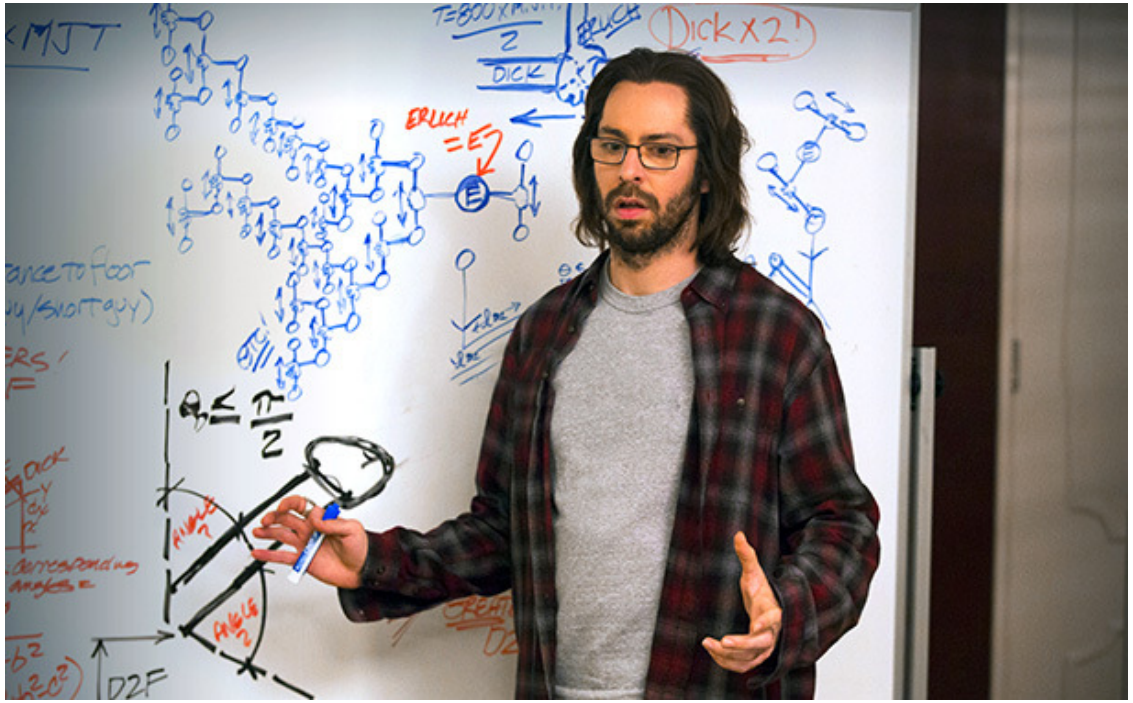
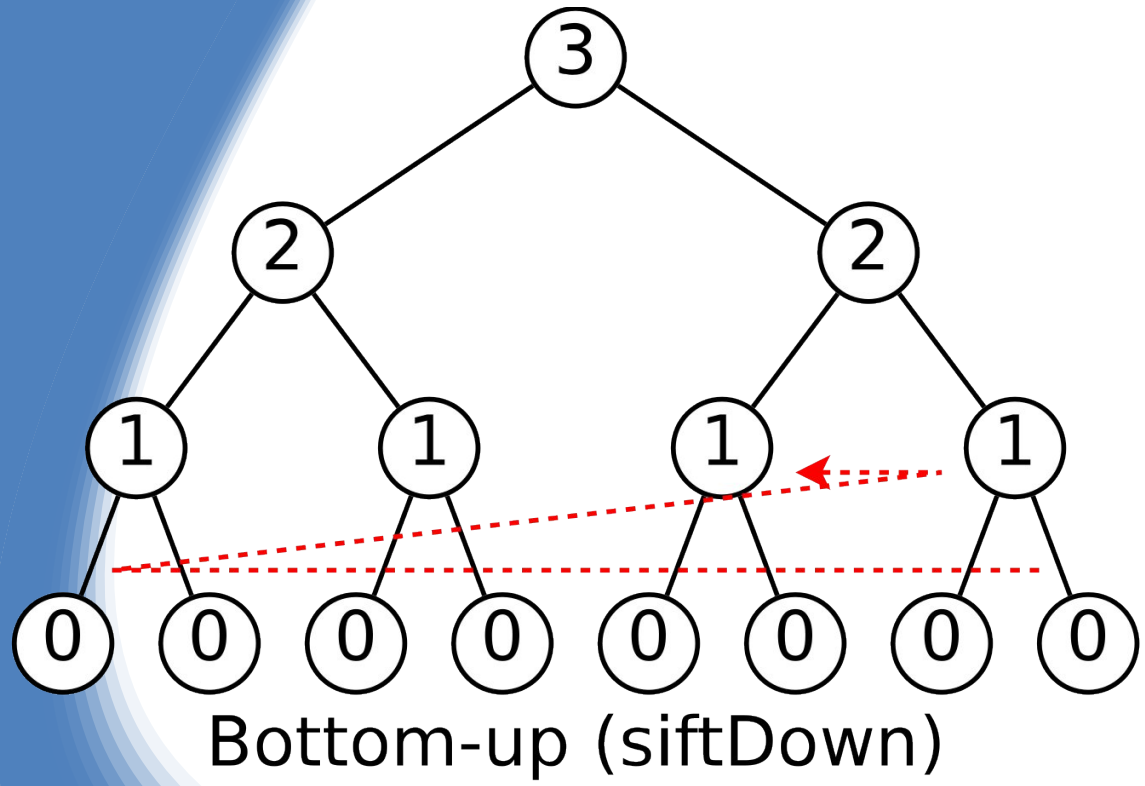
BOTTOM-UP AND DOWN AGAIN:

A Hybrid Planning Approach in action

Andrei Lepikhov



Bottom-Up approach



- Scan -> Join -> Group-by order of planning
- Join tree planning from the leafs to the root
- Optimise Subplan before the upper query
- Optimise CTE before the query

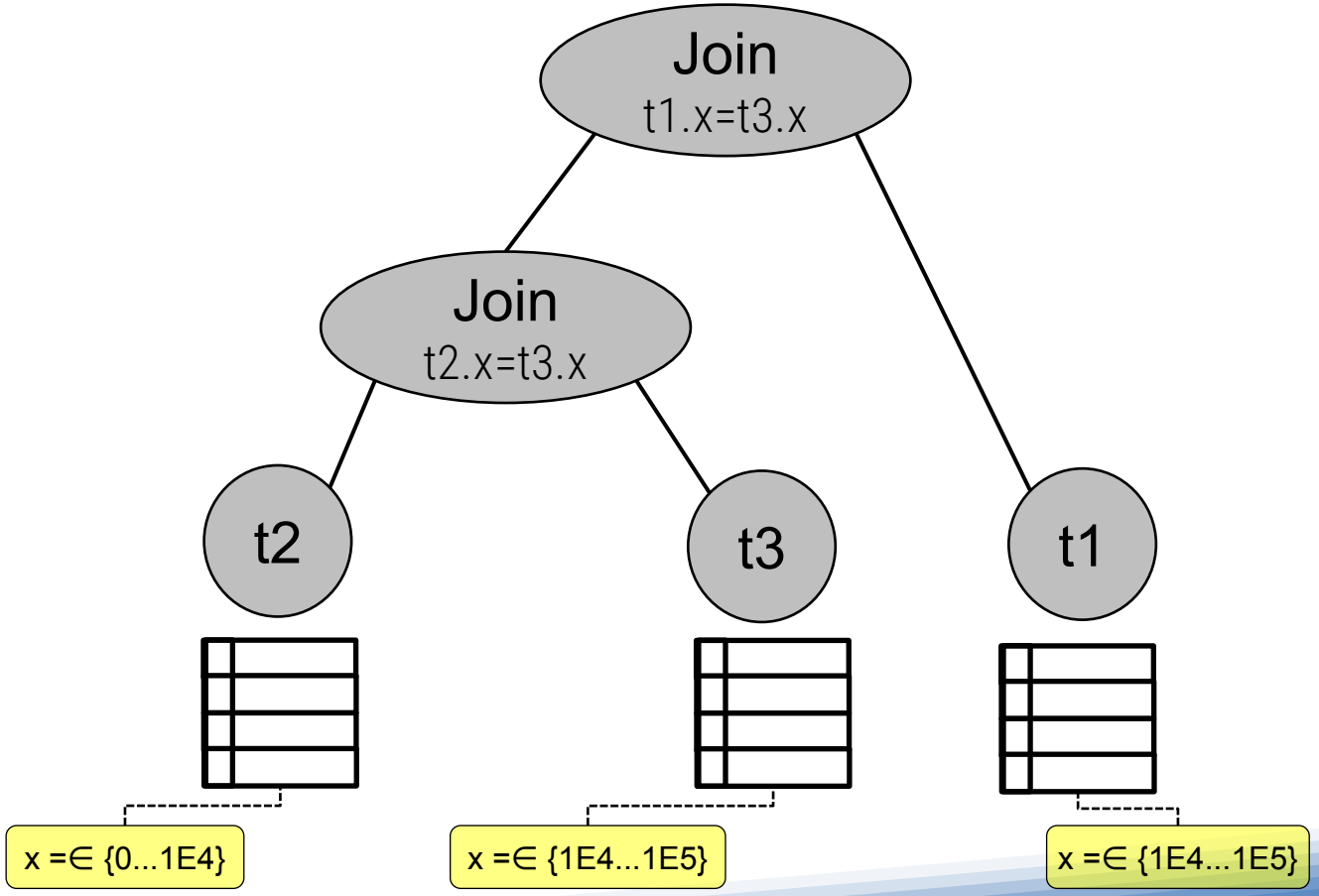
WHAT OPTIONS DO WE LOSE USING THE BOTTOM-UP APPROACH?

Bottom-Up planning weaknesses

- *How much subplan evaluations?*
- *How much subtree re-scans?*
- *Emerging fractional paths and is the LIMIT applicable to my node?*

Emerging fractional path

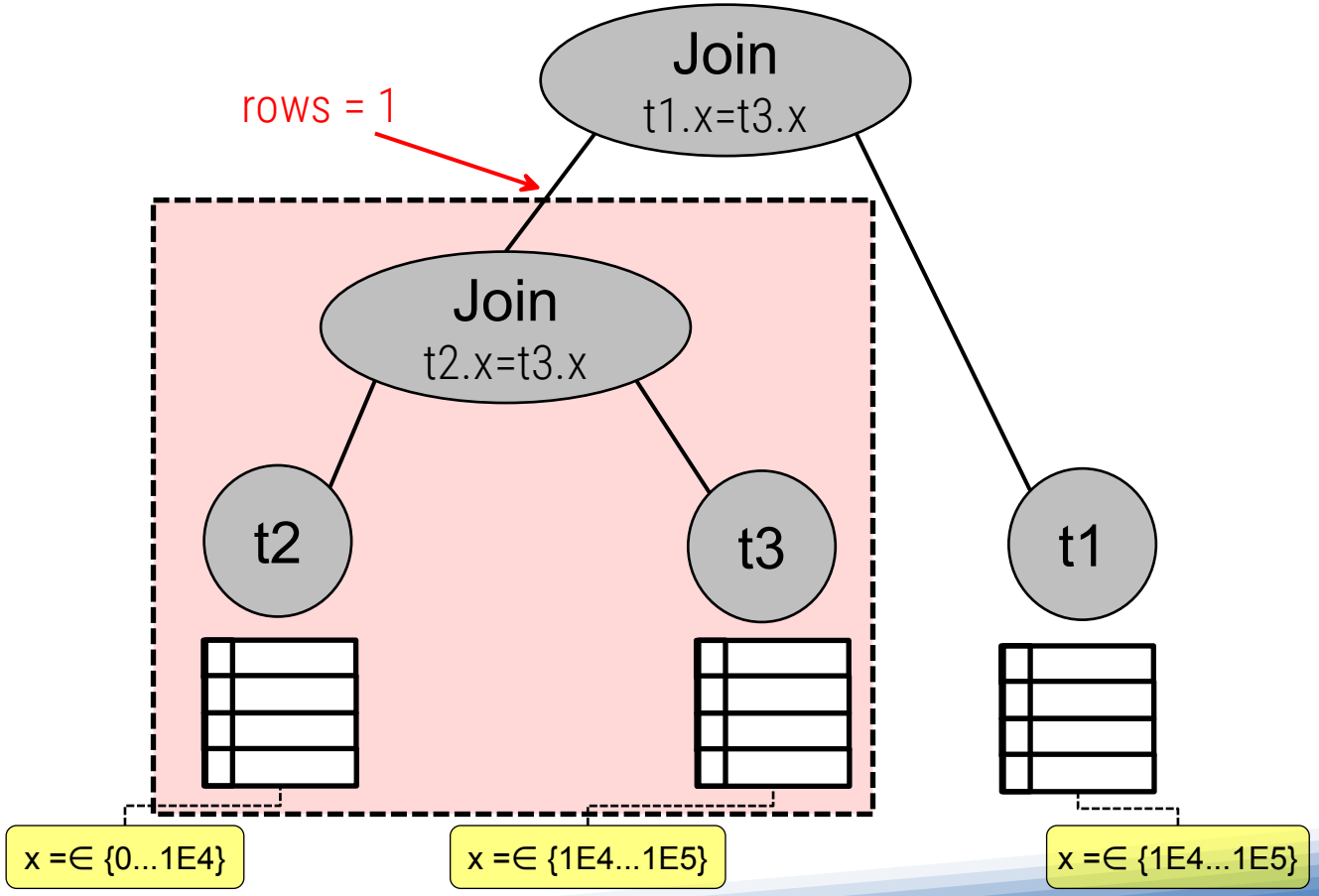
```
SELECT * FROM t1
  JOIN t3
    LEFT JOIN t2
    ON (t2.x=t3.x)
ON (t1.x=t3.x);
```



1. Reproduction: https://github.com/danolivo/conf/blob/main/2025-MiddleOut/example-emerging-fractional-path.sql
2. partition table optimizer join cost misestimation

Emerging fractional path

```
SELECT * FROM t1
  JOIN t3
    LEFT JOIN t2
    ON (t2.x=t3.x)
ON (t1.x=t3.x);
```



1. Reproduction: https://github.com/danolivo/conf/blob/main/2025-MiddleOut/example-emerging-fractional-path.sql
2. partition table optimizer join cost misestimation

Emerging fractional path: EXPLAIN

```
SELECT * FROM t1
  JOIN t3
    LEFT JOIN t2
   ON (t2.x=t3.x)
ON (t1.x=t3.x);
```

```
Hash Join  (rows=1001 width=61) (actual rows=1 loops=1)
  Hash Cond: (t1.x = t3.x)
  ->  Seq Scan on t1  (rows=10001 width=19) (actual rows=10001 loops=1)
  ->  Hash  (rows=1001 width=42) (actual rows=1001 loops=1)
        ->  Merge Left Join  (rows=1001 width=42) (actual rows=1001 loops=1)
              Merge Cond: (t3.x = t2.x)
              ->  Index Scan using t3_x_idx on t3
                    (rows=1001 width=21) (actual rows=1001 loops=1)
              ->  Index Scan using t2_x_idx on t2
                    (rows=90001 width=21) (actual rows=1002 loops=1)
Execution Time: 9.322 ms
```

Emerging fractional path: EXPLAIN

Just add limit 1 ;)

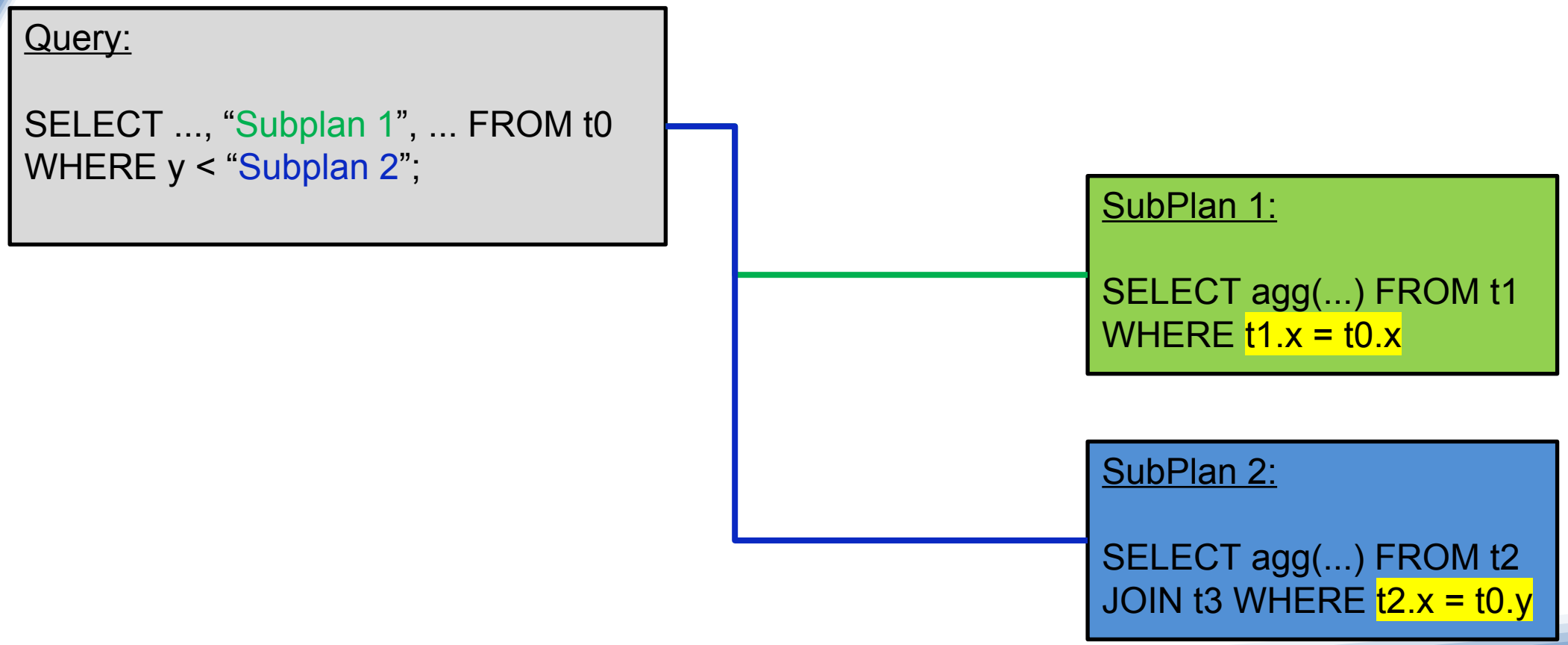
```
SELECT * FROM t1
  JOIN t3
    LEFT JOIN t2
   ON (t2.x=t3.x)
ON (t1.x=t3.x)
LIMIT 1;
```

```
Limit  (rows=1 width=61) (actual rows=1 loops=1)
  ->  Merge Join  (rows=1001 width=61) (actual rows=1 loops=1)
        Merge Cond: (t3.x = t1.x)
        ->  Merge Left Join  (rows=1001 width=42) (actual rows=1 loops=1)
              Merge Cond: (t3.x = t2.x)
              ->  Index Scan using t3_x_idx on t3
                    (rows=1001 width=21) (actual rows=1 loops=1)
              ->  Index Scan using t2_x_idx on t2
                    (rows=90001 width=21) (actual rows=1 loops=1)
        ->  Index Scan using t1_x_idx on t1
              (rows=10001 width=19) (actual rows=1 loops=1)
Execution Time: 0.182 ms
```

```
Execution Time: 9.322 ms
```

Correlated Subplan Caching

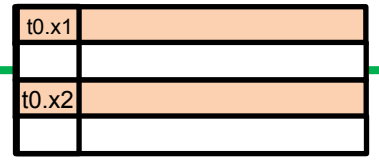
Subplan Caching



Subplan Caching

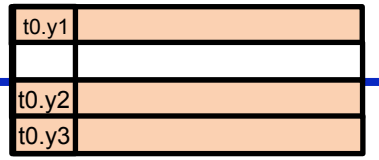
Query:

SELECT ..., “Subplan 1”, ... FROM t0
WHERE y < “Subplan 2”;



SubPlan 1:

SELECT agg(...) FROM t1
WHERE t1.x = t0.x



SubPlan 2:

SELECT agg(...) FROM t2
JOIN t3 WHERE t2.x = t0.y

An example

```
-- Show all employees who are paid less than the average
EXPLAIN (COSTS OFF)
SELECT name FROM employees e1
WHERE salary < (
  SELECT avg(salary)
  FROM employees e2
  WHERE e2.position = e1.position
);
```

Table Employees:

- 10000 records
- 100 positions

An example: Postgres EXPLAIN

```
Seq Scan on employees e1
                    (actual time=4.359..9350.156 rows=4991 loops=1)
  Filter: (salary < (SubPlan 1))
  SubPlan 1
    ->  Aggregate (actual time=0.934..0.934 rows=1 loops=10000)
          ->  Seq Scan on employees e2
              (actual time=0.007..0.925 rows=100 loops=10000)
               Filter: ("position" = e1."position")
Planning Time: 0.147 ms
Execution Time: 9350.361 ms
```

Table Employees:

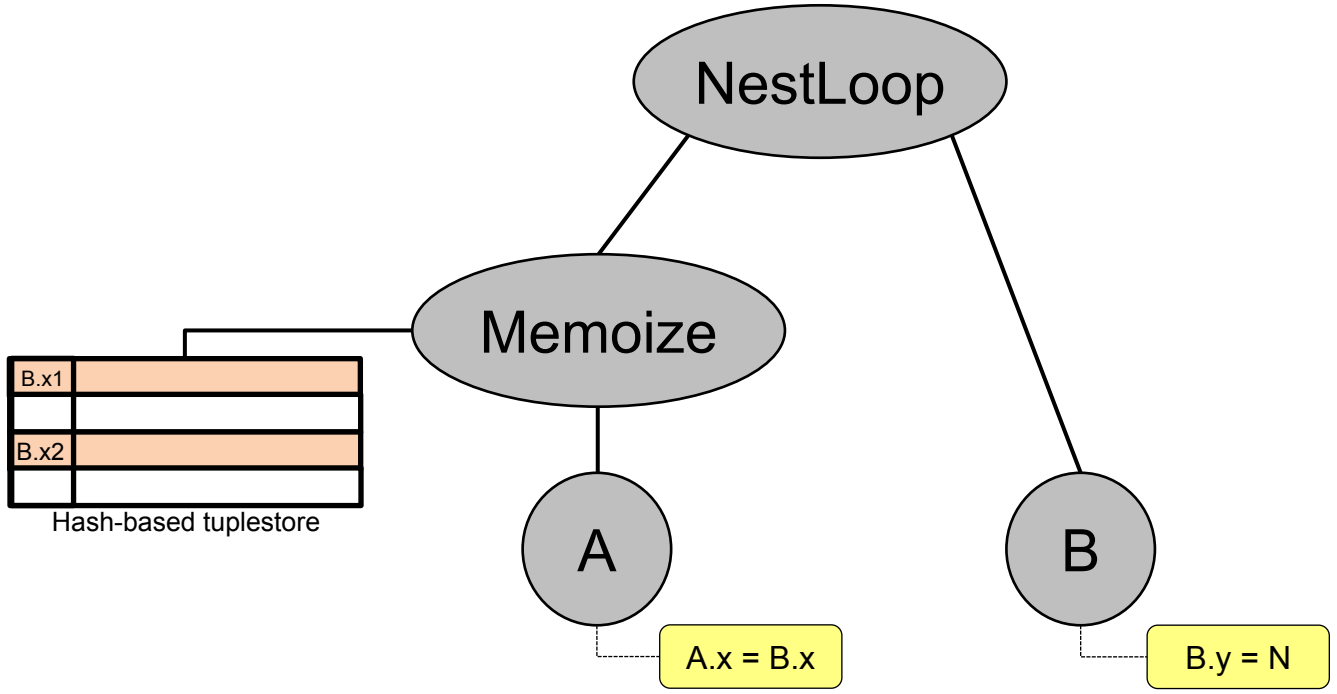
- 10000 records
- 100 positions

The Purpose

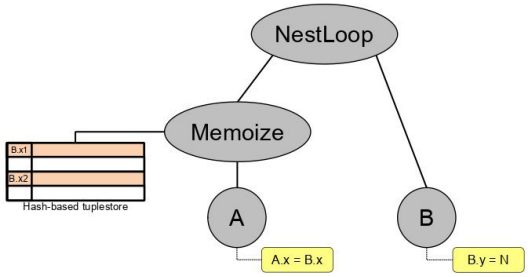
Correlated Subplan evaluation may be expensive. If parameter set, handed over to subplan isn’t changed, evaluation result will be the same. In this work we want to reduce number of Subplan calls by caching imcoming parameters and corresponding result.

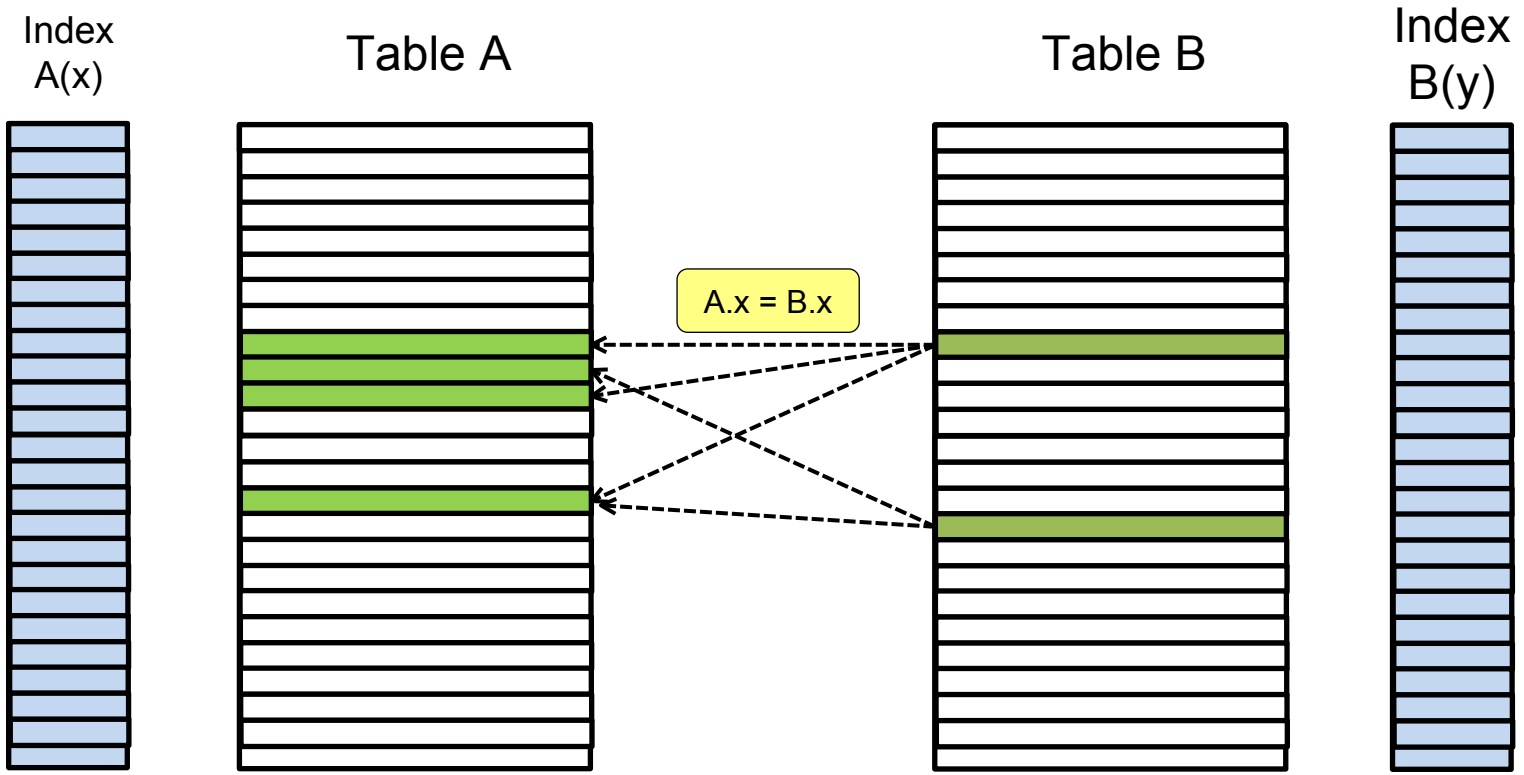
Do we have something related in the Postgres core?

NestLoop + Memoize



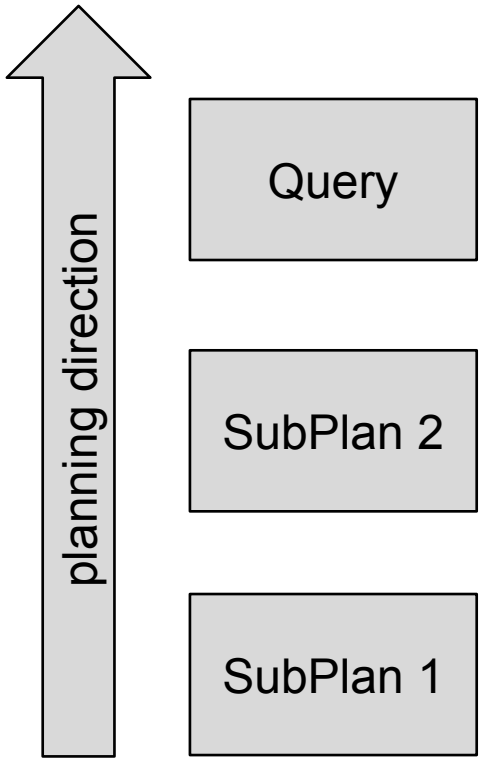
Use Case





Parameterised NestLoop is the most effective when it has highly selective clauses on both sides employing indexes and B.x includes lots of duplicates

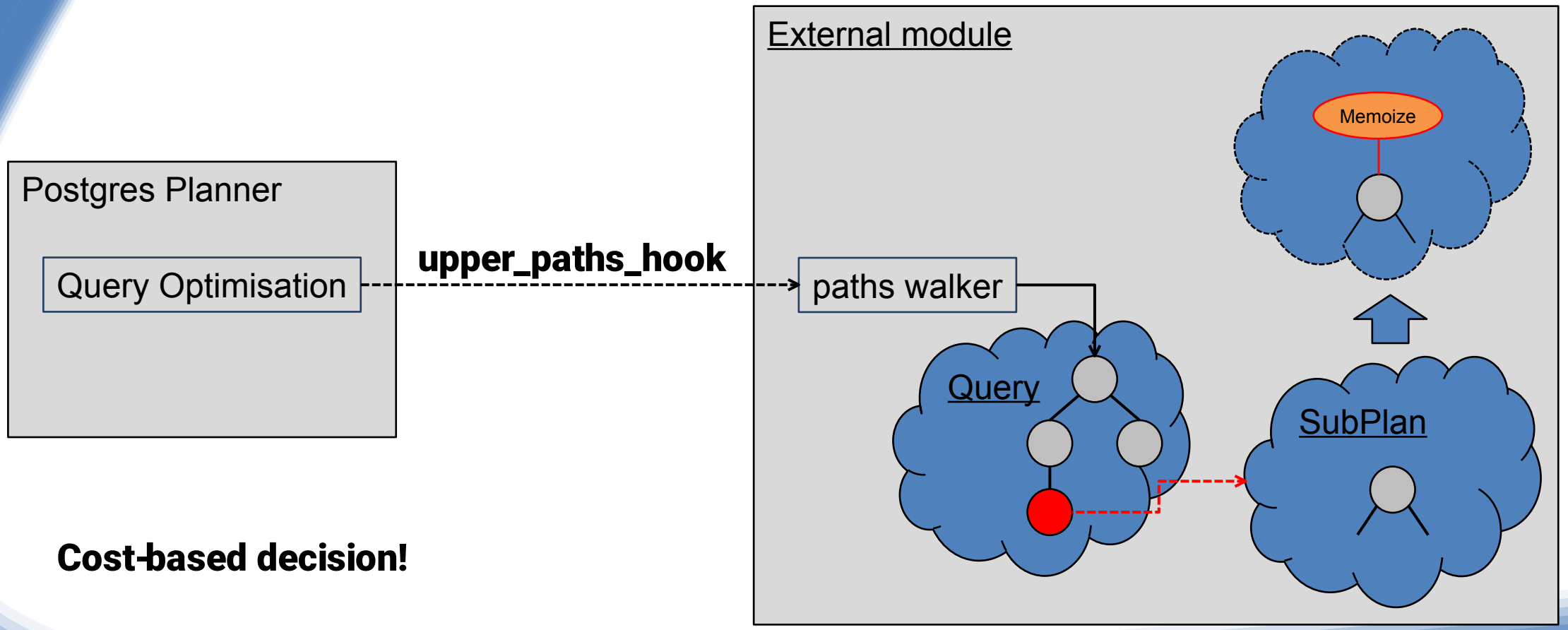
Why Postgres doesn’t cache subplans?



- At the Subplan planning stage Postgres have only parse tree of the upper Query
- Info on parameters that comes from the upper query is not available
- No statistics on columns of the upper query

The top-down approach to identifying alternative execution paths for the subplan, after planning the main query, provides sufficient information for the subquery caching mechanism.

How does it work?



Cost-based decision!

The example: demo

```
Seq Scan on employees e1 (actual time=6.257..127.318 rows=5011.00 loops=1)
  Filter: (salary < (SubPlan 1))
  SubPlan 1
    ->  Memoize (actual time=0.012..0.012 rows=1.00 loops=10000)
          Cache Key: e1."position"
          Cache Mode: binary
          Hits: 9900  Misses: 100
          ->  Aggregate (actual time=1.209..1.209 rows=1.00 loops=100)
                ->  Seq Scan on employees e2 (actual time=0.008..1.196 rows=100.00 loops=100)
                      Filter: ("position" = e1."position")
Planning Time: 0.460 ms
Execution Time: 127.555 ms
```

Table Employees:

- 10000 records
- 100 positions

```
                (actual time=0.007..0.925 rows=100 loops=10000)
                 Filter: ("position" = e1."position")
Planning Time: 0.147 ms
Execution Time: 9350.361 ms
```

Limitations :(

- No min / max aggregates yet
- No GROUPING SETS
- Cache only keys referencing immediate upper query
- Path walker is under construction – not all the places in the query may be visited in search of subplans

That’s it!
Questions?

Any critics welcome