DB 2

14 – Query Optimization

Summer 2020

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 (≈ *SF* × 1.5×10^6 rows)

l_orderkey	l_linenumber	l_partkey	l_quantity	l_extendedprice	…
o					

lineitem (≈ *SF* × 6×10^6 rows)

c_custkey	c_name	c_acctbal	c_nationkey	…
c			*n*	

customer (≈ *SF* × 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. 😱

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.

3 ¦ Pre-Processing: Query Unnesting

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

```
SELECT c.c_name                                        1
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                                    2
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 e1                                         SELECT DISTINCT e1
FROM   q1,…,qi                                    FROM   q1,…,qi,qi+1,…,qn
WHERE  p1                                    ≜*   WHERE  p1
  AND  e2 IN (SELECT e3                             AND  e2 = e3
              FROM   qi+1,…,qn                      AND  p3
              WHERE  p3)
```

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

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

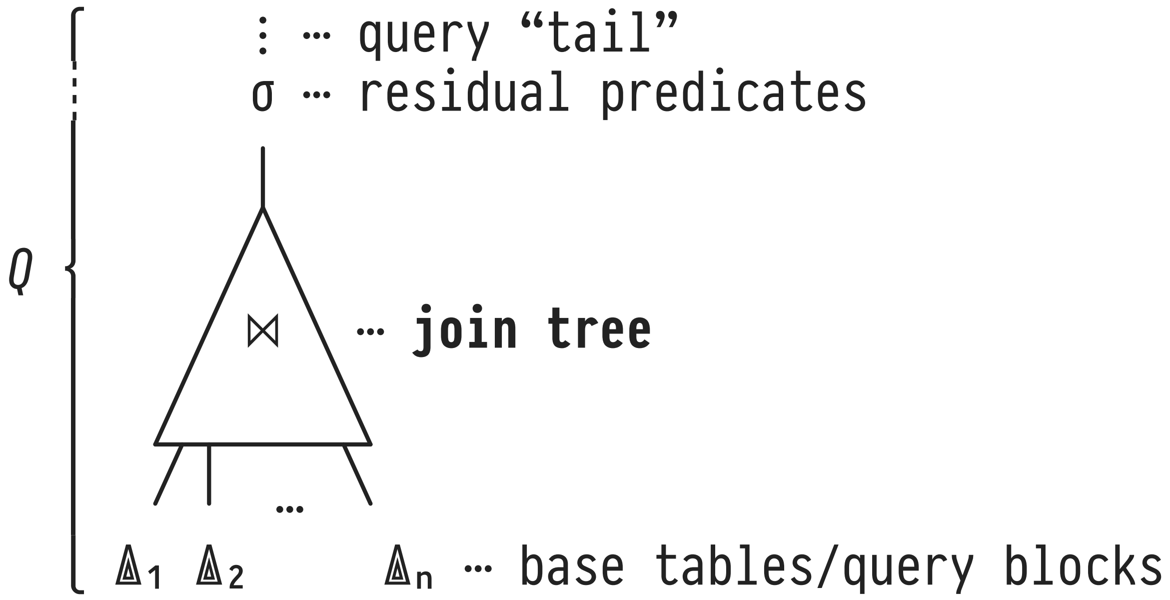
Processing a SQL query Q starts out with its FROM and WHERE clauses which describe a **join tree** over Q's inputs:

SQL

Q:

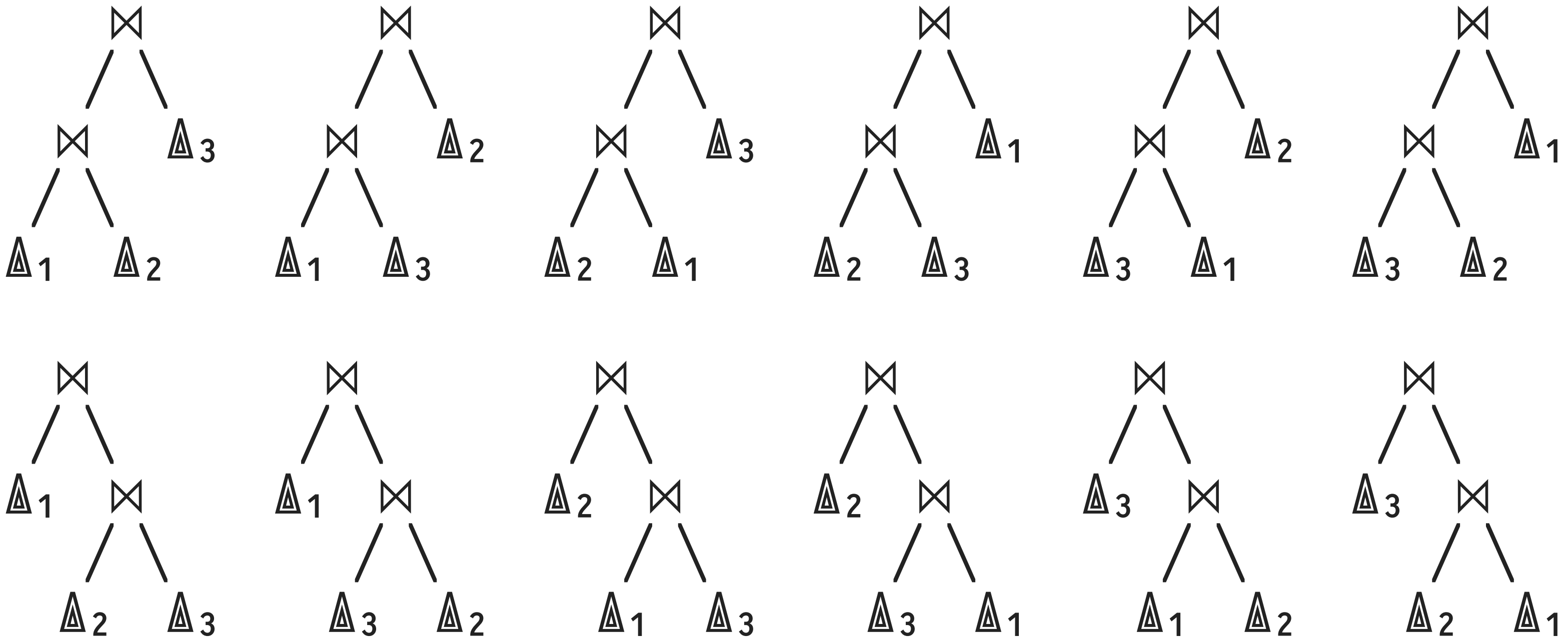
⋮
FROM $\mathbb{A}_1$, $\mathbb{A}_2$, …, $\mathbb{A}_n$
WHERE p
⋮

Canonical Plan



Join Tree Optimization

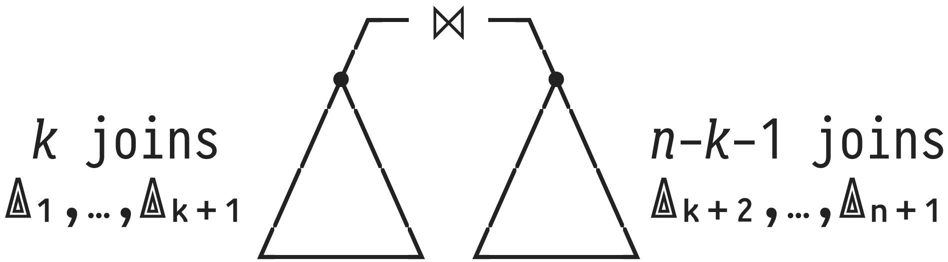
Given n join inputs, the number of possible **join tree shapes** is *huge*. Consider $n = 3$:



- Shapes based on associativity and commutativity of $\bowtie$.

How Many Possible Join Trees Are There?

1. A join of $n+1$ inputs $\triangle$ requires n binary joins. The root $\bowtie$ 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 $\triangle$ 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 $\alpha = 3$ 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 $opt[\{\mathbb{A}_1,\dots,\mathbb{A}_n\}]$ that joins n inputs $\mathbb{A}_1,\dots,\mathbb{A}_n$.

1. **Iteration 1:** For each $\mathbb{A}_j$, find and memorize **best 1-input plan** $opt[\{\mathbb{A}_j\}]$ 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
   - the best $(k-i)$-input plans.

   simple lookups in $opt[\cdot]$ 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

$$\begin{aligned} opt[\{\mathbb{A}_1,\mathbb{A}_2,\mathbb{A}_3\}] \leftarrow prune(&opt[\{\mathbb{A}_1\}] \circledast opt[\{\mathbb{A}_2,\mathbb{A}_3\}] \cup \\ &opt[\{\mathbb{A}_2\}] \circledast opt[\{\mathbb{A}_1,\mathbb{A}_3\}] \cup \\ &opt[\{\mathbb{A}_3\}] \circledast opt[\{\mathbb{A}_1,\mathbb{A}_2\}] \;) \end{aligned}$$

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

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

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({▲₁,…,▲ₙ}):
 foreach p ∈ {▲₁,…,▲ₙ}                              } 1-input plans
  ⌊ opt[{p}] ← prune(access(p));

 for k in 2,…,n                                     } k-input plans
  │ foreach S ⊆ {▲₁,…,▲ₙ} with |S| = k              } enumerate subsets
  │  │ opt[S] ← ϕ;
  │  │ foreach T ⊂ S with T ≠ ϕ
  │  │  ⌊ opt[S] ← opt[S] ∪ { opt[T] ⋈ᵃ opt[S \ T] };
  ⌊  ⌊  opt[S] ← prune(opt[S]);

return opt[{▲₁,…,▲ₙ}];
```

- *access*(·), *prune*(·) defined as above, $\bowtie^a$ 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).