



Does PostgreSQL respond to the challenge of analytical queries?



Lepikhov Andrey

PostgreSQL Thailand Development Group

2024





Self Introduction

- 2007 - First touch to PostgreSQL (distributed query execution)
- 2010 - designer and user of databases based on PostgreSQL (and SQLite)
- 2017 - now - PostgreSQL Enthusiast, sporadic contributor and patch reviewer

My projects:

- Multimaster, Shardman, Joinsel
- AQO, sr_plan, SwitchJoin, re-optimisation ...
- Patches: GROUP-BY, Self-Join Removal, OR->ANY Optimisation ...





Purpose

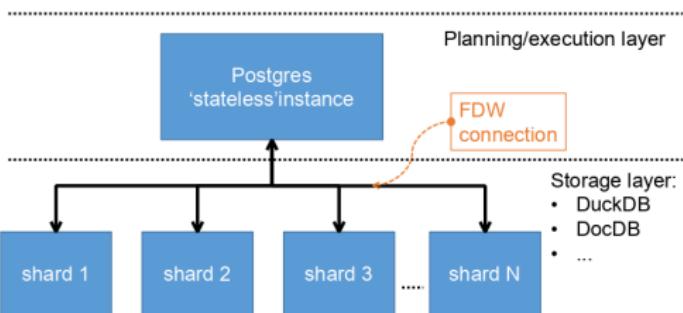
- The scope of the analytic queries problem
- What is the PostgreSQL current progress?
- What is needed to be done (compared to other DBMSes)?





The risky tendency

- YugaByte
- Shardman
- Crunchy Data
DuckDB integration



Greenplum, Citus, and Postgres-XL altered the Postgres optimiser code. However, the new approach uses Postgres almost 'as is'.





What is analytic query?

- It touches a massive part of the tuples in a relation
- Uses a mix of aggregate functions over the same data
- Have a complex query structure (join tree, groupings, limits, sortings, etc)
- Actively uses subqueries and CTEs
- Relations are burdened with many indexes





What is the real progress?

Date	Description
26/03/2024	Propagate pathkeys from CTEs up to the outer query.
19/03/2024	Postpone reparameterization of paths until <code>create_plan()</code>
15/02/2024	pull-up correlated subqueries
21/01/2024	Explore alternative orderings of group-by pathkeys during optimization
17/11/2023	Extract column statistics from CTE references, if possible.
30/01/2023	Make Vars be outer-join-aware
27/01/2023	Teach planner about more monotonic window functions
18/01/2023	Remove redundant grouping and DISTINCT columns
02/08/2022	Improve performance of ORDER BY / DISTINCT aggregates
08/04/2022	Teach planner and executor about monotonic window funcs
02/04/2021	Add Memoize executor node
31/03/2021	Add support for asynchronous execution
06/04/2020	Implement Incremental Sort
27/03/2019	Add support for multivariate MCV lists
09/02/2019	Create the infrastructure for planner support functions
05/12/2017	Support Parallel Append plan nodes
06/10/2017	Basic partition-wise join functionality
05/04/2017	Collect and use multi-column dependency stats
27/03/2017	Support hashed aggregation with grouping sets
24/03/2017	Implement multivariate n-distinct coefficients
09/03/2017	Add a Gather Merge executor node
07/12/2016	Implement table partitioning
12/05/2015	Add support for doing late row locking in FDWs
01/05/2015	Allow FDWs and custom scan providers to replace joins with scans
30/04/2015	Create an infrastructure for parallel computation in PostgreSQL
21/02/2013	Add <code>postgres_fdw</code> contrib module
20/02/2011	Implement an API to let foreign-data wrappers actually be functional
14/10/2010	Support <code>MergeAppend</code> plans, to allow sorted output from append relations
04/01/2010	Fetch the actual column min or max value using an index scan

- Big storage
- Aggregates
- Query structure
- Subqueries & CTEs
- Indexes





- Split data into partitions
- Distribute data (foreign data wrappers)
- Parallel execution techniques
- Smart caching (Memoize, Materialize)



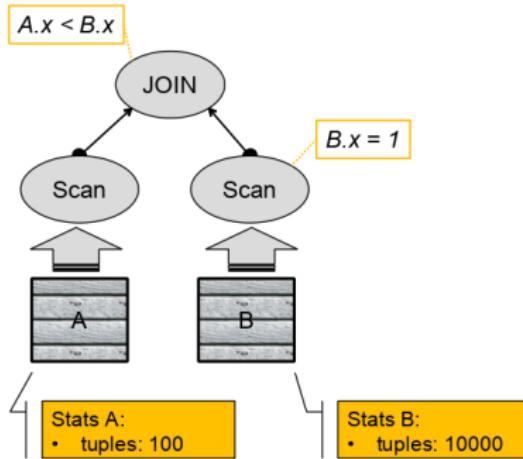
Smart caching



Materialize

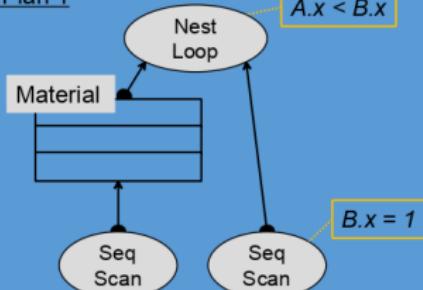


Query Tree

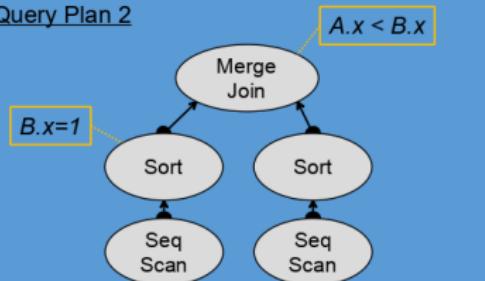


?

Query Plan 1



Query Plan 2



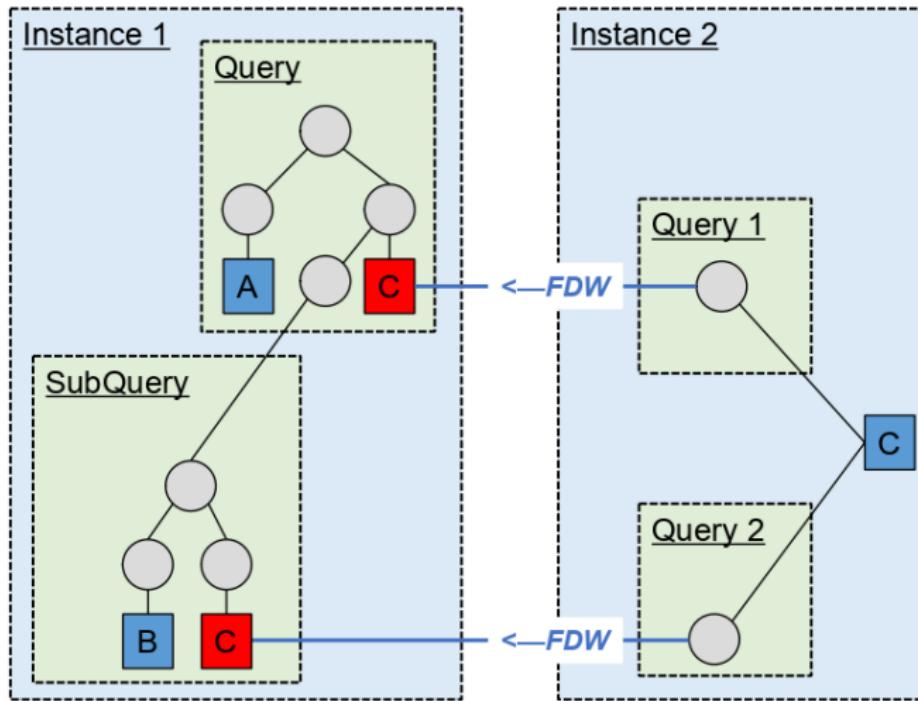


What's more with materialization?

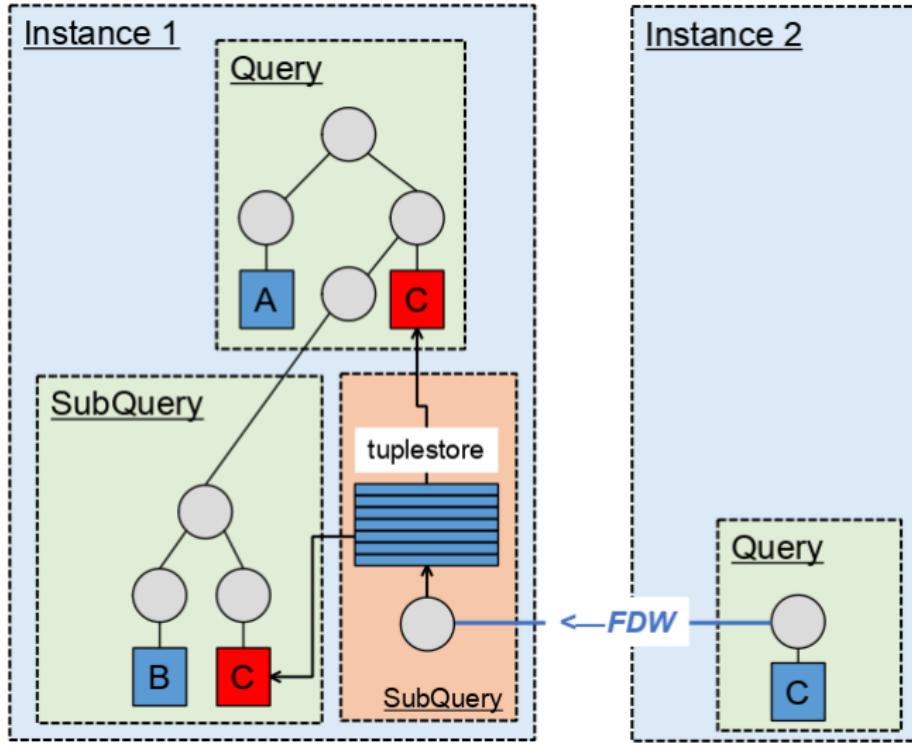
```
SELECT * FROM a LEFT JOIN foreign_table_b b1  
ON b1.x IN (  
    SELECT y FROM foreign_table_b b2  
    WHERE b1.z=a.z  
);
```



What's more with materialization? - II



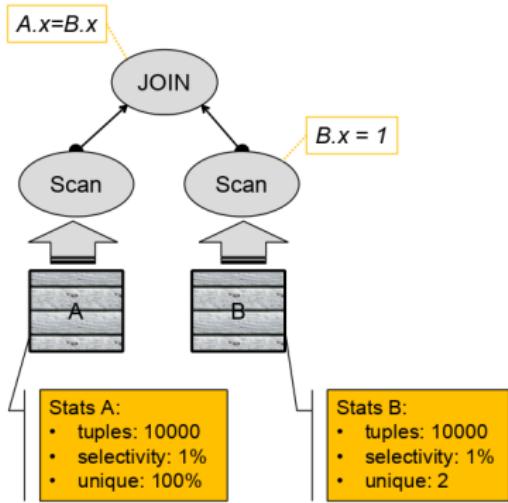
Cross-query materialization





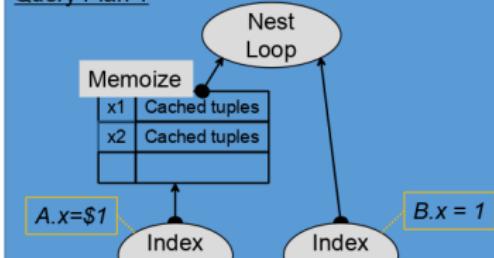
Memoize

Query Tree

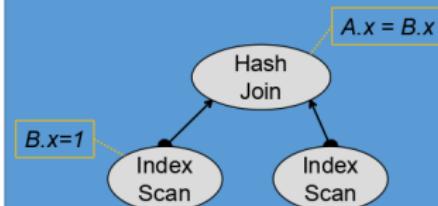


?

Query Plan 1

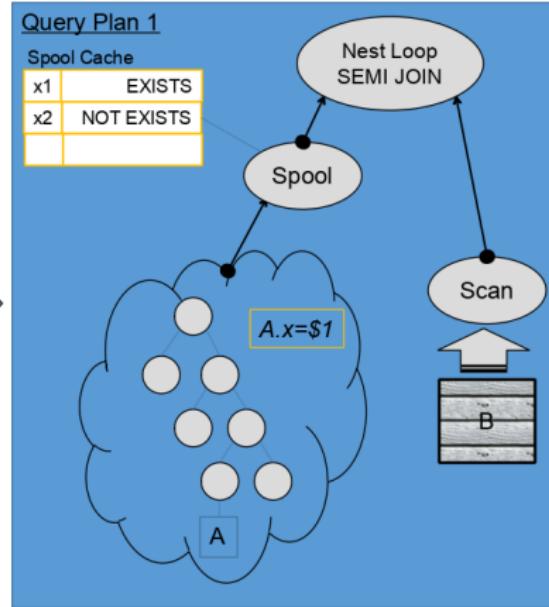
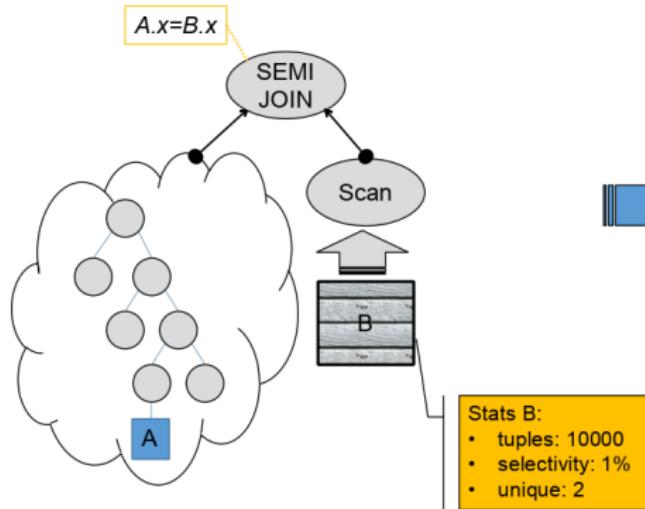


Query Plan 2



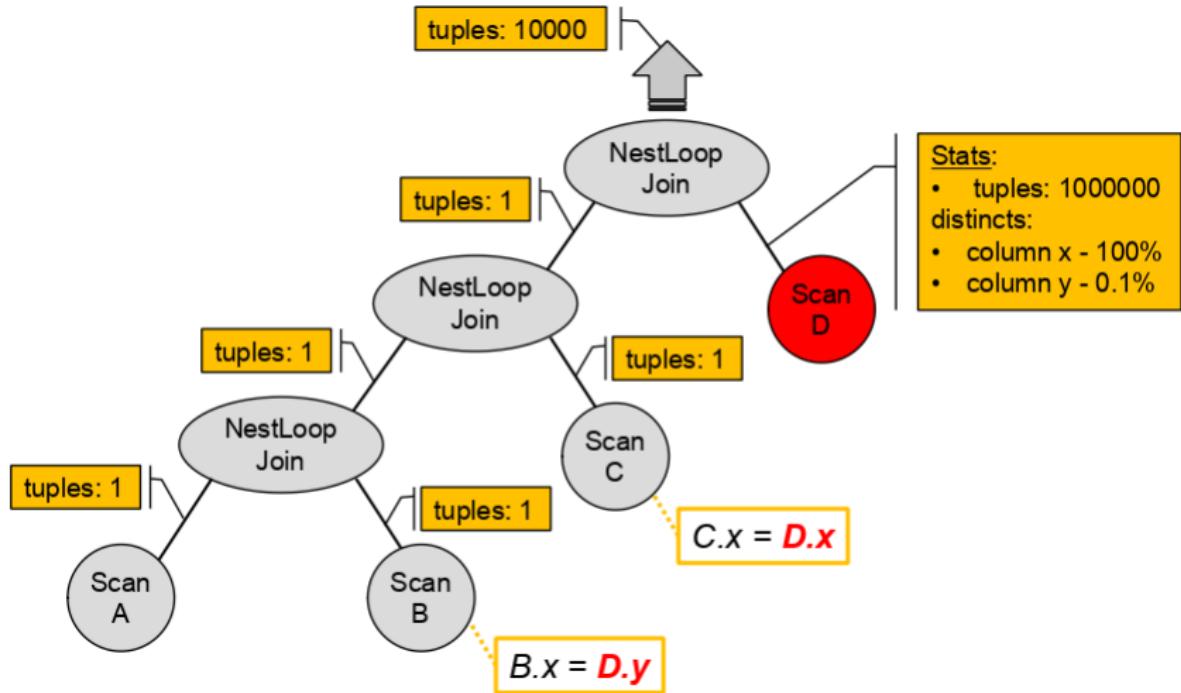


Memoization: What's more?



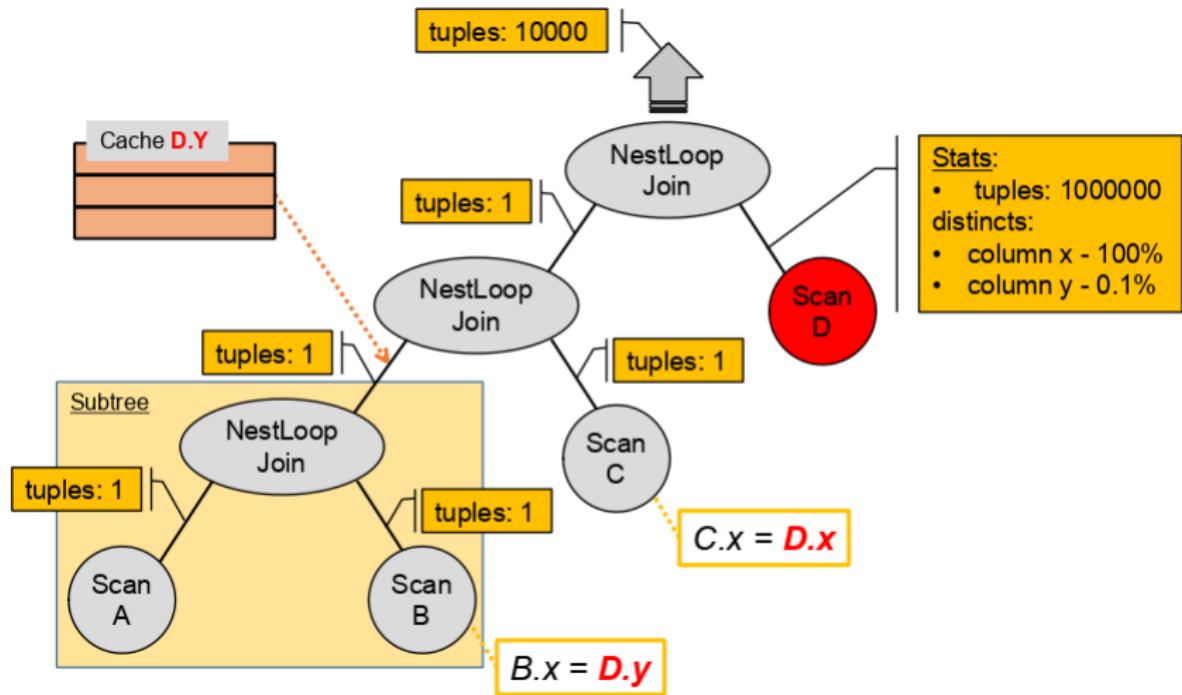


Parameters cache: example





Parameters cache: example





Postgres optimisation passes from base relations to the top GROUP/ORDER-BY operation. We probably need a second, top-bottom optimisation pass to find out points of frequent rescanning and insert caching (Material, Memoize) nodes.



Subqueries & CTE





Subqueries & CTE

- Subquery -> JOIN transformation
- Precalculated Subquery and CTE result
- Use subquery stat (sort order, rows number, uniqueness) at the top-query planning
- Choose the best-fitting subquery plan





Unnesting Correlated Subqueries (CSQ)

```
SELECT * FROM t1 WHERE y IN (  
    SELECT y FROM t2 WHERE t1.x=t2.x);
```

BEFORE Unnesting

Seq Scan **on** t1
Filter: (SubPlan 1)
SubPlan 1
-> Seq Scan **on** t2
Filter: (t1.x = x)

AFTER Unnesting

Nested Loop Semi **Join**
Join Filter: ((t1.x = t2.x) **AND** (t1.y = t2.y))
-> Seq Scan **on** t1
-> Seq Scan **on** t2





Postgres can transform:

```
SELECT * FROM t1 WHERE y IN (  
    SELECT y FROM t2 WHERE t1.x=t2.x);
```

The long list of possible transformations:

- Subquery in select list
- In disjunctive filter
- With GROUP-BY and inequality
- Must return single row (equality operator)
- Subquery references outer side (multilevel case)





CSQ - Just an example

```
CREATE UNIQUE INDEX ON t2(x);
```

```
SELECT * FROM t1 WHERE y = (
    SELECT y FROM t2 WHERE t1.x=t2.x);
```

Proved single row return

Seq Scan **on** t1

Filter: (y = (SubPlan 1))

SubPlan 1

 -> **Index Scan using** t2_x_idx **on** t2

Index Cond: (x = t1.x)





New CTE features

Before 2024, the general idea was to isolate the planning of the query and its subplans to simplify the process. In 2024, there is a big leap ahead: propagation of sorting.

Example:

EXPLAIN (COSTS OFF)

```
WITH x AS MATERIALIZED (
    SELECT unique1 FROM tenk1 b
    ORDER BY unique1
) SELECT count(*) FROM tenk1 a
WHERE unique1 IN (SELECT * FROM x);
```





Sort info propagation - example

Before:

```
Aggregate
CTE x
-> Index Only Scan using tenk1_unique1 on tenk1 b
-> Nested Loop
    -> HashAggregate
        Group Key: x.unique1
        -> CTE Scan on x
-> Index Only Scan using tenk1_unique1 on tenk1 a
    Index Cond: (unique1 = x.unique1)
```

After:

```
Aggregate
CTE x
-> Index Only Scan using tenk1_unique1 on tenk1 b
-> Merge Semi Join
    Merge Cond: (a.unique1 = x.unique1)
-> Index Only Scan using tenk1_unique1 on tenk1 a
-> CTE Scan on x
```

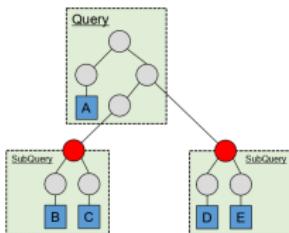




CTE - what's more?

Break membrane between query and subquery planning (maybe on demand):

- Expose alternative paths to the upper query
- Push parameters into a subquery
- Replan subquery if the upper query has additional info



Indexes ?

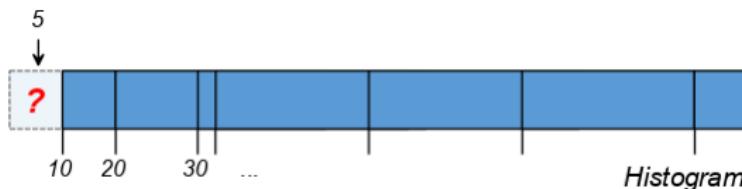




Indexes as an optimisation tool

2010 - inequality estimations with indexes:

```
SELECT * FROM table WHERE x < 5;
```



Optimiser fetches index tuples to estimate cardinality in cases:

- Histogram too simple
- The value hits the first (last) histogram bin
- The value out of the histogram





What's more with indexes

Index estimation is a promising way to tackle optimistic '1-tuple' estimations.

- Direct index fetch is too heavy
- Change Access Method interface to allow index estimations based on the number of pages/average width data.
- Implement Index Estimation for equalities
- Multi-clause filters estimations covered by an index



Aggregates & Window Functions

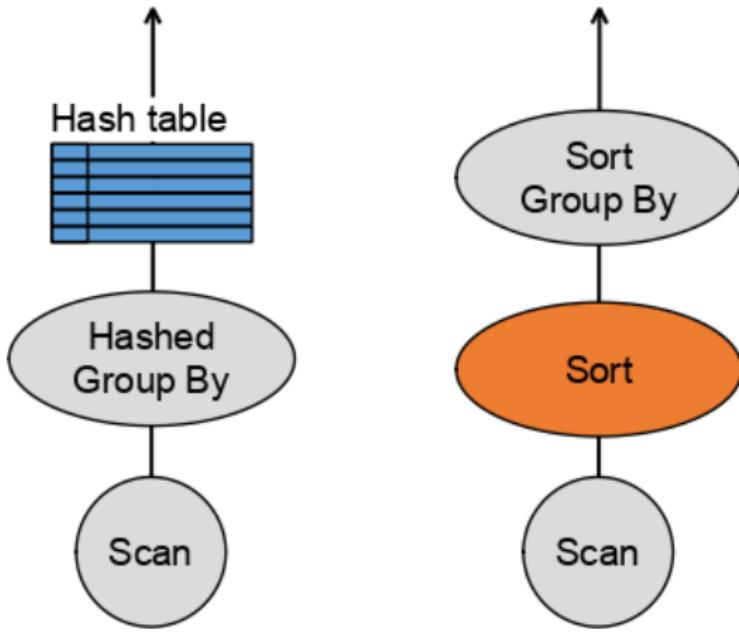




- Hashed Aggregates
- Combine aggregate's sort orders
- Prosupport machinery



Hashed Aggregates

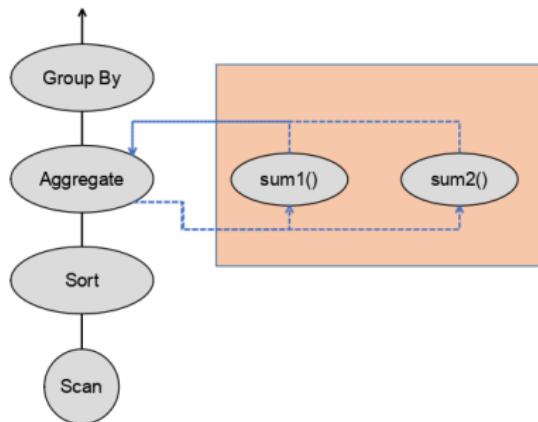
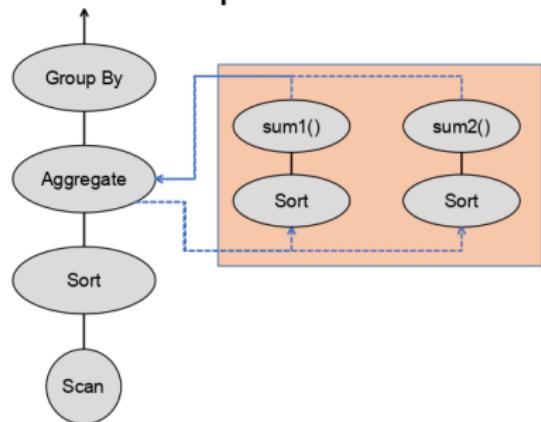




Combine sortings

```
SELECT sum(unique1 ORDER BY ten), sum(unique1 ORDER BY ten,two)
FROM tenk1 GROUP BY ten;
```

Execution plans:





Combine sortings

```
EXPLAIN (ANALYZE, TIMING OFF, COSTS ON)
SELECT sum(unique1 ORDER BY ten), sum(unique1 ORDER BY ten,two)
FROM tenk1 GROUP BY ten;

-- PG13:

GroupAggregate (cost=1108.97..1209.02) (actual rows=10 loops=1)
  Output: sum(unique1 ORDER BY ten), sum(unique1 ORDER BY ten, two), ten
  Group Key: tenk1.ten
    -> Sort (cost=1108.97..1133.95) (actual rows=10000 loops=1)
        Output: ten, unique1, two
        Sort Key: tenk1.ten
          -> Seq Scan on public.tenk1 (cost=0.00..444.95)
              Output: ten, unique1, two
Execution Time: 116.375 ms
```

```
-- PG17:

GroupAggregate (cost=1109.39..1209.49) (actual rows=10 loops=1)
  Output: sum(unique1 ORDER BY ten), sum(unique1 ORDER BY ten, two), ten
  Group Key: tenk1.ten
    -> Sort (cost=1109.39..1134.39) (actual rows=10000 loops=1)
        Output: ten, unique1, two
        Sort Key: tenk1.ten, tenk1.two
          -> Seq Scan on public.tenk1 (cost=0.00..445.00)
              Output: ten, unique1, two
Execution Time: 12.650 ms
```





- A function has been a black box for the planner: rows? cost? selectivity? ...
- Postgres community respond to the challenge - prosupport
- Prosupport - routine which can be attached to any stored procedure
- At the planning stage, Prosupport can provide cost, rows, selectivity recommendations
- full function replacement, index condition
- Window function optimisations





Prosupport machinery - an example

```
PREPARE stmt(int) AS SELECT x FROM test WHERE int4mul(x, $1) < 100;
```

```
EXECUTE stmt(0);
```

QUERY PLAN

```
Seq Scan on test (cost=0.00..20.00 rows=333 width=4) (actual rows=1000 loops=1)
  Filter: (int4mul(x, 0) < 100)
```

```
EXECUTE stmt(1);
```

QUERY PLAN

```
Seq Scan on test (cost=0.00..20.00 rows=333 width=4) (actual rows=99 loops=1)
  Filter: (int4mul(x, 1) < 100)
```

```
EXECUTE stmt(2);
```

QUERY PLAN

```
Seq Scan on test (cost=0.00..20.00 rows=333 width=4) (actual rows=49 loops=1)
  Filter: (int4mul(x, 2) < 100)
```





Prosupport machinery - use help routine

```
UPDATE pg_proc SET prosupport = 'int4mul_support' WHERE proname = 'int4mul';
PREPARE stmt(int) AS SELECT x FROM test WHERE int4mul(x, $1) < 100;
EXECUTE stmt(0);
```

QUERY PLAN

```
Seq Scan on test (cost=0.00..15.00 rows=1000 width=4) (actual rows=1000 loops=1)
```

```
EXECUTE stmt(1);
```

QUERY PLAN

```
Seq Scan on test (cost=0.00..17.50 rows=99 width=4) (actual rows=99 loops=1)
  Filter: (x < 100)
```

```
EXECUTE stmt(2);
```

QUERY PLAN

```
Seq Scan on test (cost=0.00..20.00 rows=333 width=4) (actual rows=49 loops=1)
  Filter: (int4mul(x, 2) < 100)
```





Prosupport - usage

```
SELECT * FROM a  
WHERE x IN (SELECT ...);
```



```
SELECT * FROM a  
WHERE x IN test_expr(a);
```

test_expr_support(a):
rows = ...
cost = ...
selectivity = ...





- Prosupport for aggregates
- Same technique for Query tree nodes?
- Sublink, Coalesce, BoolExpr - can we register prosupport routines to transform them on-demand?



Query structure





Manage complex Query

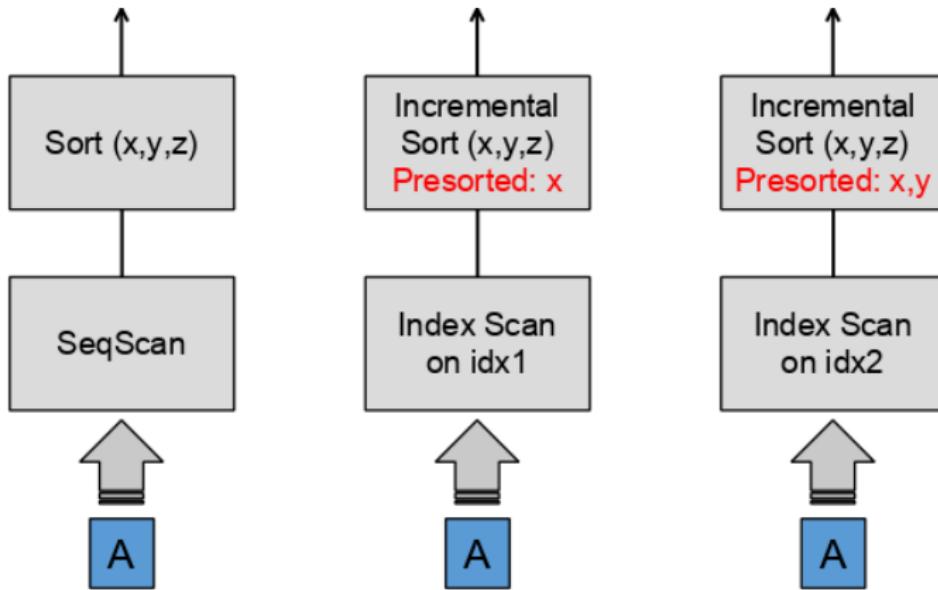
- Multiple joins
- Minimize of sort operations
- Left Joins and inner-generated NULLS
- Multicolumn statistics



Incremental Sort



```
CREATE INDEX idx1 ON test (x);
CREATE INDEX idx2 ON test (x,y);
SELECT x,y,z FROM test ORDER BY x,y,z;
```





Minimize of sort operations

```
SELECT x,y,z FROM test ORDER BY x,y,z;
```

Full sort plan

```
Sort (cost=9845.82..10095.82 rows=100000 width=12)
  Sort Key: x, y, z
    -> Seq Scan on test (cost=0.00..1541.00 rows=100000 width=12)
```

Exploit presorted column x

```
Incremental Sort (cost=7.62..8599.52 rows=100000 width=12)
  Sort Key: x, y, z
  Presorted Key: x
    -> Index Scan using test_x_idx on test (cost=0.29..4007.59 rows=100000 width=12)
```

Exploit presorted columns x and y

```
Incremental Sort (cost=0.86..7118.86 rows=100000 width=12)
  Sort Key: x, y, z
  Presorted Key: x, y
    -> Index Scan using test_x_y_idx on test (cost=0.29..4007.90 rows=100000 width=12)
```



Sort - what's more?

```
EXPLAIN SELECT x,y,z FROM test ORDER BY x,y;
```

QUERY PLAN

Incremental Sort (cost=42.85..9765.17 **rows**=100000 width=12)

Sort **Key**: x, y

Presorted **Key**: x

-> **Index Scan using** idx1 **on** test (cost=0.29..4028.28 **rows**=100000 width=12)

```
EXPLAIN SELECT x,y,z FROM test ORDER BY x,y,z;
```

QUERY PLAN

Incremental Sort (cost=42.85..9765.17 **rows**=100000 width=12)

Sort **Key**: x, y, z

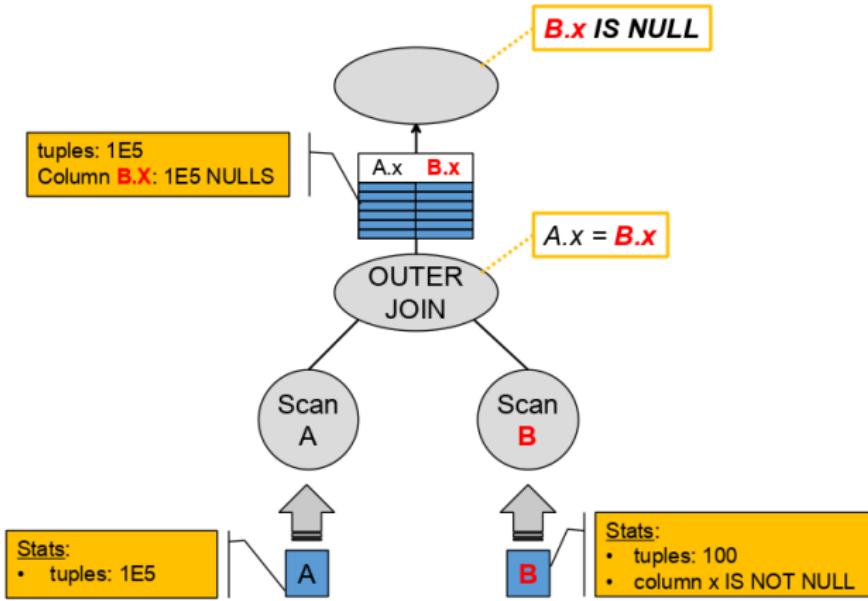
Presorted **Key**: x

-> **Index Scan using** idx1 **on** test (cost=0.29..4028.28 **rows**=100000 width=12)

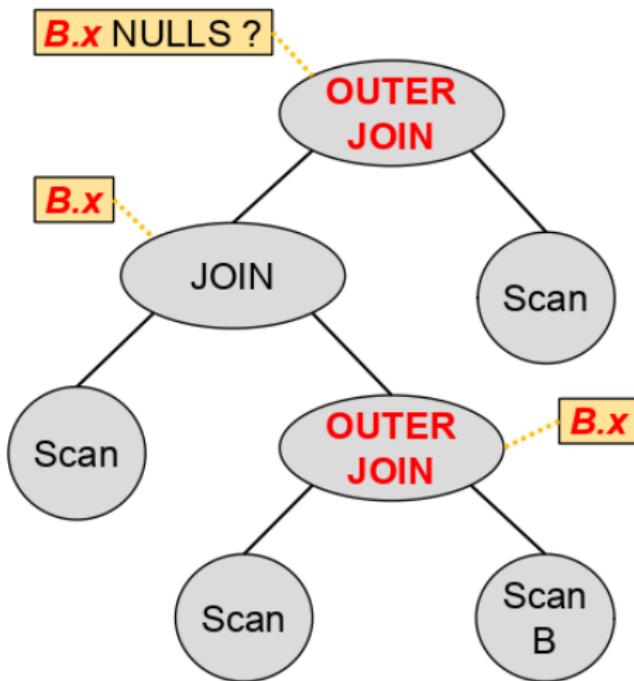




NULLs Counting Problem



NULLs Counting Problem - levels





NULLs Counting Problem - join types

```
EXPLAIN (COSTS OFF)
SELECT oid, relname
FROM pg_class c1
LEFT JOIN pg_class c2
ON c1.relname = c2.relname;
```

Hash **Left Join**

Hash Cond: (c1.relname = c2.relname)
→ Seq Scan **on** pg_class c1
→ Hash
→ Seq Scan **on** pg_class c2

```
EXPLAIN (COSTS OFF)
SELECT c1.oid, c1.relname
FROM pg_class c1
LEFT JOIN pg_class c2 ON true
WHERE c1.relname = c2.relname;
```

Hash **Join**

Hash Cond: (c1.relname = c2.relname)
→ Seq Scan **on** pg_class c1
→ Hash
→ Seq Scan **on** pg_class c2



Extended Statistics



```
SELECT * FROM table WHERE
    id IN (1,2,3,4,5) AND
    enabled = true AND
    status = 'ready';
```

Selectivity **'id IN (1,2,3,4,5)' -> 0.00001**

Selectivity **'enabled = true' -> 0.5**

Selectivity **'status = ready' -> 0.1**

Selectivity of the whole clause - ?





Extended Statistics

CREATE STATISTICS ON id,enabled,status FROM tablename;

- **MCV** - Most Common Values on composite value of (x_1, x_2, x_3) .
- **ndistinct** - number of distinct values on all combinations of columns: $(x_1, x_2), (x_1, x_3), (x_1, x_2, x_3) \dots$
- **dependencies** - functional dependencies between combinations of columns: $x_1 \rightarrow x_2, (x_1, x_2) \rightarrow x_3, \dots$



Vondra. T. CREATE STATISTICS improvements,
PGConf.DE 2022





- **MCV** - two arrays: `values[]` and `frequencies[]`.
- **ndistinct** - 1 integer for each of $2^n - (n + 1)$ combinations
- **dependencies** - 1 float value for each of combinations

columns:	2	3	4	...	8
distinct combinations:	1	4	11	...	247
dependency combinations:	2	9	28	...	1016





Index to define statistics

Table "Parcels":

Indexes:

- "parcel_pkey" PRIMARY KEY, btree (id)
- "parcel_parcel_id" btree (parcel_id)
- "parcel_id_prik" btree (parcel_id, prik)
- "parcel_id_sn_pol" btree (parcel_id, sn_pol)
- "parcel_par_begin" btree (parcel_id, prik_begin_period)
- "parcel_par_patient" btree (parcel_id, patient_id)
- "parcel_par_recid" btree (parcel_id, recid)
- "parcel_per_prik" btree (period, prik)





- Extended statistics for JOINs
- Reduction of combinations inside Distinct and Dependencies based on index definition
- New methods - multidimensional histogram



Miscellaneous optimisations





- VALUES -> ANY transformation
- OR list -> ANY transformation
- GROUP-BY makes column 'unique'
- Static functions evaluation
- ...



Miscellaneous optimisations - VALUES -> ANY



```
SELECT * FROM a  
WHERE a.x IN (VALUES (1), (2), (3));
```

BEFORE transformation

Hash Semi **Join**

Hash Cond:

(a.x = "***VALUES***".column1)

-> Seq Scan **on** a

-> Hash

-> **Values Scan on**
"*VALUES*"

```
SELECT * FROM a  
WHERE a.x = ANY ('{1,2,3}');
```

AFTER transformation

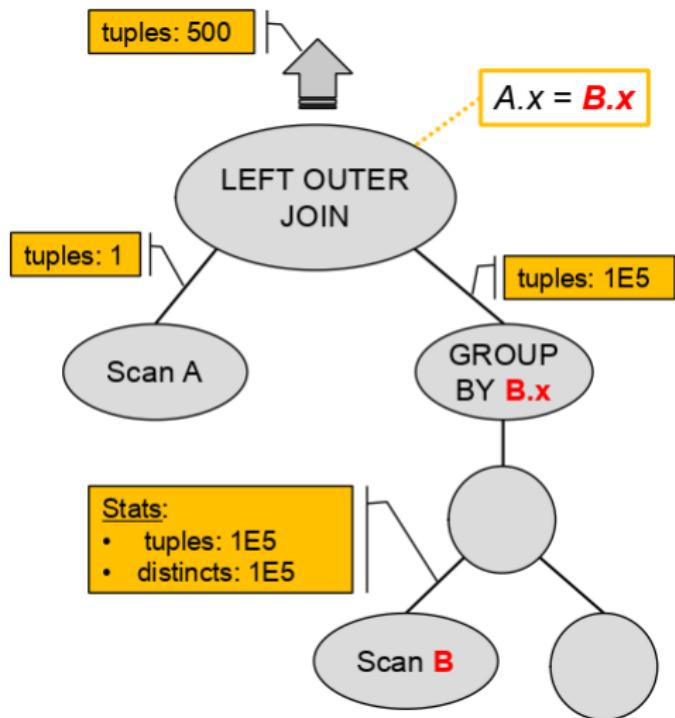
Seq Scan **on** a

Filter:

(x = **ANY ('{1,2,3}'::integer[]))**



Miscellaneous optimisations - GROUP-BY



GROUP-BY makes grouped column 'almost' unique. We can use it in cardinality estimations.





- It may be dozens of such optimisations
- Have quite a narrow application
- May add a lot of overhead in general case
- May hide better query plans
- Complicates the core code





Let extensions do it!

- Query transformation hooks
- Selectivity estimation/statistics hooks
- Optimisation stages hooks
- the add_path hooks
- EXPLAIN node hook
- Extensible parts in the planner structures: PlannerInfo, Query and PlanStatement nodes





Questions ?

