkey RENAME TO indexed_a;
  ANALYZE indexed;

  \d indexed
                Table "public.indexed"
```

Column	Type	Collation	Nullable	Default
a b c	integer text numeric(3,2)		not null	

```
Indexes:
    "indexed_a" PRIMARY KEY, btree (a)


-- ❷ Obtain meta data about table indexed

  SELECT reltuples AS "#rows(indexed)", relpages AS "#pages(indexed)"
  FROM   pg_class
  WHERE  relname = 'indexed';
```

#rows(indexed)	#pages(indexed)
1e+06	9346

```
-- ❸ Simple Seq Scan (no filter)

  EXPLAIN VERBOSE
    SELECT i.a
    FROM   indexed AS i;
```

QUERY PLAN

```
Seq Scan on public.indexed i  (cost=0.00..19346.00 rows=1000000 width=4)
  Output: a
```

```
- startup_cost  = startup_cost(pred) + startup_cost(expr) = 0 + 0
                = 0.00 ✓

- cpu_run_cost  =   #rows(indexed) × (cpu_tuple_cost + run_cost(pred))
                  + #rows(indexed) × sel(pred) × run_cost(expr)
                =   $10^6$ × (0.01 + 0)
                  + $10^6$ × 1.0 × 0
                = 10000.00

- disk_run_cost = #pages(indexed) × seq_page_cost
                = 9346 × 1.0
                = 9346.00

- total_cost    = startup_cost + cpu_run_cost + disk_run_cost
                = 0.00 + 10000.00 + 9346.00
                = 19346.00 ✓


-- ❹ Simple Seq Scan (no filter but "complex" SELECT clause expr)

  EXPLAIN VERBOSE
    SELECT i.a * 2 + 1
    FROM indexed AS i;
```

```
                               QUERY PLAN
Seq Scan on public.indexed i  (cost=0.00..24346.00 rows=1000000 width=4)
  Output: ((a * 2) + 1)
```

```
- startup_cost  = startup_cost(pred) + startup_cost(expr) = 0 + 0
                = 0.00 ✓

- cpu_run_cost  =   #rows(indexed) × (cpu_tuple_cost + run_cost(pred))
                  + #rows(indexed) × sel(pred) × run_cost(expr)
                =   $10^6$ × (0.01 + 0)
                  + $10^6$ × 1.0 × 2 × 0.0025
                = 15000.00        ▲
                                  run_cost(expr) = 2 × cpu_operator_cost: · * ·, · + ·

- disk_run_cost = #pages(indexed) × seq_page_cost
                = 9346 × 1.0
                = 9346.00

- total_cost    = startup_cost + cpu_run_cost + disk_run_cost
                = 0.00 + 15000.00 + 9346.00
                = 24346.00 ✓


-- ❺ Simple Seq Scan (with filter and SELECT clause expression)

  -- enforce Seq Scan
  set index_scan = off;
  set bitmap_scan = off;

  EXPLAIN VERBOSE
    SELECT i.a * 2 + 1
    FROM indexed AS i
    WHERE i.a <= 100000;
┌───────────────────────────────────────────────────────────────────────┐
│                              QUERY PLAN                               │
├───────────────────────────────────────────────────────────────────────┤
│ Seq Scan on public.indexed i  (cost=0.00..22342.17 rows=99235 width=4)│
│   Output: ((a * 2) + 1)                                               │
│   Filter: (i.a <= 100000)                                             │
└───────────────────────────────────────────────────────────────────────┘

- Estimated #rows(out) = 99325  ⇛  sel(i.a <= 100000) = 99325/10⁶ = 0.099325

- startup_cost  = startup_cost(pred) + startup_cost(expr) = 0 + 0
                = 0.00 ✓

- cpu_run_cost  =   #rows(indexed) × (cpu_tuple_cost + run_cost(pred))
                  + #rows(indexed) × sel(pred) × run_cost(expr)
                =   10⁶ × (0.01 + 0.0025)       ◀ run_cost(pred) = cpu_operator_cost: · <= ·
                  + 10⁶ × 0.099325 × 2 × 0.0025 ◀ run_cost(expr) = 2 × cpu_operator_cost: · * ·, · + ·
                = 12996.625

- disk_run_cost = #pages(indexed) × seq_page_cost
                = 9346 × 1.0
                = 9346.00

- total_cost    = startup_cost + cpu_run_cost + disk_run_cost
                = 0.00 + 12996.625 + 9346.00
                = 22342.625 ✓


-- ❺ Simple Seq Scan (with complex subquery filter)
```

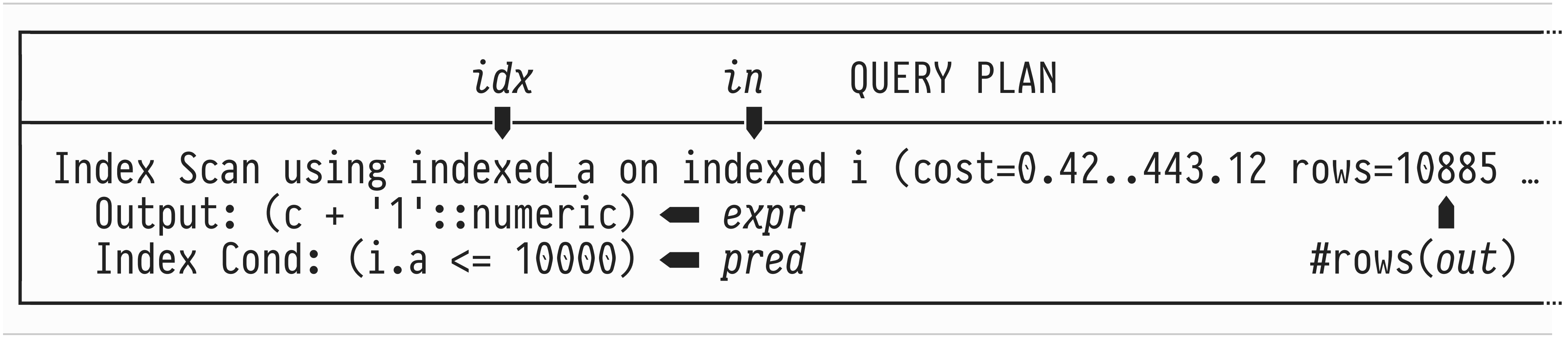
```
EXPLAIN VERBOSE
    SELECT i.a
    FROM indexed AS i
    WHERE i.a <= (SELECT AVG(i.a) FROM indexed AS i);
```

```
                                      QUERY PLAN
-----------------------------------------------------------------------------------------
 Seq Scan on public.indexed i  (cost=21846.01..46192.01 rows=333333 width=4)
   Output: i.a
   Filter: ((i.a)::numeric <= $0)
   InitPlan 1 (returns $0)
     ->  Aggregate  (cost=21846.00..21846.01 rows=1 width=32)
           Output: avg(i_1.a)
           ->  Seq Scan on public.indexed i_1  (cost=0.00..19346.00 rows=1000000 width=4)
                 Output: i_1.a, i_1.b, i_1.c
```

```
  set enable_indexscan = on;
  set enable_bitmapscan = on;

- Complex predicate i.a <= (SELECT AVG(i.a) FROM indexed AS i):
  - startup_cost(i.a <= (SELECT AVG(i.a) FROM indexed AS i))
      = run_cost(SELECT AVG(i.a) FROM indexed AS i)
      = 21846.01
  - run_cost(i.a <= (SELECT AVG(i.a) FROM indexed AS i))
      = 2 × cpu_operator_cost ⬅ · :: numeric, · <= $0 ($0 is a constant once InitPlan 1 has been evaluated)
      = 2 × 0.0025
  - sel(i.a <= (SELECT AVG(i.a) FROM indexed AS i) = 333333 / 1000000 = 0.33 ⬅ ⚠ arbitrary (1/3)


- startup_cost  = startup_cost(pred) + startup_cost(expr)
                = 21846.01 + 0
                = 21846.01 ✓

- cpu_run_cost  =   #rows(indexed) × (cpu_tuple_cost + run_cost(pred))
                  + #rows(indexed) × sel(pred) × run_cost(expr)
                =   10⁶ × (0.01 + 2 × 0.0025)
                  + 10⁶ × 0.33 × 0
                = 15000.0

- disk_run_cost = #pages(indexed) × seq_page_cost
                = 9346 × 1.0
                = 9346.00

- total_cost    = startup_cost + cpu_run_cost + disk_run_cost
                = 21846.01 + 15000.0 + 9346.00
                = 46192.01 ✓
```

Cost of Index Scan ①

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).

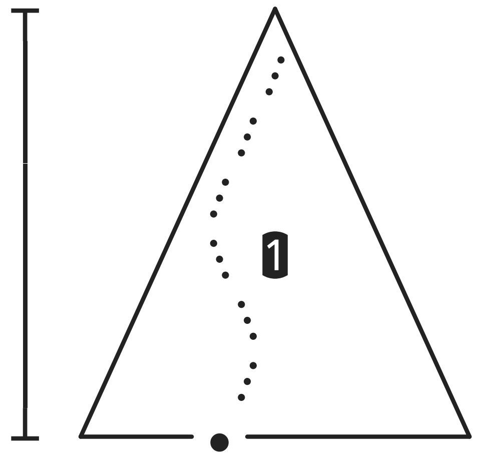
Cost of Index Scan ❷

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 $h = \log_{2\times o}(\#\text{rows}(idx))$

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

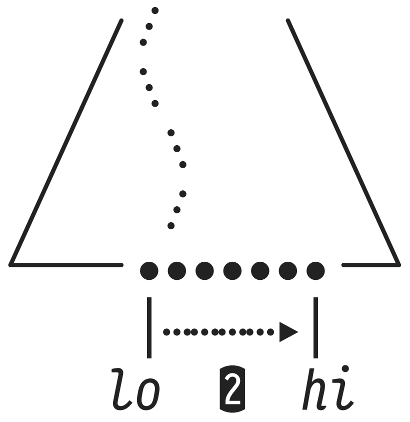
$$\lceil \underbrace{\log_2(2 \times o)}_{\text{binary search in inner B⁺Tree node with fan-out } F = 2 \times o} \times h\rceil = \lceil \log_2(\#\text{rows}(idx))\rceil$$

startup_cost ≝
   startup_cost(*pred*) + startup_cost(*expr*)
 $+\ (\underbrace{\lceil \log_2(\#\text{rows}(idx))\rceil}_{\text{B⁺Tree descent}} + \underbrace{(h + 1)}_{\vdots\ +\ \bullet} \times \underbrace{50}_{\text{index node processing}}) \times \text{cpu\_operator\_cost}$

$$
\begin{aligned}
&\log_2(2 \times o) \times \mathrm{h} \\
&= \log_2(2 \times o) \times \log_{2 \times o}(\#\mathrm{entries}(idx)) \\
&= \frac{\log(2 \times o)}{\log(2)} \times \frac{\log(\#\mathrm{entries}(idx))}{\log(2 \times o)} \\
&= \frac{\log(\#\mathrm{entries}(idx)))}{\log(2)} \\
&= \log_2(\#\mathrm{entries}(idx)))
\end{aligned}
$$

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:**

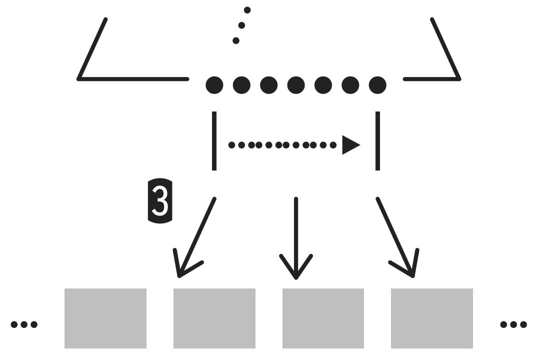


$$\text{index\_cpu\_cost} \overset{\text{def}}{=} \overbrace{\text{sel}(\mathit{pred}) \times \#\text{rows}(\mathit{idx})}^{\#\text{ rows in scanned range } |\cdots\blacktriangleright|} \times (\underbrace{\text{cpu\_index\_tuple\_cost}}_{\text{decode index leaf entry}} + \underbrace{\text{run\_cost}(\mathit{pred})}_{\text{evaluate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

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

Cost of Index Scan ❺ (Heap File Access)

Heap file accesses ❸ incur additional CPU and I/O costs (no I/O cost if we perform an Index **Only** Scan):

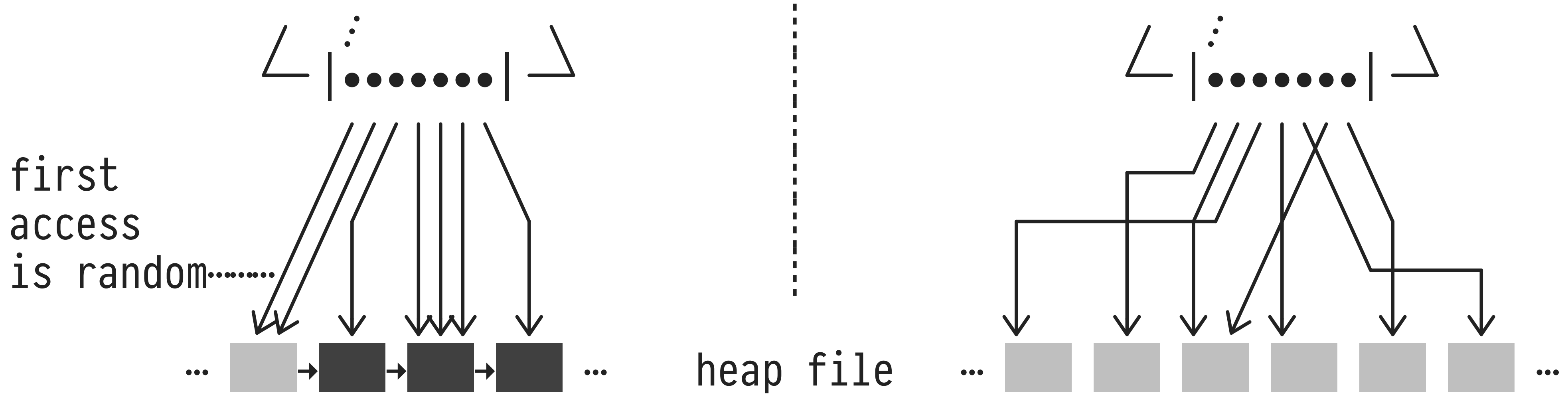


$$\text{heap\_cpu\_cost} \overset{\text{def}}{\equiv} \overbrace{\text{sel}(\mathit{pred}) \times \#\text{rows}(\mathit{in})}^{\text{\# rows accessed in heap file}} \times (\underbrace{\text{cpu\_tuple\_cost}}_{\text{decode row on ▨}} + \underbrace{\text{run\_cost}(\mathit{expr})}_{\text{evaluate Output}})$$

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

$$\text{heap\_IO\_cost} \overset{\text{def}}{\equiv} \text{unclustered\_IO\_cost} + \underbrace{\text{corr}(\mathit{idx})^2}_{\approx \text{ clustering factor } \in \{0,\ldots,1\}} \times (\text{clustered\_IO\_cost} - \text{unclustered\_IO\_cost})$$

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

Index Correlation (Clustering Factor)

Given ordered index *idx* over column A with values $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$, where $\mathrm{pos}(a_i) \in \{1,\dots,n\}$ gives the position of a_i in the heap file for A.[6]

- **Index Correlation** $\mathrm{corr}(idx) \in \{-1,\dots,1\}$ measures how far $[\mathrm{pos}(a_1),\dots,\mathrm{pos}(a_n)]$ deviates from $[1,\dots,n]$, *i.e.*, *idx*'s clustering degree:

$$\mathrm{corr}(idx) = \frac{n \times (\sum_{i=1\dots n} i \times \mathrm{pos}(a_i)) - (\sum_{i=1\dots n} i)^2}{n \times (\sum_{i=1\dots n} i \times i) - (\sum_{i=1\dots n} i)^2}$$

[6] After CLUSTER ‹table› USING *idx*, we have $\mathrm{pos}(a_i) = i$ and thus $\mathrm{corr}(idx) = 1$.

Demonstrate the cost derivation for Seq Scan on table indexed.

```
-- ❶ Prepare input table and indexes

  CREATE INDEX indexed_c ON indexed USING btree (c);
  CLUSTER indexed USING indexed_c;
  ANALYZE indexed;

  \d indexed
                Table "public.indexed"
┌────────┬──────────────┬───────────┬──────────┬─────────┐
│ Column │     Type     │ Collation │ Nullable │ Default │
├────────┼──────────────┼───────────┼──────────┼─────────┤
│ a      │ integer      │           │ not null │         │
│ b      │ text         │           │          │         │
│ c      │ numeric(3,2) │           │          │         │
└────────┴──────────────┴───────────┴──────────┴─────────┘
Indexes:
    "indexed_a" PRIMARY KEY, btree (a)
    "indexed_c" btree (c) CLUSTER


-- ❷ Obtain meta data about table indexed

  SELECT relname, reltuples AS "#rows(•)", relpages AS "#pages(•)"
  FROM   pg_class
  WHERE  relname LIKE 'indexed%';
┌───────────┬──────────┬───────────┐
│  relname  │ #rows(•) │ #pages(•) │
├───────────┼──────────┼───────────┤
│ indexed   │    1e+06 │      9343 │
│ indexed_a │    1e+06 │      2745 │
│ indexed_c │    1e+06 │      2745 │
└───────────┴──────────┴───────────┘

  SELECT correlation AS "corr(indexed_a)"
  FROM   pg_stats
  WHERE  tablename = 'indexed' AND attname = 'a';
┌─────────────────┐
│ corr(indexed_a) │
├─────────────────┤
│      0.00518881 │
└─────────────────┘

  SELECT level AS "h(indexed_a)"
  FROM   bt_metap('indexed_a');
```

```
┌──────────────┐
│ h(indexed_a) │
├──────────────┤
│            2 │
└──────────────┘


-- ❸ Perform a index range scan over a non-clustered index
--   (cf. with Seq Scan query ❺ above which had cost=0.00..22342.17)

  set enable_bitmapscan = off;
  set enable_seqscan = off;

  EXPLAIN VERBOSE
    SELECT i.c * 2 + 1
    FROM indexed AS i
    WHERE i.a <= 100000;
┌─────────────────────────────────────────────────────────────────────────────────────────────┐
│                                          QUERY PLAN                                         │
├─────────────────────────────────────────────────────────────────────────────────────────────┤
│ Index Scan using indexed_a on public.indexed i  (cost=0.42..40779.96 rows=101712 width=32) │
│   Output: ((c * '2'::numeric) + '1'::numeric)                                               │
│   Index Cond: (i.a <= 100000)                                                               │
└─────────────────────────────────────────────────────────────────────────────────────────────┘

- sel(pred): sel(i.a <= 100000) = 101712 / 1000000 = 0.101712

- startup_cost =   startup_cost(pred) + startup_cost(expr)
                 + (⌈log₂(#rows(idx))⌉ + (h + 1) × 50) × cpu_operator_cost
               =   0 + 0
                 + (⌈log₂(10⁶)⌉ + (2 + 1) × 50) × 0.0025
               =   0.42 ✓

- index_cpu_cost =   sel(pred) × #rows(indexed_a)
                   × (cpu_index_tuple_cost + run_cost(pred))
                 =   0.101712 × 10⁶
                   × (0.005 + 1 × 0.0025)   ◀  run_cost(pred) = 1 × cpu_operator_cost: · <= 100000
                 = 762.84

- index_IO_cost  = ⌈sel(pred) × #pages(indexed_a)⌉ × random_page_cost
                 = ⌈0.101712 × 2745⌉ × 4.0
                 = 1120

- heap_cpu_cost  =   sel(pred) × #rows(indexed)
                   × (cpu_tuple_cost + run_cost(expr))
                 =   0.101712 × 10⁶
                   × (0.01 + 2 × 0.0025)   ◀  run_cost(expr) = 2 × cpu_operator_cost: · * 2, · + 1
```

```
                = 1525.68

- clustered_IO_cost = 1 × random_page_cost + (sel(pred) * #pages(indexed) - 1) × seq_page_cost
                    = 1 × 4.0 + (0.101712 × 9343 - 1) * 1.0
                    = 953.295

- unclustered_IO_cost = #pages(indexed) × random_page_cost
                      = 9343 × 4.0
                      = 37372.0

- corr(indexed_a) = 0.00518881

- heap_IO_cost =   unclustered_IO_cost
                 + corr(indexed_a)² × (clustered_IO_cost - unclustered_IO_cost)
               =   37372.0
                 + 0.00518881² × (953.295 - 37372.0)
               =   37371.0194

- total_cost = startup_cost + index_cpu+cost + index_IO_cost
                            + heap_cpu_cost  + heap_IO_cost
             = 0.42 + 762.84 + 1120
                    + 1525.68 + 37371.0194
             = 40779.9594 ✓




-- ❹ Perform a index range scan over a clustered index
--   (cf. with Seq Scan query ❺ above which had cost=0.00..22342.17)

  CLUSTER indexed USING indexed_a;
  ANALYZE indexed;

  SELECT correlation AS "corr(indexed_a)"
  FROM   pg_stats
  WHERE  tablename = 'indexed' AND attname = 'a';
```

corr(indexed_a)
1

```
  EXPLAIN VERBOSE
    SELECT i.c * 2 + 1
    FROM indexed AS i
    WHERE i.a <= 100000;
```

```
                                        QUERY PLAN
┌────────────────────────────────────────────────────────────────────────────┐
│ Index Scan using indexed_a on public.indexed i  (cost=0.42..4299.33 rows=100218 width=32)
│   Output: ((c * '2'::numeric) + '1'::numeric)
│   Index Cond: (i.a <= 100000)
└────────────────────────────────────────────────────────────────────────────┘

- sel(pred): sel(i.a <= 100000) = 100218 / 1000000 = 0.100218

- startup_cost =   startup_cost(pred) + startup_cost(expr)
                 + (⌈log₂(#rows(idx))⌉ + (h + 1) × 50) × cpu_operator_cost
               =   0 + 0
                 + (⌈log₂(10⁶)⌉ + (2 + 1) × 50) × 0.0025
               =   0.42 ✓

- index_cpu_cost =   sel(pred) × #rows(indexed_a)
                   × (cpu_index_tuple_cost + run_cost(pred))
                 =   0.100218 × 10⁶
                   × (0.005 + 1 × 0.0025)   ◀  run_cost(pred) = 1 × cpu_operator_cost: · <= 100000
                 = 751.63

- index_IO_cost  = ⌈sel(pred) × #pages(indexed_a)⌉ × random_page_cost
                 = ⌈0.100218 × 2745⌉ × 4.0
                 = 1104

- heap_cpu_cost  =   sel(pred) × #rows(indexed)
                   × (cpu_tuple_cost + run_cost(expr))
                 =   0.100218 × 10⁶
                   × (0.01 + 2 × 0.0025)   ◀  run_cost(expr) = 2 × cpu_operator_cost: · * 2, · + 1
                 = 1503.27

- clustered_IO_cost = 1 × random_page_cost + (sel(pred) * #pages(indexed) - 1) × seq_page_cost
                    = 1 × 4.0 + (0.100218 × 9343 - 1) * 1.0
                    = 939.336

- unclustered_IO_cost = #pages(indexed) × random_page_cost
                      = 9343 × 4.0
                      = 37372.0

- corr(indexed_a) = 1.0

- heap_IO_cost =   unclustered_IO_cost
                 + corr(indexed_a)² × (clustered_IO_cost - unclustered_IO_cost)
               =   37372.0
                 + 1.0² × (939.336 - 37372.0)
               =   939.336
```

```
- total_cost = startup_cost + index_cpu+cost + index_IO_cost
                            + heap_cpu_cost  + heap_IO_cost
             = 0.42 + 751.63 + 1104
                    + 1503.27 + 939.336
             = 4298.656  ✓
```

```
-- ❺ Perform index-ONLY scan over a clustered index

  -- make sure, dead rows are removed (visibility map update)
  VACUUM;

  EXPLAIN VERBOSE
    SELECT i.a * 2 + 1
    FROM indexed AS i
    WHERE i.a <= 100000;
```

```
                                   QUERY PLAN
 ------------------------------------------------------------------------------------
 Index Only Scan using indexed_a on public.indexed i  (cost=0.42..3359.33 rows=100218 width=4)
   Output: ((a * 2) + 1)
   Index Cond: (i.a <= 100000)
```

```
- sel(pred): sel(i.a <= 100000) = 100218 / 1000000 = 0.100218

- startup_cost =   startup_cost(pred) + startup_cost(expr)
                 + (⌈log₂(#rows(idx))⌉ + (h + 1) × 50) × cpu_operator_cost
               =   0 + 0
                 + (⌈log₂(10⁶)⌉ + (2 + 1) × 50) × 0.0025
               =   0.42 ✓

- index_cpu_cost =   sel(pred) × #rows(indexed_a)
                   × (cpu_index_tuple_cost + run_cost(pred))
                 =   0.100218 × 10⁶
                   × (0.005 + 1 × 0.0025)   ⬅  run_cost(pred) = 1 × cpu_operator_cost: · <= 100000
                 = 751.63

- index_IO_cost  = ⌈sel(pred) × #pages(indexed_a)⌉ × random_page_cost
                 = ⌈0.100218 × 2745⌉ × 4.0
                 = 1104

-- We do not access the heap file to process the rows but still need to
-- process the rows read off the index leaves, thus heap_cpu_cost > 0 even
-- for an Index Only Scan:

- heap_cpu_cost  =   sel(pred) × #rows(indexed)
                  × (cpu_tuple_cost + run_cost(expr))
                 =   0.100218 × 10⁶
                  × (0.01 + 2 × 0.0025)   ⬅  run_cost(expr) = 2 × cpu_operator_cost: · * 2, · + 1
                 = 1503.27

- heap_IO_cost   =  0 (Index Only Scan)

- total_cost = startup_cost + index_cpu+cost + index_IO_cost
                            + heap_cpu_cost  + heap_IO_cost
             = 0.42 + 751.63 + 1104
                    + 1503.27 + 0
             = 3359.32  ✔
```

TODO:

- Explain selectivity estimation in PostgreSQL [mcv, histograms, correlation and CREATE STATISTICS?]
- Explain cost model for a join operator (see http://www.interdb.jp/pg/pgsql03.html for cost models for (Index) Nested Loop Join, Merge Join)