Query Processing: Query Planning in (Object) Relational Systems

Dirk Van Gucht[1]

[1] Indiana University

Query Plans

Let Q denote a SQL statement (query, insert, delete etc)

- A query plan for Q can be obtained using the statement '`explain` Q'
  - this statement returns a parse tree whose leaf nodes are operands and whose internal nodes are operators; the nodes are annotated with time and space costs estimates
  - this parse tree designates the algorithm selected by the system to process Q
- A query plan without cost values can be obtained using the statement '`explain (costs off)` Q'
- A query plan with execution costs can be obtained using the statement '`explain analyze` Q'

Query plan configuration parameters (algorithms)

Certain algorithms/operations used by the system can be enabled or disabled during query planning and optimization

This is a mechanism to force the query compiler to use certain algorithms

The following table shows certain query plan configuration parameters and how they can be enabled or disabled

Algorithm	enable	disable
nested loops	set enable_nestloop = on	set enable_nestloop = off
mergejoin	set enable_mergejoin = on	set enable_mergejoin = off
hashjoin	set enable_hashjoin = on	set enable_hashjoin = off
sort	set enable_sort = on	set enable_sort = off
hash aggregation	set enable_hashagg = on	set enable_hashagain = off
main memory materialization	set enable_material = on	set enable_material = off

For more detail in PostgreSQL version 13:

https://www.postgresql.org/docs/current/runtime-config-query.html

Query plan configuration parameters (Memory and processors)

- It is possible to set the size of working memory

`set_workmemo = '64kB'`	smallest
`set_workmemo = '4MB'`	default
`set_workmemo = '1GB'`	depends on size of main memory availability

- It is possible to affect the parallel CPU cores (workers)

`set max_worker_processes = 1`	maximally 1 worker process
`set max_worker_processes = 2`	maximally 2 worker processes
...	

Objectives

- Query plans for Relational Algebra Expressions
- Query plans for queries with aggregate functions, GROUP BY [HAVING], and WINDOW functions
- Query plans for SQL queries with subqueries
  - Queries in Pure SQL with set predicates
  - Optimized versions of these queries in RA SQL
- Time complexity $O(\cdot)$ in terms of size of relations; throughout, B denotes the size of the buffer.[1]

[1] Note that in this complexity we typically ignore the output size of the query. This is reasonable since any algorithms has the same complexity in this regard.

Query plan configuration parameters

- Force nested loop join without in-memory materialization:

```
set enable_mergejoin = off; set enable_hashjoin = off; set enable_material = off;
```

- Force hash join:

```
set enable_hashjoin = on; set enable_nestloop = off; set enable_mergejoin = off;
```

- Force merge join:

```
set enable_mergejoin = on; set enable_nestloop = off; set enable_hashjoin = off;
```

Query plans for Relation Algebra (RA) expressions

Let R denote a relation name and E and F denote RA expressions. Let A denote an attribute name and **a** a constant.

RA expression
$(A : \mathbf{a})$
R, S, etc
$\sigma_C(E)$
$\pi_L(E)$
$E \cup F$
$E \cap F$
$E - F$
$E \times F$
$E \bowtie_C F$
$E \bowtie F$
$E \ltimes F$
$E \overline{\ltimes} F$

Query plan for $(A : \mathbf{a})$

```
select 'a' as A
```

```
Query Plan
---------
 Result
(1 row)

                                                   Query Plan with Costs
-------------------------------------------------------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=32) (actual time=0.001..0.001 rows=1 loops=1)
 Planning Time: 0.006 ms
 Execution Time: 0.007 ms
(3 rows)
```

$O(1)$

Query plan for R

Assume `R(x, y)`

```
select x, y
from   R
```

Implemented using a sequential scan (**Seq scan**)

```
Query Plan
---------------
 Seq Scan on r
(1 row)
```

$O(|R|)$

Query plans for selection $\sigma_C(E)$

- Sequential scan with filter conditions
  - **Seq scan** and **Filter**
- Index scan with index condition with optional filter
  - **Index Scan** and **Index Cond** [**filter**]

Query plan for $\sigma_C(E)$ with sequential scan

$$\sigma_{x=2 \wedge \neg(y>5)}(R)$$

```
select x, y
from   R
where  x = 2 and not(y > 5)
```

```
              Query Plan
-----------------------------------
 Seq Scan on r
   Filter: ((y <= 5) AND (x = 2))
```

$O(|R|)$

Query plan for $\sigma_C(E)$ with B$^+$-tree index

$$\sigma_{x \leq 2 \wedge x \leq 200}(R)$$

```
create index on R using Btree(x);

select x, y
from   R
where  2 <= x and x <= 200
```

```
                  Query Plan
-----------------------------------------
 Index Scan using index_x on r
   Index Cond: ((x >= 2) AND (x <= 200))
```

$O(\log_B(|R|))$

Query plan for $\sigma_C(E)$ with B$^+$ tree index and filter

$$\sigma_{x=1 \wedge y \leq 3}(R)$$

```
create index on R using Btree(x);

select x, y
from   R
where  x = 1 and y <= 3
```

```
              Query Plan
-----------------------------------
 Index Scan using index_x on r
   Index Cond: (x = 1)
   Filter: (y <= 3)
```

$O(\log_B(|R|))$

Query plans for Projection $\pi_L(E)$

- Sort-based projection with sort key on L followed by duplicate elimination
    - **Sort** by **Sort Key**, duplicate elimination **Unique**

- Hash-based projection with hash aggregation on group key L.

  Hash aggregation scans the tuples in E and places them in buckets based on their group key L components.

  Tuples with the same L components are mapped to the same bucket.

    - **HashAggregate** by **Group Key**

Query plan for $\pi_L(E)$ using sorting on L and duplicate elimination

Assume $\mathtt{R}(\mathtt{x}, \mathtt{y})$

```
select distinct x
from   R
```

```
             Query Plan
------------------------------
Unique
   ->   Sort
          Sort Key: x
          ->   Seq Scan on r
```

$O(|R| \log_B |R|)$

Query plan for $\pi_L(E)$ using hash aggregation on L

Assume $\mathtt{R}(\mathtt{x}, \mathtt{y})$

```
select distinct x
from   R
```

```
        Query Plan
---------------------
HashAggregate
    Group Key: x
  ->   Seq Scan on r
```

$O(|R|)$

Set operation $E \cup F$

Recall that `UNION` is a set operation in SQL and, as such, duplicate elimination is performed

- Sort-based Union:
  E and F are appended and then sorted on the sort key which is the schema of E; subsequently duplicate elimination is performed
  - **Append**, **Sort** by **Sort Key**, followed by **Unique**
- Hash-based Union:
  E and F are appended and then hash aggregated on the group key corresponding to the schema of E. This will guarantee that a tuple that is both in E and F will be mapped to the same bucket.
  - **Append**, **HashAggregate** by **Group Key**

Query plan for $E \cup F$ using sorting

Assume $\mathtt{R}(x, y)$ and $\mathtt{S}(x, y)$

```
select * from R
union
select * from S
```

```
               Query Plan
---------------------------------
 Unique
   ->  Sort
         Sort Key: r.x, r.y
         ->  Append
               ->  Seq Scan on r
               ->  Seq Scan on s
```

$O((|R| + |S|)\log_B(|R| + |S|))$

Query plan for $E \cup F$ using hash aggregation

Assume $\texttt{R}(x, y)$ and $\texttt{S}(x, y)$

```
select * from R
union
select * from S
```

```
     Query Plan
---------------------------
HashAggregate
   Group Key: r.x, r.y
  ->  Append
        ->  Seq Scan on r
        ->  Seq Scan on s
```

$O(|R| + |S|)$

Set operation $E \cap F$

Note that only tuples that are both in E and F need to be retained

- Sort-based Intersection:

  E and F are appended and then sorted in a list wherein each tuple in E is tagged by the schema of E and each tuple in F is tagged by the schema of F; subsequently this sorted list is scanned and only tuples that have both the E tag and the F tag are retained

  - **Append**, **Sort Key** with tagging, then scanning with **Setop Intersect**

- Hash-based Intersection:

  E and F are appended and then hash aggregated wherein each tuple in E is tagged by the schema of E and each tuple in F is tagged by the schema of F; subsequently the hash table is scanned and only tuples that have both the E tag and the F tag are retained

  - **Append** with tags, then scanning with **HashSetOp Intersect**

Query plan for $E \cap F$ using sorting

Assume $\texttt{R}(x, y)$ and $\texttt{S}(x, y)$

```
select * from R
intersect
select * from S
```

```
                        Query Plan
--------------------------------------------------
 SetOp Intersect
   -> Sort
        Sort Key: "*SELECT* 1".x, "*SELECT* 1".y
        -> Append
              -> Subquery Scan on "*SELECT* 1"
                    -> Seq Scan on r
              -> Subquery Scan on "*SELECT* 2"
                    -> Seq Scan on s
```

$$O((|R| + |S|)\log_B(|R| + |S|))$$

Query plan for $E \cap F$ using hash aggregation

Assume $\texttt{R}(x, y)$ and $\texttt{S}(x, y)$

```
select * from R
intersect
select * from S
```

```
                    Query Plan
------------------------------------------
  HashSetOp   Intersect
    ->     Append
          ->     Subquery Scan on "*SELECT* 1"
                ->     Seq Scan on r
          ->     Subquery Scan on "*SELECT* 2"
                ->     Seq Scan on s
```

$O(|R| + |S|)$

Set operation $E - F$

Note that only tuples that are both in E but and in F need to be retained

- Sort-based Difference:

  E and F are appended and then sorted in a list wherein each tuple in E is tagged by the schema of E and each tuple in F is tagged by the schema of F; subsequently this sorted list is scanned and only tuples that have the E tag but not the F tag are retained

  - **Append**, **Sort Key** with tagging, then scanning with **SetOp Except**

- Hash-based Difference:

  E and F are appended and then hash aggregated wherein each tuple in E is tagged by the schema of E and each tuple in F is tagged by the schema of F; subsequently the hash table is scanned and only tuples that have the E tag but not the F tag are retained

  - **Append** with tags, then scanning with **HashSetOp Except**

Query plan for $E - F$ using sorting

Assume $\mathtt{R}(x, y)$ and $\mathtt{S}(x, y)$

```
select * from R
except
select * from S
```

```
                    Query Plan
------------------------------------------------------------
 SetOp Except
   ->  Sort
         Sort Key: "*SELECT* 1".x, "*SELECT* 1".y
         ->  Append
               ->  Subquery Scan on "*SELECT* 1"
                     ->  Seq Scan on r
               ->  Subquery Scan on "*SELECT* 2"
                     ->  Seq Scan on s
```

$O((|R| + |S|) \log_B(|R| + |S|))$

Query plan for $E - F$ using hash aggregation

Assume $\mathtt{R}(x, y)$ and $\mathtt{S}(x, y)$

```
select * from R
except
select * from S
```

```
                     Query Plan
-------------------------------------------
 HashSetOp Except
   -> Append
        -> Subquery Scan on "*SELECT* 1"
             -> Seq Scan on r
        -> Subquery Scan on "*SELECT* 2"
             -> Seq Scan on s
```

$O(|R| + |S|)$

Query plan for $E \times F$

Since each (E, F) pair needs to be retained this is simply a doubly nested loop.

```
select *
from   R cross join S
```

```
        Query Plan
----------------------
  Nested Loop
   ->   Seq Scan on r
   ->   Seq Scan on s
```

$O\left(|S| + \frac{|R||S|}{B}\right)$ block nested loop

Query plan for $E \times F$ with materialization in buffer

This is the situation where F fits in the buffer and can therefore be loaded entirely into it before E is being scanned.

```
select *
from   R cross join S
```

```
          Query Plan
------------------------------
 Nested Loop
   ->  Seq Scan on r
   ->  Materialize
         ->  Seq Scan on s
```

$O(|S| + |R|)$ when S fits in main memory

Equality joins $E \bowtie_{E.x=F.y} F$ and natural joins $E \bowtie F$

- Block nested-loop join
  - **Nested Loop** with **Join Filter** for equality condition $E.x = F.y$
- Sort-based merge join:
  Sort E on sort key $E.x$; Sort F on sort key $F.y$; Merge on condition $E.x = F.y$.
  - **Sort** E by **Sort Key**; **Sort** F by **Sort Key**; **Merge Join** with **Merge Cond**
- Hash-join:
  Hash E into hash table based on key $E.x$. Then scan F hashing on key $F.y$ and joining with tuples in E that are in the bucket with that key, and thus satisfying the join condition $E.x = F.y$.
  - **Hash Seq Scan** E; **Hash Join**, **Hash Condition**, **Seq Scan** F

Query plan for equality join or $E \bowtie_{E.x=F.y} F$ with block nested-loop join

Assume $\mathtt{R}(\mathtt{x}, \mathtt{y})$ and $\mathtt{S}(\mathtt{y}, \mathtt{z})$

```
select *
from   R join S on (r.y = s.y)
```

```
              Query Plan
------------------------------
  Nested Loop
    Join  Filter: (r.y = s.y)
   ->   Seq Scan on s
   ->   Seq Scan on r
```

$$O\left(|S| + \frac{|R||S|}{B}\right)$$

Query plan for equality join or $E \bowtie_{E.x=F.y} F$ with sort-merge join

Assume `R(x, y)` and `S(y, z)`

```
select *
from   R join S on (r.y = s.y)
```

```
           Query Plan
---------------------------
  Merge Join
    Merge Cond: (s.y = r.y)
   ->     Sort
            Sort Key: s.y
          ->    Seq Scan on s
   ->     Sort
            Sort Key: r.y
          ->    Seq Scan on r
```

$O(|R| \log_B(|R|) + |S| \log_B(|S|))$

Query plan for equality join or $E \bowtie_{E.x=F.y} F$ with hash-join

Assume $\mathtt{R}(\mathtt{x},\mathtt{y})$ and $\mathtt{S}(\mathtt{y},\mathtt{z})$

```
select *
from   R join S on (r.y = s.y)
```

```
            Query Plan
---------------------------
  Hash Join
    Hash Cond: (s.y = r.y)
   ->   Seq Scan on s
   ->    Hash
          ->    Seq Scan on r
```

$O(|R| + |S|)$

Query plan for $E \bowtie_C F$ with block nested-loop join

When the condition C is not an equality join, we can only do a block nested-loop join.
Assume $\mathtt{R}(\mathtt{x},\mathtt{y})$ and $\mathtt{S}(\mathtt{y},\mathtt{z})$ and C is the condition `R.y <> S.y`.

```
select *
from   R join S on (r.y <> s.y)
```

```
              Query Plan
--------------------------------
 Nested Loop
   Join  Filter: (r.y <> s.y)
  ->    Seq Scan on s
  ->    Seq Scan on r
```

$$O\left(|S| + \frac{|R||S|}{B}\right)$$

Query plans for semi-join $E \ltimes F$

- Can be implemented in RA SQL using the same techniques as those for equality join and projections.

  $$E \ltimes F = E \bowtie \pi_x(F)$$

  where x represents the attributes that appear in both E and F
- Can be implemented in Pure SQL using the `IN` set predicate
- Can be implemented in Pure SQL using the `EXISTS` set predicate

Query plan for $E \ltimes F$ in RA SQL

Assume `R(x,y)` and `S(y,z)` and consider $R \ltimes S$

```
select r.*
from   r natural join (select s.y from s) q
```

```
            Query Plan
---------------------------
  Hash Join
    Hash Cond: (s.y = r.y)
   ->   Seq Scan on s
   ->    Hash
          ->   Seq Scan on r
```

$O(|R| + |S|)$

Query plan for $E \ltimes F$ using IN predicate

```
select  r.*
from    R
where   r.y  in  (select s.y from s)
```

```
               Query Plan
---------------------------------
  Hash Join
    Hash Cond: (r.y = s.y)
   ->  Seq Scan on r
   ->  Hash
         ->  Hash Aggregate
               Group Key: s.y
               ->  Seq Scan on s
```

Notice that in this query plan, we build a hash table for $\pi_y(S)$ before we hash join with R.

$O(|R| + |S|)$

Query plan for $E \ltimes F$ with EXISTS predicate

```
select r.*
from   r
where  exists (select 1
               from   s
               where  r.y = s.y)
```

```
             Query Plan
---------------------------------
 Hash Join
   Hash Cond: (r.y = s.y)
  ->   Seq Scan on r
  ->    Hash
        ->    Hash Aggregate
                Group Key: s.y
              ->    Seq Scan on s
```

Note that this is the same query plan as that for the semi-join using the `IN` predicate.

$O(|R| + |S|)$

Query plans for anti semi-join $E \,\overline{\ltimes}\, F$

Observe that

$$E \,\overline{\ltimes}\, F = E - E \ltimes F.$$

- Can be implemented in RA SQL using the same techniques as those for semi-joins and set difference
- Can be implemented in Pure SQL using the `NOT IN` set predicate
- Can be implemented in Pure SQL using the `NOT EXISTS` set predicate

Query plan for anti-semijoin $E \overline{\ltimes} F$ in RA SQL

Recall

$$E \overline{\ltimes} F = E - E \ltimes F.$$

```
select *
from   R
except
select *
from   R natural join (select y from S) q
```

```
                        Query Plan
---------------------------------------------------
  HashSetOp   Except
   ->    Append
         ->    Subquery Scan on "*SELECT* 1"
               ->    Seq Scan on r
         ->    Subquery Scan on "*SELECT* 2"
               ->    Hash Join
                       Hash Cond: (s.y = r_1.y)
                     ->    Seq Scan on s
                     ->    Hash
                           ->    Seq Scan on r r_1
```

Query plan for $E \overline{\ltimes} F$ with **NOT IN** predicate

```
select r.*
from   r
where  r.y not in (select s.y from s)
```

Seq Scan on r
Filter: (NOT (hashed SubPlan 1))
SubPlan 1
-> Seq Scan on s

First a hash table is constructed for S (based on $S.y$ key). Next scan R and retain a tuple $(u, v) \in R$ if its hash value for v is NOT hashed to an non-empty bucket of the hash table for S. I.e., (u, v) is retained if v is hashed to an empty bucket.

$O(|R| + |S|)$

Query plan for $E \overline{\ltimes} F$ with NOT EXISTS predicate

```
select r.*
from   r
where  not exists (select 1
                   from   s
                   where  r.y = s.y)
```

```
             Query Plan
---------------------------
 Hash Anti Join
   Hash Cond: (r.y = s.y)
  ->  Seq Scan on r
  ->  Hash
        ->  Seq Scan on s
```

This query plan is identical to the query plan for the semi-join $E \ltimes F$ in RA SQL, except that there we have a hash semi join and here we have an hash anti-join.

$O(|R| + |S|)$

Query plans for GROUP BY HAVING

- Sorting: GroupAggregate
  - Sorting on a sort key induces a partition of cells. The aggregate function can then be map-applied over these cells
- Hashing: HashAggregate
  - Hashing on a key induces a partition of cells (buckets). The aggregate function can then be map-applied over these cells.

Query plan for *GROUP BY* with sorting

```
select x, count(y)
from   r
group  by (x)
```

```
GroupAggregate
  Group Key: x
 ->    Sort
         Sort Key: x
        ->    Seq Scan on r
```

Sorting R by the key $R.x$ partitions the relation in cells labeled by these key values. Subsequently, the `count` aggregate function is map-applied over these cells. Note that the query plan does not mention the name of the `count` aggregate function. In fact the same query plan would be constructed for any other aggregate function such as `sum`, `min`, `max`, and `avg`.

$$O(|R| \log_B |R|)$$

Query plan for *GROUP BY* with hashing

```
select x, count(y)
from   r
group  by (x)
```

```
         Query Plan
----------------------
 Hash Aggregate
   Group Key: x
   ->   Seq Scan on r
```

The hash-table of R constructed on the key $R.x$ partitions the relation in cells labeled by these key values.

Subsequently, the `count` aggregate function is map-applied over these Note that the query plan does not mention the name of the `count` aggregate function. In fact the same query plan would be constructed for any other aggregate function such as `sum`, `min`, `max`, and `avg`.

$O(|R|)$

Query plan for *GROUP BY HAVING*

```
select x, count(y)
from   r
group  by (x)
having count(y) >= 3
```

```
            Query Plan
------------------------------
 Hash Aggregate
   Group Key: x
   Filter: (count(y) >= 3)
   ->  Seq Scan on r
```

$O(|R|)$