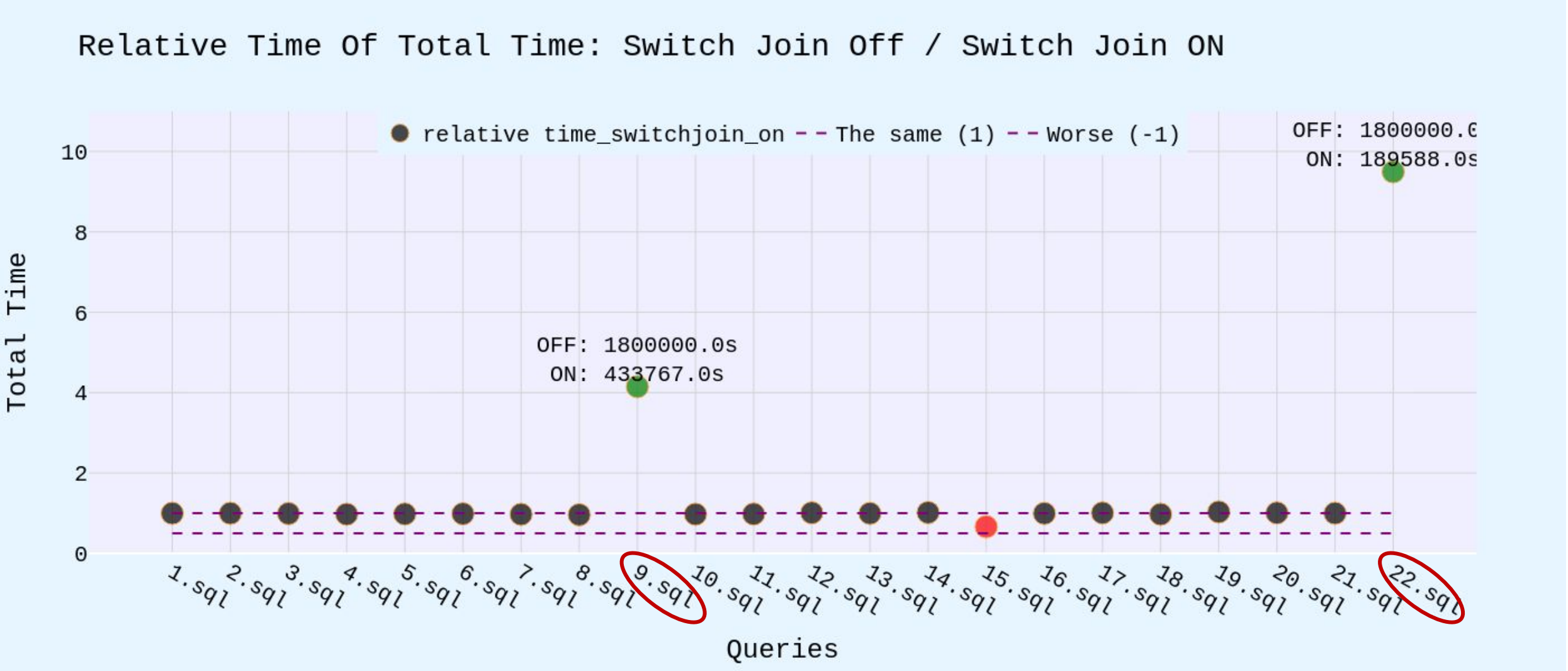
PGConf.dev 2025

PostgresPro

Switching Between Query Plans in Real Time (Switch Join)

Author: Alena Rybakina

Switch Join Can Give Significant Improvement



About Me

- Core developer in Postgres Professional since 2021
- Contributing to the PostgreSQL project since 2023
  - OR to ANY transformation
  - Values to ANY transformation
  - Others
- Participated in extension development:
  - AQO
  - Replaning

OUTLINE OF THE REPORT

Outline of The Report

1. The Reasons for Switch Join
2. Switch Join
3. Extension Implementation
4. Benchmarks
5. Conclusion and development

1. The Reasons for Switch Join

Why Are My Queries So Slow

Planner and Optimizer Imperfections

- Cardinality under- or over-estimations
- Cost estimation mistakes
- Use of SubPlans instead of joins
- Inefficient Sort/Group strategy

Common Bottlenecks

- Redundant algorithm cycles
- High data volume without filtering

Other planner and executor issues…

The Main Problem

The wrong cardinality estimation leads to the choice a suboptimal algorithm

Query Planning Gone Wrong
Robert Haas

Adaptive Cardilnality Estimation
O.Ivanov, S.Bartunov,
Arxiv, Nov.2017

How good are query optimizers, really?
V.Levis, A.Gubichev, A.Mirchev,
P.Boncz,
A.Kemper and T.Neumann

A generalized join algorithm
Goetz Graefe
Hewlett-Packard Laboratories

ABOUT INITIAL ISSUES

ABOUT SWITCH JOIN

Some Optimizations To Simplify The Problem

- What Can We Optimize
  - Group By
  - Skip Scan
  - Self-Join
  - Pull-Up
  - Others
- What Can We use
  - Extended Stats

Some More Solutions

These solutions store the actual cardinality and allow the optimizer to use it during query plan generation

Postgres query re-optimisation
Lepikhov Andrei, Belyalov Damir, Rybakina Alena
Postgres Professional Nov. 2023

Adaptive Query Optimization
Postgres Professional

A generalized join algorithm
Goetz Graefe
Hewlett-Packard Laboratories

Adaptive Join
SQL Server 2017

Some More Solutions

They combine multiple join strategies and dynamically choose the most efficient one based on runtime information.

Postgres query re-optimisation
Lepikhov Andrei, Belyalov Damir, Rybakina Alena
Postgres Professional Nov. 2023

Adaptive Query Optimization
Postgres Professional

A generalized join algorithm
Goetz Graefe
Hewlett-Packard Laboratories

Adaptive Join
SQL Server 2017

A Problem Scenario

Table names	Table "public.a"	Table "public.b"
Schema		
Initial Data Population		
Updates statistics		
Additional Data		

ABOUT INITIAL ISSUES

ABOUT SWITCH JOIN

A Problem Scenario

Table names	Table "public.a"	Table "public.b"
Schema	x: numeric y: numeric Indexes: "a_y_idx" btree (y)	x: numeric y: numeric
Initial Data Population		
Updates statistics		
Additional Data		

A Problem Scenario

Table names	Table "public.a"	Table "public.b"
Schema	x: numeric y: numeric Indexes: "a_y_idx" btree (y)	x: numeric y: numeric
Initial Data Population	Inserts 100 rows x: values from 1 to 100 y: repeats values from 0 to 4	Also 100 rows x: values from 1 to 100 y: repeats values from 0 to 7
Updates statistics		
Additional Data		

A Problem Scenario

Table names	Table "public.a"	Table "public.b"
Schema	x: numeric y: numeric Indexes: "a_y_idx" btree (y)	x: numeric y: numeric
Initial Data Population	Inserts 100 rows x: values from 1 to 100 y: repeats values from 0 to 4	Also 100 rows x: values from 1 to 100 y: repeats values from 0 to 7
Updates statistics	VACUUM ANALYZE a,b;	
Additional Data		

A Problem Scenario

Table names	Table "public.a"	Table "public.b"
Schema	x: numeric y: numeric Indexes: "a_y_idx" btree (y)	x: numeric y: numeric
Initial Data Population	Inserts 100 rows x: values from 1 to 100 y: repeats values from 0 to 4	Also 100 rows x: values from 1 to 100 y: repeats values from 0 to 7
Updates statistics	VACUUM ANALYZE a,b;	
Additional Data	Inserts 10,000 rows x: repeats values from 1 to 7 y: 6 (fixed)	Also 10,000 rows x: 6 (fixed) y: values from 1 to 10,000

ABOUT INITIAL ISSUES

ABOUT SWITCH JOIN

An Initial Problem: The Query Plan Without Switch Join

This query: `EXPLAIN SELECT count(*) FROM a join b on a.x=b.x and a.y=6;`

- Joins on `a.x = b.x`
- Filters `a.y = 6` (a value heavily inserted but not evenly distributed)

```
->  Nested Loop  (rows=1) (actual rows=12508750.00)
      ->  Index Scan using a_y_idx on a  (rows=1) (actual rows=10000.00)
            Index Cond: (y = '6'::numeric)
            Index Searches: 1
      ->  Index Only Scan using b_x_idx on b  (rows=1) (actual rows=1250.88)
            Index Cond: (x = a.x)
            Heap Fetches: 12508750
            Index Searches: 10000
```

An Initial Problem: The Query Plan With Switch Join

```
EXPLAIN SELECT count(*) FROM a join b on a.x=b.x and a.y=6;
```

	Without Switch Join	With Switch Join
Time	Planning Time: 0.231 ms Execution Time: 6814.181 ms	

An Initial Problem: The Query Plan With Switch Join

```
->  Custom Scan (SwitchJoin)  (rows=1 width=0) (actual rows=12508750.00)
      --> Limit cardinality: 100
      ->  Nested Loop  (rows=1) (never executed)
            ->  Materialize  (rows=1) (actual rows=5050.50)
                  ->  Index Scan using a_y_idx on a  (rows=1) (actual rows=5050.50)
                        Index Cond: (y = '6'::numeric)  Index Searches: 2
            ->  Index Only Scan using b_x_idx on b  (rows=1) (never executed)
                  Index Cond: (x = a.x) Index Searches: 0
      ->  Hash Join  (rows=1) (actual rows=12508750.00)
            Hash Cond: (b.x = a.x)
            ->  Seq Scan on b  (rows=4500) (actual rows=10100.00)
            ->  Hash  (rows=1) (actual rows=10000.00)
                  ->  Materialize  (rows=1) (actual rows=5050.50)
                        ->  Index Scan using a_y_idx on a  (rows=1) (actual rows=5050.50)
                              Index Cond: (y = '6'::numeric)  Index Searches: 2
```

An Initial Problem: The Query Plan With Switch Join

```
EXPLAIN SELECT count(*) FROM a join b on a.x=b.x and a.y=6;
```

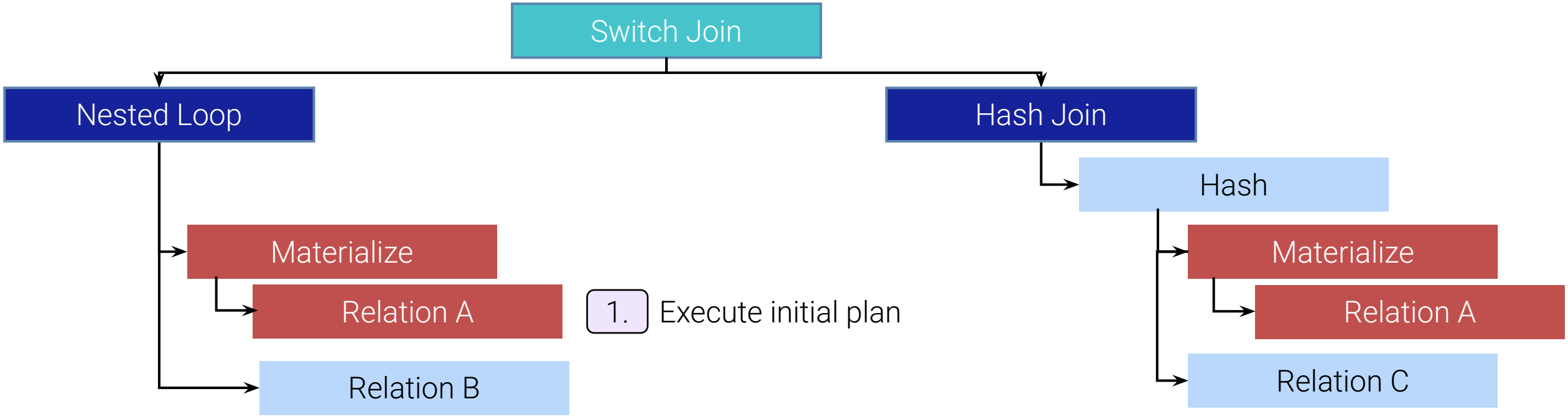
	Without Switch Join	With Switch Join
Time	Planning Time: 0.231 ms Execution Time: 6814.181 ms	Planning Time: 0.259 ms Execution Time: 1240.181 ms

2. Switch Join

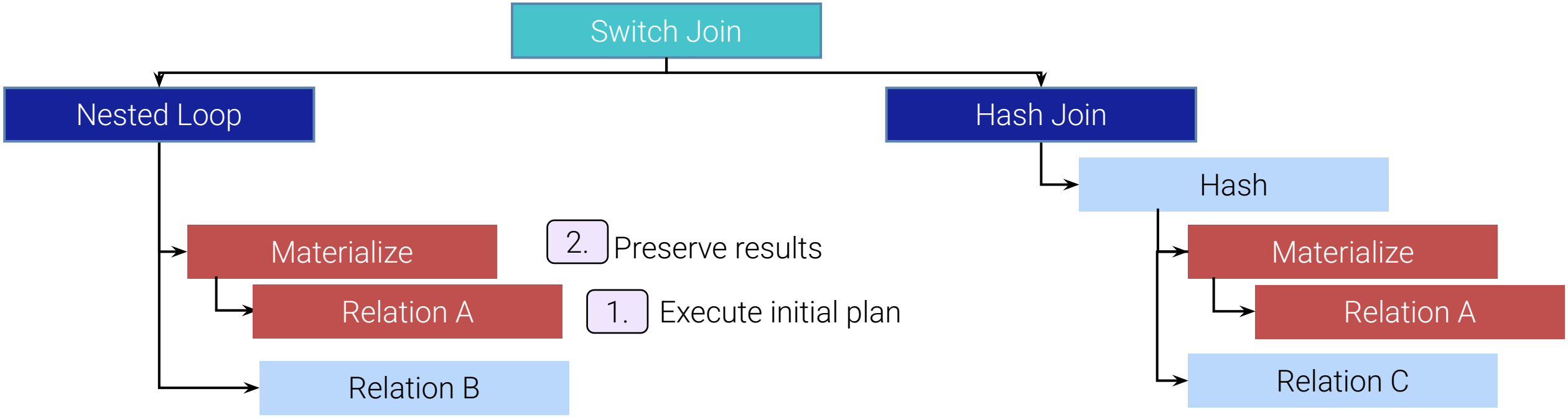
Switch Join

- It is an extension of PostgreSQL
- It saves two plans that are generated by the optimizer:
  - As-Is – assumes the optimizer's prediction is correct
  - Pessimistic – assumes the optimizer underestimates cardinality
- The query engine chooses between these plans only after observation of the actual cardinality at the switching point during runtime.

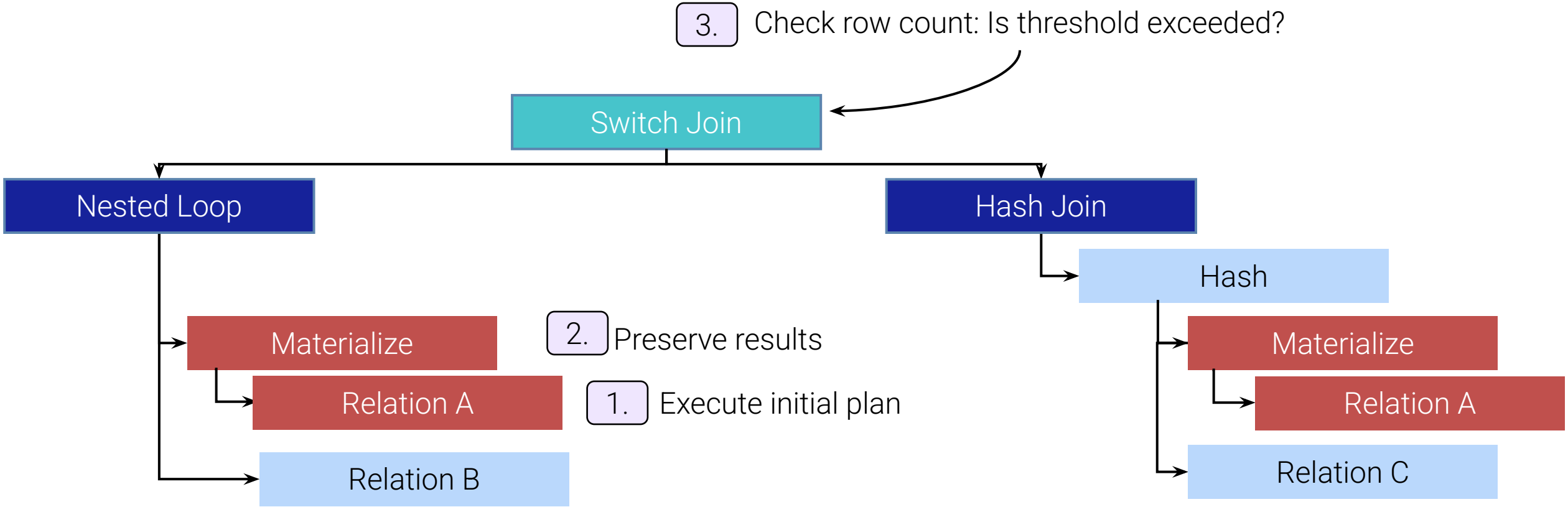
The Switching Process



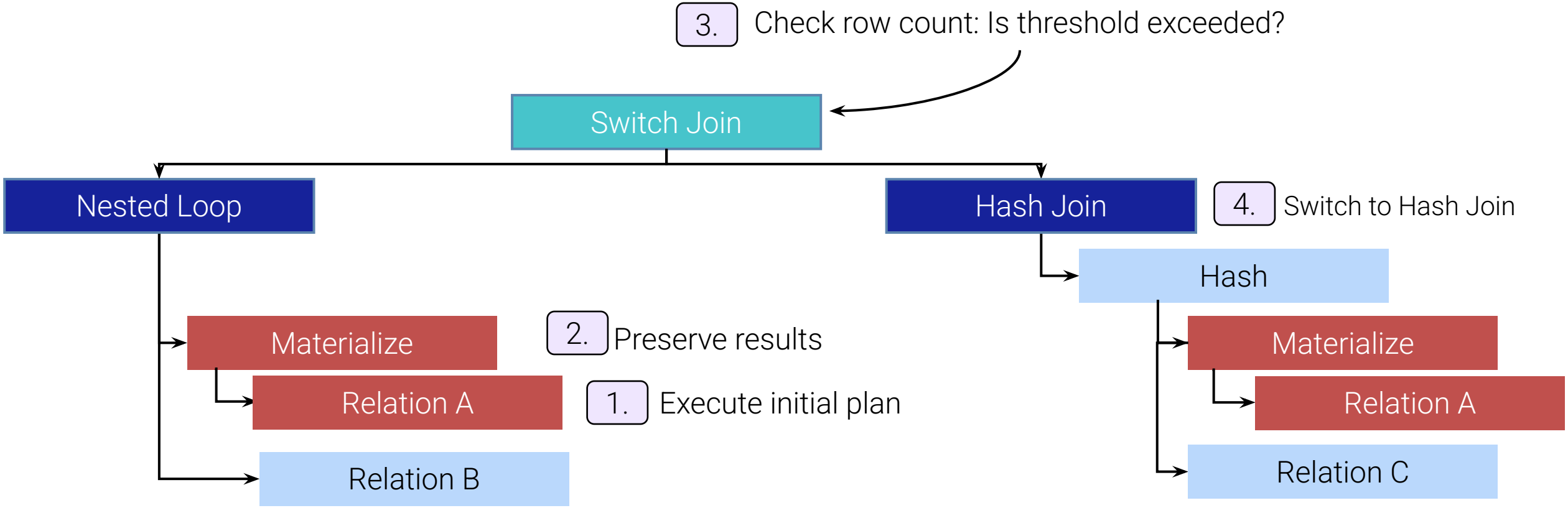
The Switching Process



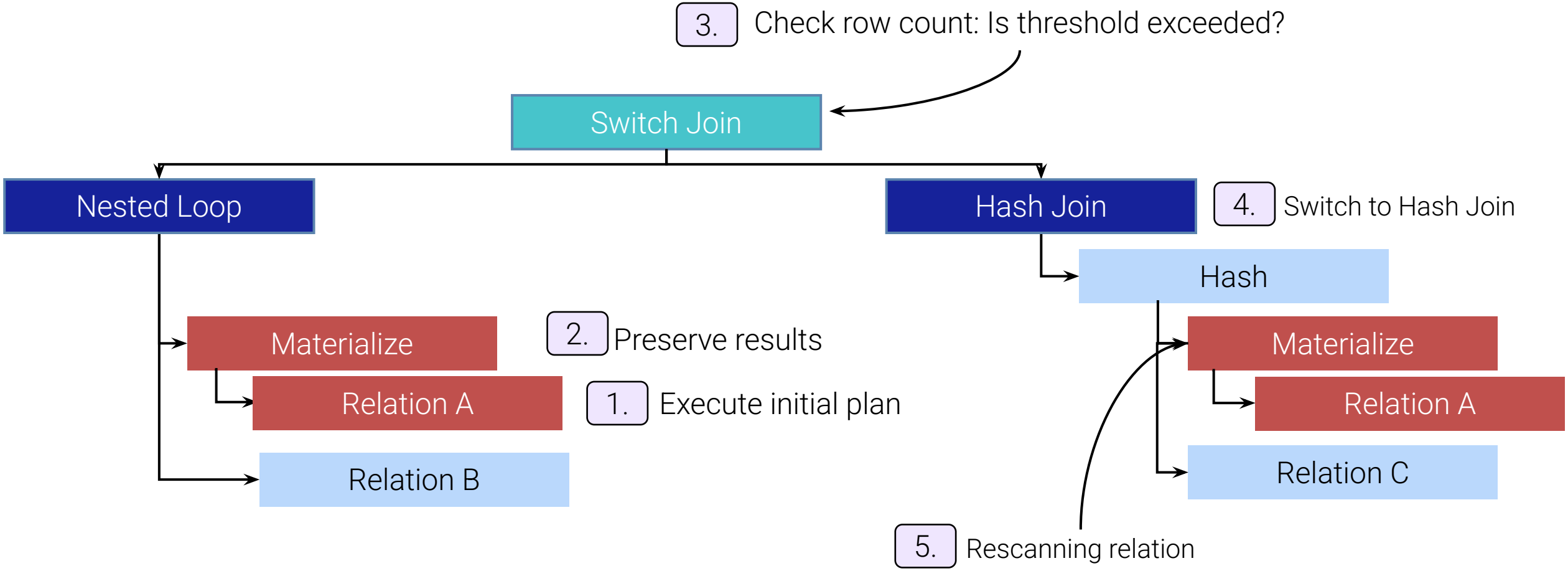
The Switching Process



The Switching Process



The Switching Process



Switch Join Simplified

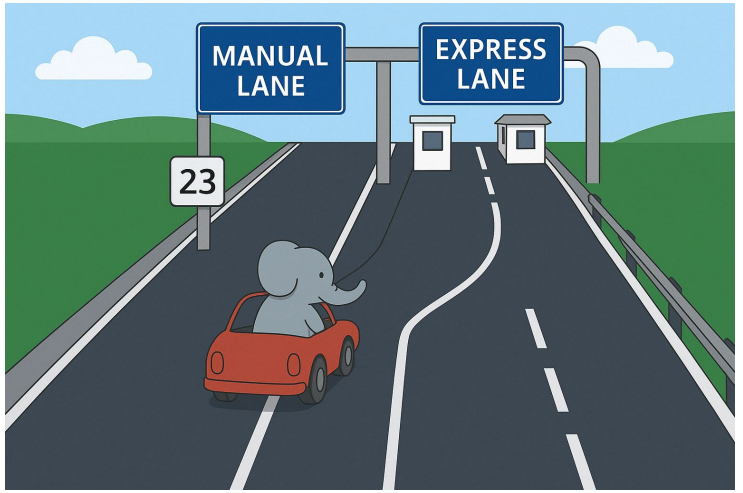
Imagine a mini elephant is driving on a highway approaching a toll plaza.

There are two types of lanes:

- ✅ Manual Lane — free and fast, but only when the traffic is light.
- ⚡ Express Lane — built for the heavy traffic, but comes at a cost.

The elephant doesn't have to choose the lane before he sees the traffic.

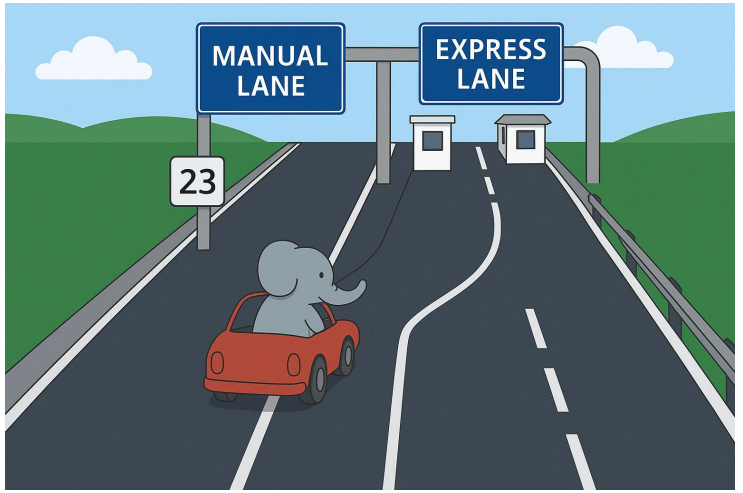
A sensor counts how many cars are ahead of the elephant.



Switch Join Simplified

The elephant is Switch Join.
The traffic sensor checks how many cars are ahead is the observation process how many tuples in the outer node of Nested Loop.
The mini elephant decides whether path should be more optimal:

- ✅ Manual Lane — Uses a Nested Loop; efficient for the light traffic.
- ⚡ Express Lane — Uses a Hash Join; adds overhead but excels under heavy traffic.



Switch Join Implementation with NestLoop, HashJoin, and MergeJoin

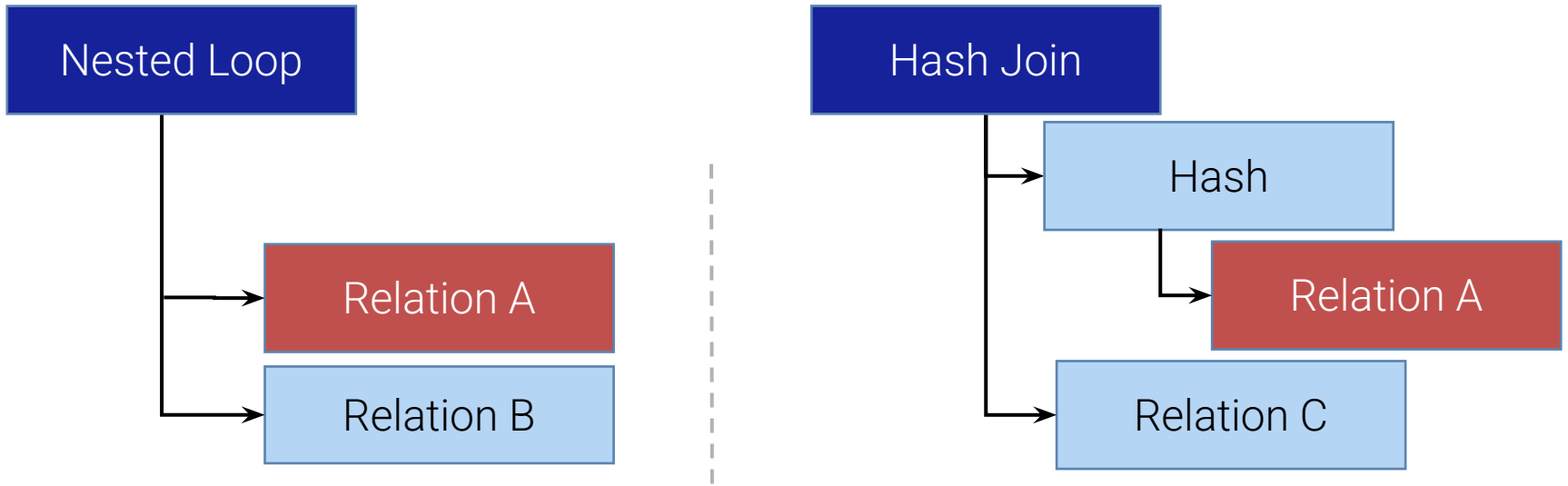
Behind The Scenes: Switch Join Mechanics

- Prepare for Nested Loop available the Hash or Merge Join node
- Prepare shared outer input to avoid recomputation
- Define a row count threshold to decide when to switch
- Estimate the costs for both plans
- Define the switching algorithm:
  - Start executing the probe input (the outer side)
  - If the row count exceeds a predetermined threshold, switches to Hash Join
  - The row count threshold is based on the internal cost-based heuristics

Preparing The Appropriate Path

Preparing The Appropriate Path

The same relation used as the outer input in the Nested Loop is also used in the Hash Join.
Relation B and Relation C may differ, since Relation C is always recomputed.



Preparing The Appropriate Path

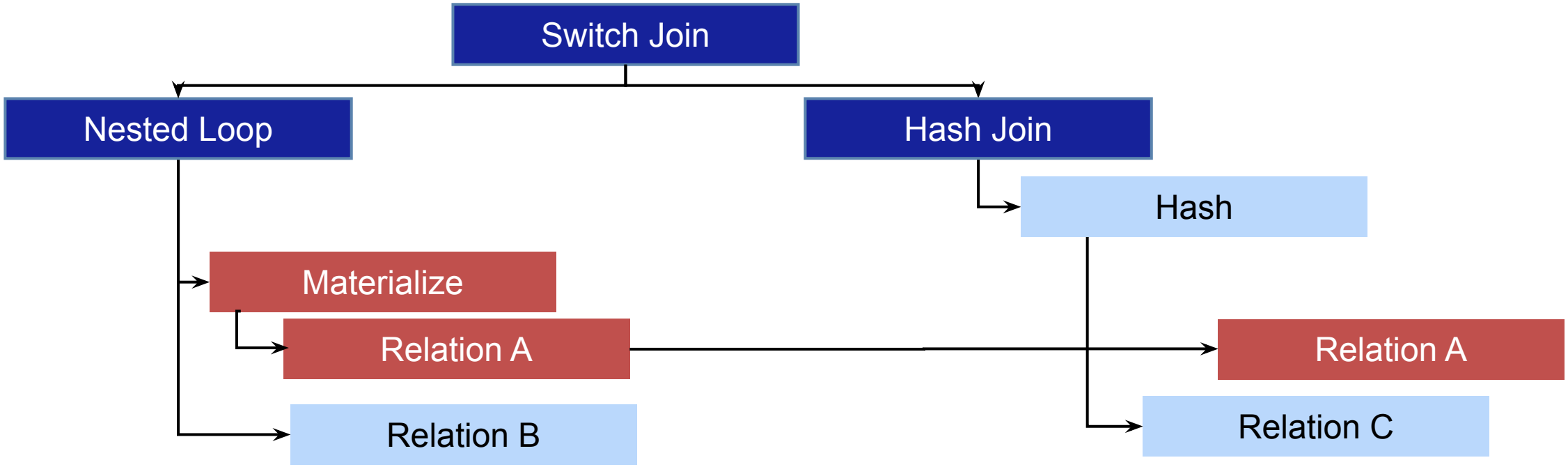
The same path as the outer of the Nested Loop

Why is keeping the same outer path crucial?

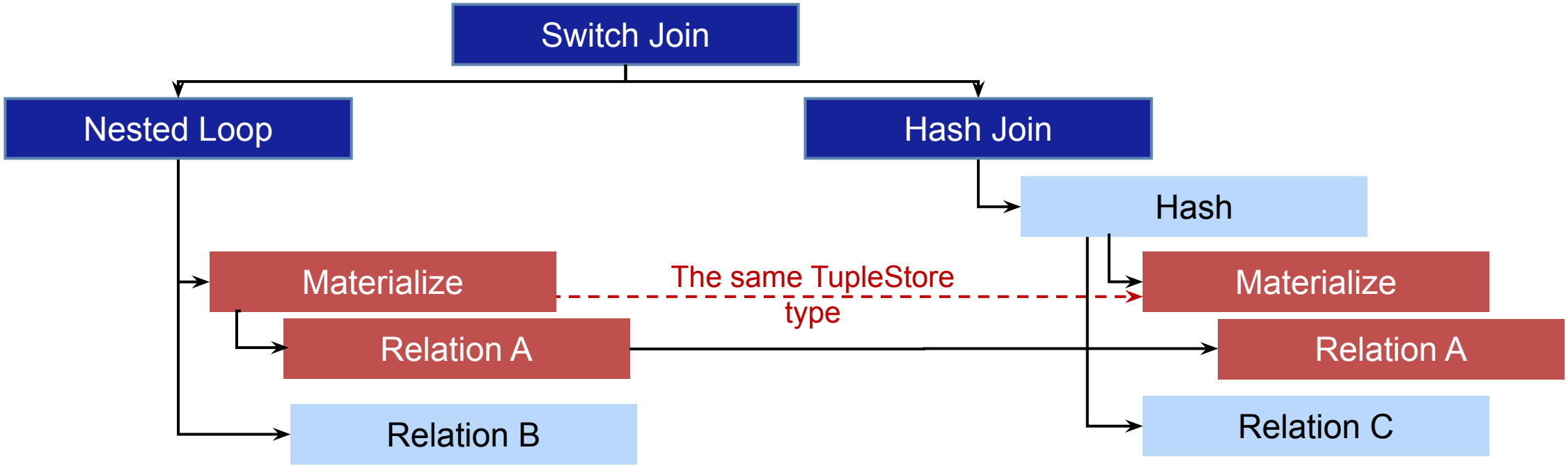
- Avoid Re-Execution or Duplication
- Preserve Execution Semantics

Sharing Outer Node’s Results

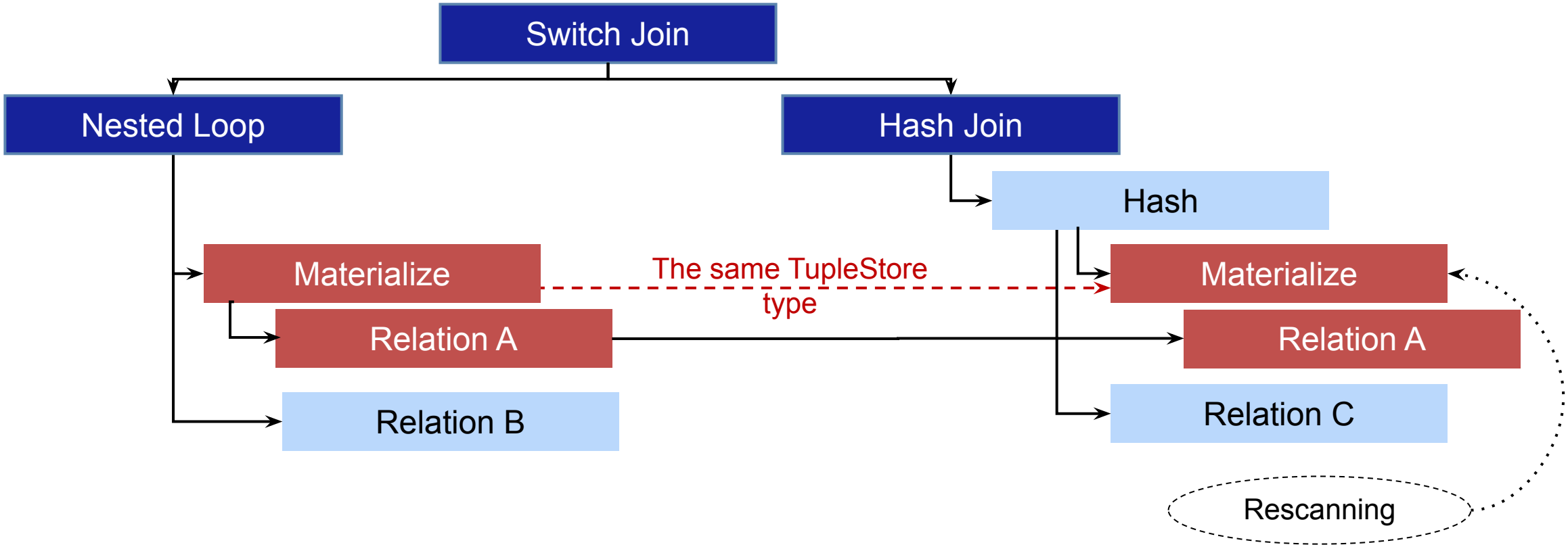
Sharing Outer Node Via Materialization



Sharing Outer Node Via Materialization



Sharing Outer Node Via Materialization



Sharing Outer Node Via Materialization

⚠️ The following node types should NOT be used as outer input for switching:

- Volatile Functions (e.g., random(), now())
- Set-Returning Functions (e.g. generate_series())
- CTEs with Side Effects (e.g., using nextval() or UPDATE)
- Cursor or Portal Usage

❗ Why it can’t be reused - it is not safe

the result might change between outer rows

Nested Loop Issues

Nested Loop With Pathkeys

✓ Avoids Additional Sorts

Indexes return data in a specific order which is preserved in the pathkeys list.

The planner can reuse this ordering in later stages and skip the sort step (for example, before a Merge Join).

X Slower for large inputs

Even if the data is ordered, Nested Loop joins may not perform well with large volumes of data.

SELECT b.x FROM a join b on a.x=b.x group by (b.x);

```
Group
  Group Key: b.x
  ->  Nested Loop
        ->  Index Only Scan using a_idx on a
        ->  Index Only Scan using b_idx on b
              Index Cond: (x = a.x)
```

Nested Loop With Pathkeys

✓ Avoids Additional Sorts

Indexes return data in a specific order which is preserved in the pathkeys list.

The planner can reuse this ordering in later stages and skip the sort step (for example, before a Merge Join).

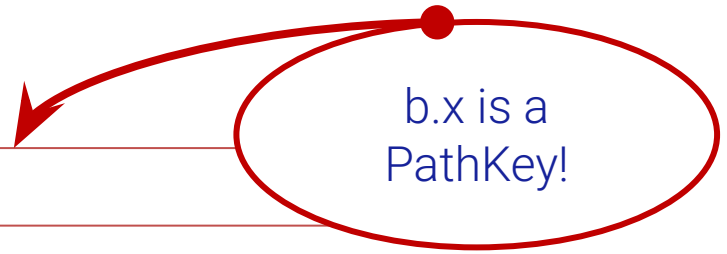
X Slower for large inputs

Even if the data is ordered, Nested Loop joins may not perform well with large volumes of data.

SELECT b.x FROM a join b on a.x=b.x group by (b.x);

```
Group
  Group Key: b.x
  ->  Nested Loop
        ->  Index Only Scan using a_idx on a
        ->  Index Only Scan using b_idx on b
              Index Cond: (x = a.x)
```

```
Group
  Group Key: b.x
  ->  Merge Join
        Merge Cond: (b.x = a.x)
        ->  Materialize
              ->  Index Only Scan using b_idx on b
        ->  Index Only Scan using a_idx on a
```



Nested Loop With Pathkeys

✓ Avoids Additional Sorts

Indexes return data in a specific order which is preserved in the pathkeys list.

The planner can reuse this ordering in later stages and skip the sort step (for example, before a Merge Join).

X Slower for large inputs

Even if the data is ordered, Nested Loop joins may not perform well with large volumes of data.

SELECT b.x FROM a join b on a.x=b.x group by (b.x);

```
Group
  Group Key: b.x
  ->  Nested Loop
        ->  Index Only Scan using a_idx on a
        ->  Index Only Scan using b_idx on b
              Index Cond: (x = a.x)
```

```
Group
  Group Key: b.x
->  Sort
      Sort Key: b.x
      ->  Hash Join
            Hash Cond: (b.x = a.x)
            ->  Index Only Scan using b_idx on b
          ->  Hash
            ->  Index Only Scan using a_idx on a
```

b.x is a PathKey!

How Nested Loop Should Be Replaced By Switch Join

NL type	Features	Switch Join construction
with pathkeys	√ Avoids Additional Sorts X Slower for large volume of data	NestedLoop + {HashJoin+Sort} NestedLoop + MergeJoin
parameterized		
paralleled		
tedious		

How Nested Loop Should Be Replaced By Switch Join

NL type	Features	Switch Join construction
with pathkeys	√ Avoids Additional Sorts X Slower for large volume of data	NestedLoop + {HashJoin+Sort} NestedLoop + MergeJoin
parameterized		NestedLoop + HashJoin
paralleled		
tedious		

Parameterized Nested Loop

✓ Efficient Index Usage

The inner side is often an Index Scan, filtered by a parameter from the outer row.

⚠️ Complexity In Planner Prediction

Since the inner side is re-evaluated for every outer row, the selectivity of parameterized lookups can vary significantly depending on data distribution.

X Slower for Large Data Volumes

It can result in many repeated index lookups, which may degrade performance.

SELECT b.x FROM a join b on a.x=b.x group by (b.x);

```
Nested Loop
  ->  Index Scan using a_y_idx on a
        Index Cond: (y = '6'::numeric)
  ->  Index Only Scan using b_x_y_idx on b
        Index Cond: (x = a.x)
```

It is filtered by a.x parameter!

HashJoin is not parameterised!

```
->  Hash Join
      Hash Cond: (b.x = a.x)
        ->  Index Only Scan using b_x_y_idx on b
      ->  Hash
        ->  Index Scan using a_y_idx on a
              Index Cond: (y = '6'::numeric)
```

How Nested Loop Should Be Replaced By Switch Join

NL type	Features	Switch Join construction
with pathkeys	√ Avoids Additional Sorts X Slower for large volume of data	NestedLoop + {HashJoin+Sort} NestedLoop + MergeJoin
parameterized	√ Reuse inner with parameters ⚠️ Complexity of prediction the cardinality for Planner X Slow with large outer rows	NestedLoop + HashJoin
paralleled		
tedious		

Paralleled Nested Loop

√ Uses Parallel Scans

The outer side is parallel-aware (like a parallel sequential scan)

X Might Be Slower

If the planner assumes efficient indexed lookups but gets full scans instead, the cost of repeating that scan per outer row grows fast.

```
SELECT b.x
FROM a
     join b
     on a.x=b.x
group by (b.x);
```

```
Finalize Aggregate
  ->  Gather
        Workers Planned: 1
        ->  Partial Aggregate
              ->  Nested Loop
                    ->  Parallel Index Only Scan using bidx on b
                    ->  Memoize
                          Cache Key: b.x
                          ->  Index Only Scan using a_idx on a
                                Index Cond: (x = b.x)
```

How Nested Loop Should Be Replaced By Switch Join

NL type	Features	Switch Join construction
with pathkeys	√ Avoids Additional Sorts X Slower for large volume of data	NestedLoop + {HashJoin+Sort} NestedLoop + MergeJoin
parameterized	√ Reuse inner with parameters ⚠️ Complexity of prediction the cardinality for Planner X Slow with large outer rows	NestedLoop + HashJoin
paralleled	√ Uses parallel scans X Slower for large volume of data	
tedious		

How Nested Loop Should Be Replaced By Switch Join

NL type	Features	Switch Join construction
with pathkeys	√ Avoids Additional Sorts X Slower for large volume of data	NestedLoop + {HashJoin+Sort} NestedLoop + MergeJoin
parameterized	√ Reuse inner with parameters ⚠️ Complexity of prediction the cardinality for Planner X Slow with large outer rows	NestedLoop + HashJoin
paralleled	√ Uses parallel scans X Slower for large volume of data	
tedious	√ Simplicity / Universality √ No Preprocessing Overhead X Quadratic complexity	

What Should Be Used - Nested Loop Or Switch Join

NL type	Nested Loop	Switch Join
with pathkeys	⚠️ Only if the TargetPath cost is too high	⚠️ Only if the NL cost is too high
parameterized	√ Always	√ Always
paralleled	⚠️ Only if the TargetPath cost is too high	⚠️ Only if the NL cost is too high
tedious	X Always	√ Always

The Row Count Threshold

Evaluate The Row Count Threshold

Row Count Threshold Strategies

- Fixed threshold

  (e.g. switch if row count = 1000)

- Relative threshold

  Multiply estimated rows by a trust coefficient

  Example: threshold = estimated_rows * 2.5

- Advanced heuristics

  Consider join type, data skew, or cost prediction

Evaluate The Row Count Threshold

Row Count Threshold Strategies

- Fixed threshold

  (e.g. switch if row count = 1000)

- Relative threshold

  Multiply estimated rows by a trust coefficient

  Example: threshold = estimated_rows * 2.5

- Advanced heuristics

  Consider join type, data skew, or cost prediction

Gucs in Switch Join Extension

Gucs:

to determine the row count limit:

- mistrust_factor - the coefficient to define the row count limit for switching process.
- min_limit_cardinality - the minimum of cardinality to define the row count limit.
- max_limit_cardinality - the maximum of cardinality to define the row count limit.

MIN(MAX(mistrust_factor * cardinality, min_limit_cardinality), max_limit_cardinality)

3. Extension Implementation

The Implementation of Switch Join Via The Custom Nodes

🔧 **Custom Nodes** are user-defined planner or executor nodes that can be seamlessly integrated into PostgreSQL's query processing pipeline.

Main components:

- Manage multiple subpaths or child nodes as part of their execution logic.
- Use only the planner hook (set_rel_pathlist_hook).

Switch Join is implemented using this mechanism

The Implementation of Switch Join Via The Custom Nodes

Overview of Custom Node:

- CustomPath
  - CreateCustomPlan - uses to define the function of creation the query plan with Switch Join
- CustomScanMethods
  - CreateCustomScanState - uses to define the function of formation Switch Join PlanState
- CustomExecMethods
  - BeginCustomScan - uses to initialize the Custom node before executing it.
  - ExecCustomScan - uses to form the logic of the switching process of Switch Join.
  - EndCustomScan - uses to pass through all the nodes and call the End() routine
  - ReScanCustomScan - uses to rescan nodes within Custom nodes
  - ExplainCustomScan - display useful information about Switch Join (like the row count limit)

4. Benchmarks

TPC-H Benchmark

TPC-H Benchmark

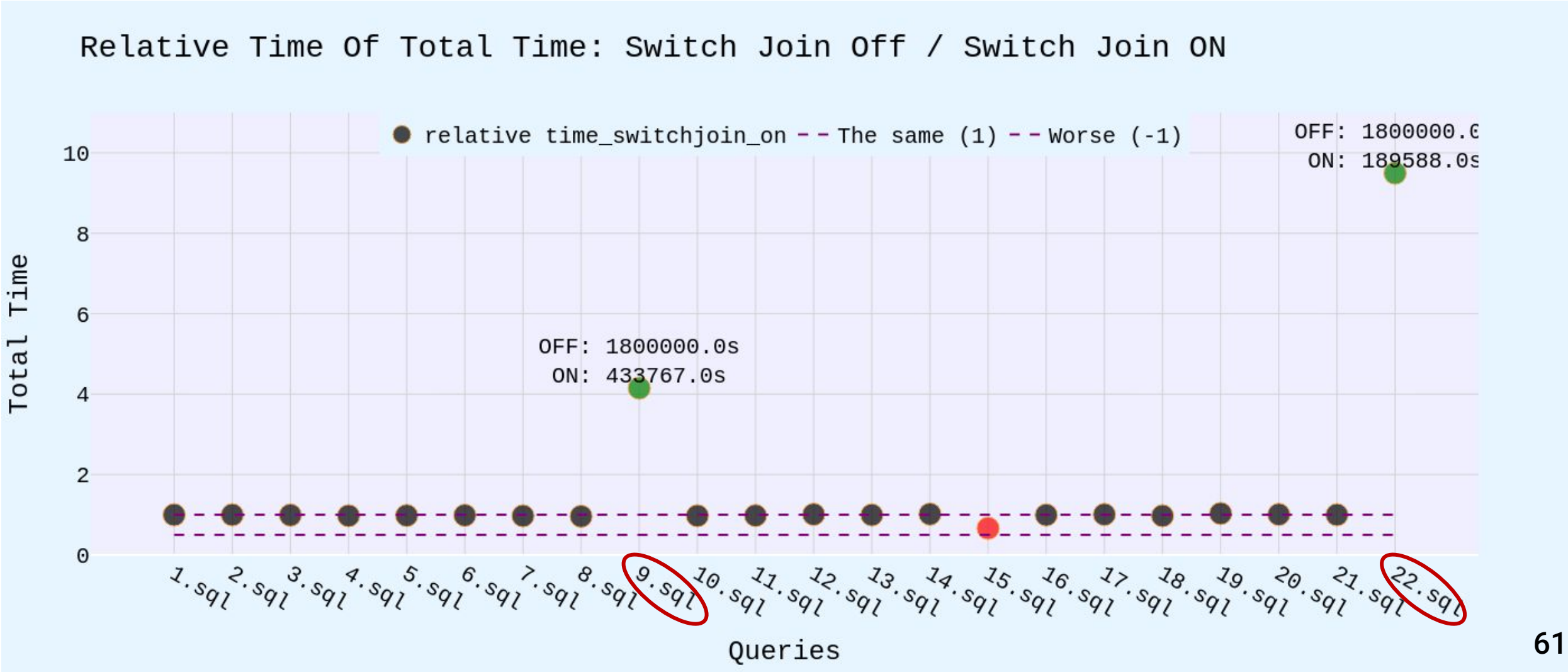
- Designed by the Transaction Processing Performance Council (TPC).
- Simulates real-world business-oriented ad hoc queries and complex data analysis workloads.
- Includes 22 complex SQL queries.
- Tests performance of query execution, indexing, joins, and optimization strategies.
- Operates on a relational database with several large tables (e.g., lineitem, orders, customer).

🎯 The testing results are available at

https://github.com/Alena0704/TPC-H-test

TPC-H Benchmark

statement_timeout = 30 minutes (triggered on queries 9.sql and 22.sql)



TPC-H: 22.sql

```
select cntrycode, count(*) as numcust, sum(c_acctbal) as totacctbal
from (select substring(c_phone from 1 for 2) as cntrycode, c_acctbal
      from customer
      where substring(c_phone from 1 for 2) in
            ('13', '31', '23', '29', '30', '18', '17')
        and c_acctbal > (
            select avg(c_acctbal) from customer
            where c_acctbal > 0.00
                and substring(c_phone from 1 for 2) in
                    ('13', '31', '23', '29', '30', '18', '17'))
        and not exists (select * from orders where o_custkey = c_custkey)) as
custsale group by cntrycode order by cntrycode LIMIT 1;
```

TPC-H: 22.sql: The Main Problem

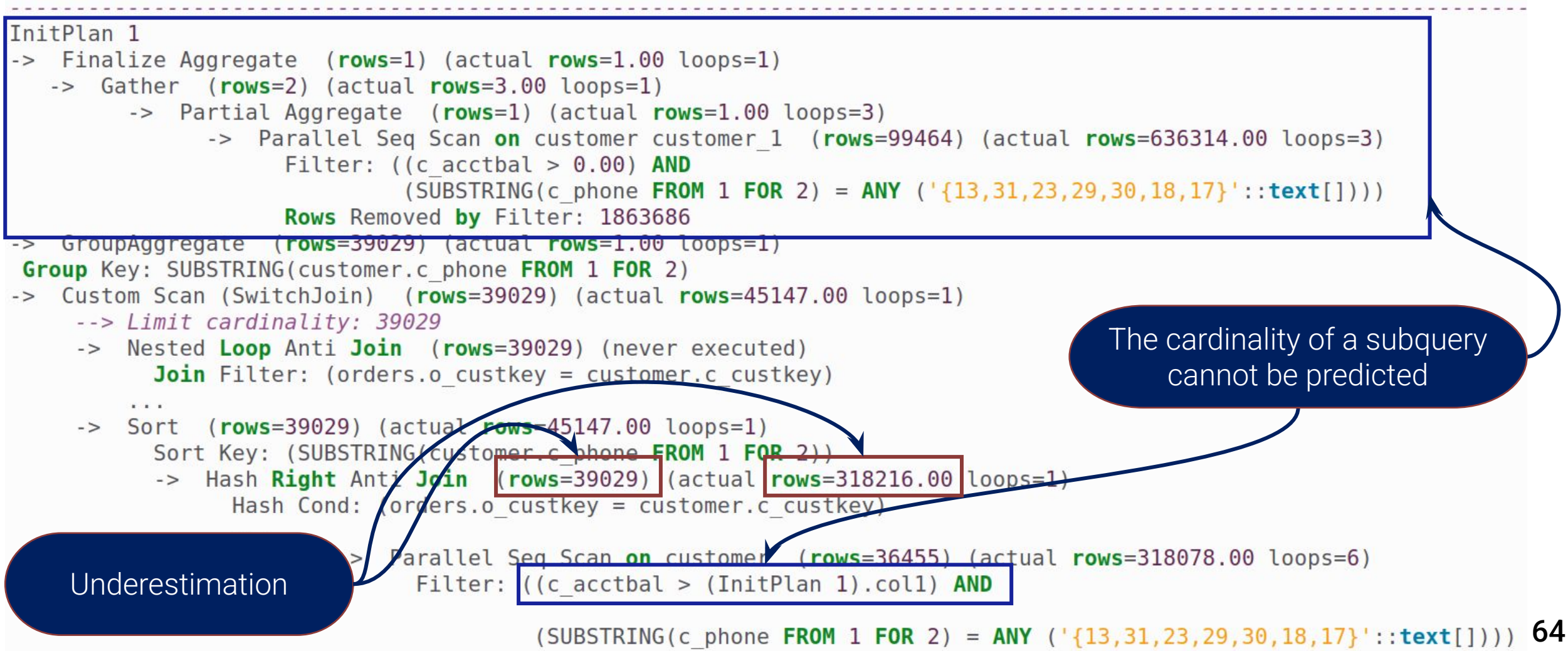
This could be fixed by calculating all InitPlan's before planning the main query

```
InitPlan 1
->  Finalize Aggregate  (rows=1) (actual rows=1.00 loops=1)
   ->  Gather  (rows=2) (actual rows=3.00 loops=1)
         ->  Partial Aggregate  (rows=1) (actual rows=1.00 loops=3)
               ->  Parallel Seq Scan on customer customer_1  (rows=99464) (actual rows=636314.00 loops=3)
                     Filter: ((c_acctbal > 0.00) AND
                              (SUBSTRING(c_phone FROM 1 FOR 2) = ANY ('{13,31,23,29,30,18,17}'::text[])))
                     Rows Removed by Filter: 1863686
->  GroupAggregate  (rows=39029) (actual rows=1.00 loops=1)
 Group Key: SUBSTRING(customer.c_phone FROM 1 FOR 2)
->  Custom Scan (SwitchJoin)  (rows=39029) (actual rows=45147.00 loops=1)
     --> Limit cardinality: 39029
     ->  Nested Loop Anti Join  (rows=39029) (never executed)
           Join Filter: (orders.o_custkey = customer.c_custkey)
           ...
     ->  Sort  (rows=39029) (actual rows=45147.00 loops=1)
           Sort Key: (SUBSTRING(customer.c_phone FROM 1 FOR 2))
           ->  Hash Right Anti Join  (rows=39029) (actual rows=318216.00 loops=1)
                 Hash Cond: (orders.o_custkey = customer.c_custkey)
                 ...
                       ->  Parallel Seq Scan on customer  (rows=36455) (actual rows=318078.00 loops=6)
                             Filter: ((c_acctbal > (InitPlan 1).col1) AND
                                      (SUBSTRING(c_phone FROM 1 FOR 2) = ANY ('{13,31,23,29,30,18,17}'::text[])))
```

The cardinality of a subquery cannot be predicted

TPC-H: 22.sql: The Main Problem

This could be fixed by calculating all InitPlan's before planning the main query



TPC-H: 9.sql

```
select nation, o_year, sum(amount) as sum_profit
from (  select n_name as nation, extract(year from o_orderdate) as o_year,
        l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
    from part, supplier, lineitem, partsupp, orders, nation
    where s_suppkey = l_suppkey and
        ps_suppkey = l_suppkey and ps_partkey = l_partkey
        and p_partkey = l_partkey and o_orderkey = l_orderkey
        and s_nationkey = n_nationkey and p_name like '%green%') as profit
group by nation, o_year
order by nation, o_year desc
LIMIT 1;
```

TPC-H: 9.sql: The Main Problem

A combination of fields (Suppkey, Partkey) represents a single interconnected entity.

The planner estimates their selectivity by multiplying independent probabilities.

```
-------------------------------------------------------------------------------
->  Nested Loop  (rows=131)
     Join Filter: (lineitem.l_orderkey = orders.o_orderkey)
     ->  Nested Loop  (rows=131)
           Join Filter: (supplier.s_suppkey = lineitem.l_suppkey)
           ->  Gather Merge  (rows=500000)
                 ->  Sort  (rows=208333)
                       Sort Key: nation.n_name
                       ->  Hash Join  (rows=208333)
                             Hash Cond: (supplier.s_nationkey = nation.n_nat
                             ...
           ->  Materialize  (rows=131)
                 ->  Gather  (rows=131)
                       ->  Parallel Hash Join  (rows=55)
                             Hash Cond: ((lineitem.l_suppkey = partsupp.ps_suppkey) AND
                                         (lineitem.l_partkey = partsupp.ps_partkey))
                             ->  Parallel Seq Scan on lineitem  (rows=124998853)
                             ->  Parallel Hash  (rows=887028)
```

The main problem

TPC-H: 9.sql: The Main Problem

A combination of fields (Suppkey, Partkey) represents a single interconnected entity.

The planner estimates their selectivity by multiplying independent probabilities.

```
->  Nested Loop  (rows=131)
      Join Filter: (lineitem.l_orderkey = orders.o_orderkey)
      ->  Nested Loop  (rows=131)
            Join Filter: (supplier.s_suppkey = lineitem.l_suppkey)
            ->  Gather Merge  (rows=500000)
                  ->  Sort  (rows=208333)
                        Sort Key: nation.n_name
                        ->  Hash Join  (rows=208333)
                              Hash Cond: (supplier.s_nationkey = nation.n_nat
                              ...
            ->  Materialize  (rows=131)
                  ->  Gather  (rows=131)
                        ->  Parallel Hash Join  (rows=55)
                              Hash Cond: ((lineitem.l_suppkey = partsupp.ps_suppkey) AND
                                          (lineitem.l_partkey = partsupp.ps_partkey))
                              ->  Parallel Seq Scan on lineitem  (rows=124998853)
                              ->  Parallel Hash  (rows=887028)
```

The planner chooses incorrectly Nested Loop to execute 16_000_000 rows

The main problem

TPC-H: 9.sql: The Solution With Switch Join

Switch Join contains two algorithms: Nested Loop and Hash Join + Sort

After considering 100 rows it switches to Hash Join.

```
-------------------------------------------------------------------------------------------------
->  Custom Scan (SwitchJoin)  (rows=96) (actual rows=652655.00 loops=1)
   --> Limit cardinality: 100
   ->  Nested Loop  (rows=96) (never executed)
         Join Filter: (supplier.s_suppkey = lineitem.l_suppkey)
         ...
   ->  Sort  (rows=96) (actual rows=652655.00 loops=1)
         ->  Hash Join  (rows=96) (actual rows=16323485.00 loops=1)
               Hash Cond: (lineitem.l_suppkey = supplier.s_suppkey)
               ->  Gather  (rows=96) (actual rows=16323485.00 loops=1)
                     ->  Parallel Hash Join  (rows=40) (actual rows=5441161.67 loops=3)
                           Hash Cond: (orders.o_orderkey = lineitem.l_orderkey)
                           ->  Parallel Seq Scan on orders  (rows=31250590) (actual rows=25000000.00)
                           ->  Parallel Hash  (rows=40) (actual rows=5441161.67 loops=3)
                                 ->  Parallel Hash Join  (rows=40) (actual rows=5441161.67 loops=3)
                                       Hash Cond: ((partsupp.ps_suppkey = lineitem.l_suppkey) AND
                                                   (partsupp.ps_partkey = lineitem.l_partkey))
                                       ->  Parallel Seq Scan on partsupp  (rows=1666827) (actual rows=133
                                       ->  Parallel Hash  (rows=5051262) (actual rows=5441161.67 loops=3)
                                             ...
```

The main problem

TPC-H: 9.sql: The Solution With Extended Statistics

🔧 Solution: Add Extended Statistics (Distinct):

l_suppkey, l_partkey and ps_suppkey, ps_partkey

```
------------------------------------------------------------------------------------------------------------------------
...
->  Sort  (rows=57) (actual rows=19780.67 loops=3)
     ->  Hash Join  (rows=57) (actual rows=5441161.67 loops=3)
           Hash Cond: (supplier.s_nationkey = nation.n_nationkey)
           ->  Parallel Hash Join  (rows=57) (actual rows=5441161.67 loops=3)
                 Hash Cond: (orders.o_orderkey = lineitem.l_orderkey)
                 ->  Parallel Seq Scan on orders  (rows=31250000) (actual rows=25000000.00 loops=3)
                 ->  Parallel Hash  (rows=57) (actual rows=5441161.67)
                       ->  Parallel Hash Join  (rows=57) (actual rows=5441161.67 loops=3)
                             Hash Cond: ((lineitem.l_suppkey = supplier.s_suppkey) AND
                                         (lineitem.l_partkey = partsupp.ps_partkey))
                             ->  Parallel Seq Scan on lineitem  (rows=125002387) (actual rows=100001937.00 loops=3)
                             ->  Parallel Hash  (rows=946057) (actual rows=725634.67 loops=3)
...
Planning Time: 43.892 ms
Execution Time: 379425.591 ms
```

TPC-H: 9.sql: The Solution With Switch Join

Switch Join contains Hash Join node

After observing 100 tuples it switches node from Nested Loop to Hash Join

```
->  Custom Scan (SwitchJoin)  (rows=96) (actual rows=652655.00 loops=1)
   --> Limit cardinality: 100
   ->  Nested Loop  (rows=96) (never executed)
         Join Filter: (supplier.s_suppkey = lineitem.l_suppkey)
   ->  Sort  (rows=96) (actual rows=652655.00 loops=1)
         ->  Hash Join  (rows=96) (actual rows=16323485.00 loops=1)
               Hash Cond: (lineitem.l_suppkey = supplier.s_suppkey)
               ->  Gather  (rows=96) (actual rows=16323485.00 loops=1)
                     ->  Parallel Hash Join  (rows=40) (actual rows=5441161.67 loops=3)
                           Hash Cond: (orders.o_orderkey = lineitem.l_orderkey)
                           ->  Parallel Seq Scan on orders  (rows=31250590) (actual rows=25000000.00)
                           ->  Parallel Hash  (rows=40) (actual rows=5441161.67 loops=3)
                                 ->  Parallel Hash Join  (rows=40) (actual rows=5441161.67 loops=3)
                                       Hash Cond: ((partsupp.ps_suppkey = lineitem.l_suppkey) AND
                                                   (partsupp.ps_partkey = lineitem.l_partkey))
                                       ->  Parallel Seq Scan on partsupp  (rows=1666827) (actual rows=133
                                       ->  Parallel Hash  (rows=5051262) (actual rows=5441161.67 loops=3)
                                             ...
```

The main problem

Join Order Benchmark

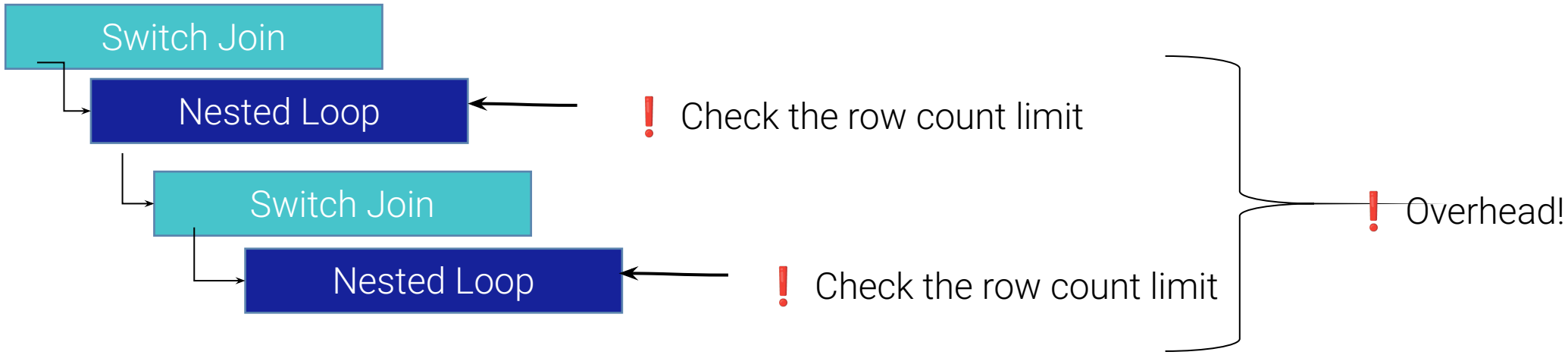
Join Order Benchmark (JOB)

- set of 113 queries
- every query has from 3 to 16 joins
- the queries answer the logical questions of a movie lover
- queries are difficult for the optimizer due to the large number of joins and correlations

🎯 The testing results are available at https://github.com/Alena0704/jo-bench/tree/switch_join_test

Hypotheses

- Using multiple Switch Joins can give noticeable overhead
- Using one Switch Join can give lower overhead but at the same time give less effect



🎯 The testing results are available at https://github.com/Alena0704/jo-bench/tree/switch_join_test

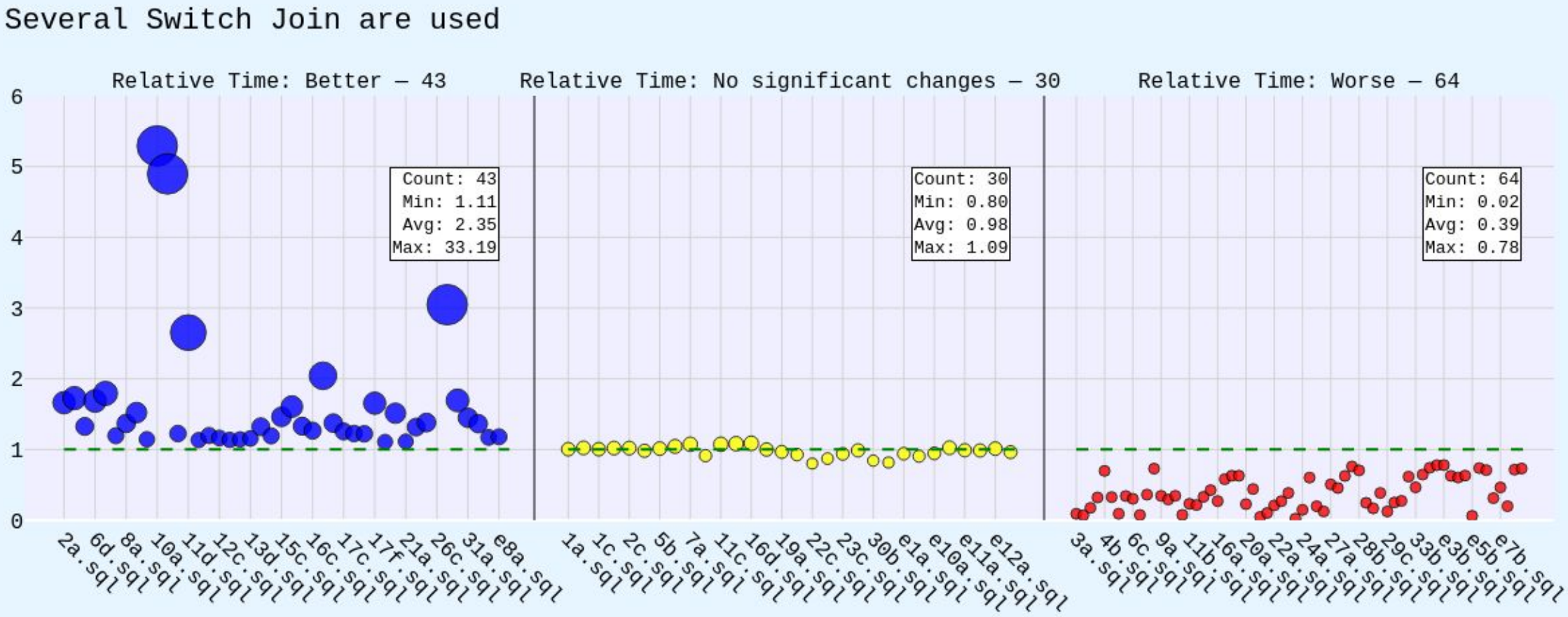
JOB: Without Limitations. 14b.sql

```
Aggregate  (cost=4896.11..4896.12 rows=1 width=64)
  ->  Custom Scan (SwitchJoin)  (cost=8.10..4896.11 rows=1 width=22)
        --> Limit cardinality: 100
        ->  Nested Loop  (cost=8.10..4896.11 rows=1 width=22)
              Join Filter: (kt.id = t.kind_id)
              ->  Materialize  (cost=8.10..4895.01 rows=1 width=26)
                    ->  Custom Scan (SwitchJoin)  (cost=8.10..4895.01 rows=1 width=26)
                          --> Limit cardinality: 100
                          ->  Nested Loop  (cost=8.10..4895.01 rows=1 width=26)
                                Join Filter: (it2.id = mi_idx.info_type_id)
                                ->  Materialize  (cost=8.10..4892.59 rows=1 width=30)
                                      ->  Custom Scan (SwitchJoin)  (cost=8.10..4892.58 rows=1 width=30)
                                            --> Limit cardinality: 100
                                            ->  Nested Loop  (cost=8.10..4892.58 rows=1 width=30)
                                                  Join Filter: (mi_idx.movie_id = t.id)
                                                  ->  Materialize  (cost=7.67..4892.08 rows=1 width=33)
                                                        ->  Nested Loop  (cost=7.67..4892.07 rows=1 width=33)
                                                              Join Filter: (it1.id = mi.info_type_id)
                                                              ->  Custom Scan (SwitchJoin)  (cost=7.67..4889.65 rows=1)
                                                                    --> Limit cardinality: 100
                                                                    ->  Nested Loop  (cost=7.67..4889.65 rows=1 width=37)
                                                                          Join Filter: (mi.movie_id = t.id)
```

JOB: Results

Several Switch Join were generated in the query plans

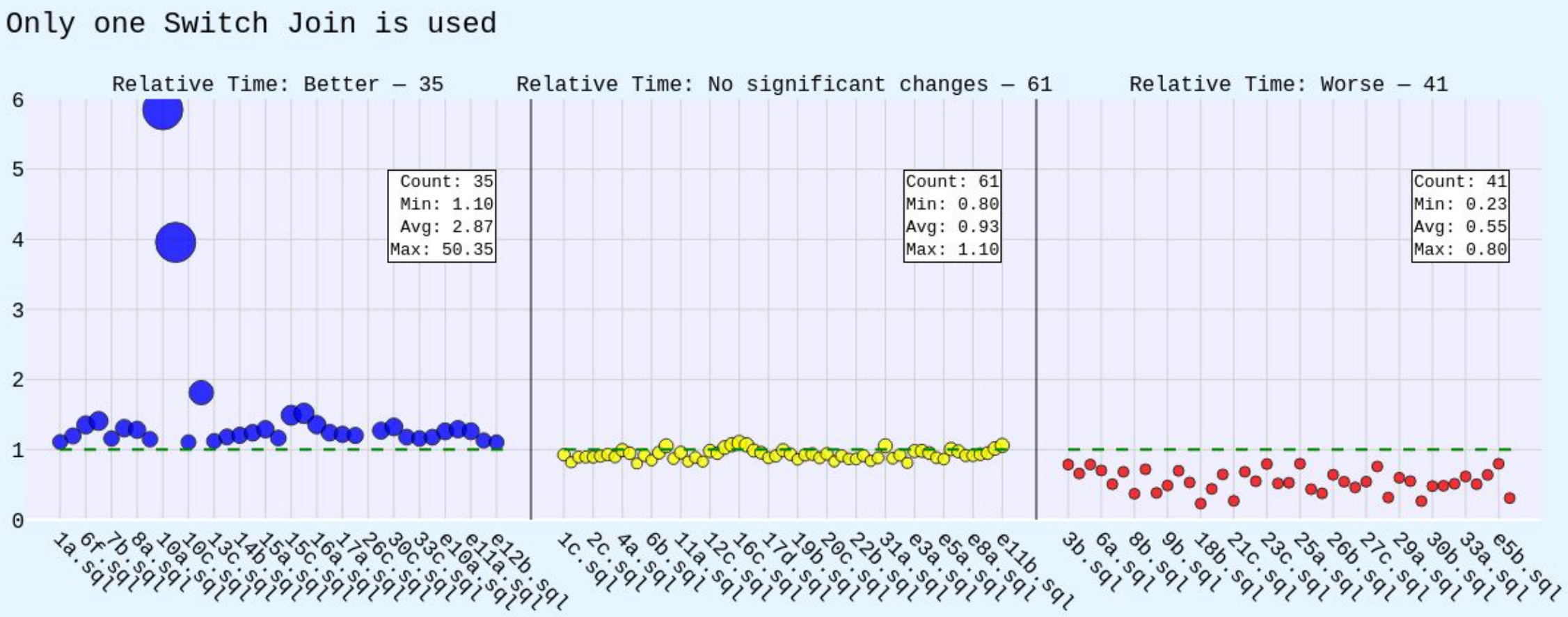
Some queries work slower because of overhead



JOB: One Switch Join

Allow using only one Switch Join in the query plan

Some queries work a bit faster because of less overhead but they execute longer



Is It Possible To Find a Compromise?

It’s necessary because the optimizer tends to choose a Nested Loop when:

- Cardinality prediction failure
- Many conditions
- Function use

❗ Switch Join helps mitigate these risks.

Summary

- In the absence of restrictions on creating a Switch Join in the query plan, additional overhead may occur.
- When the query plan contains only a single Switch Join, the impact and associated overhead are relatively minor.
- However, there are scenarios where multiple Switch Joins are necessary - particularly when the planner cannot rely on statistics to accurately estimate cardinality.

4. Conclusion and development

Conclusion

- Switch Join can be an effective approach to mitigate risks caused by suboptimal plan choices.
- Switch Join introduces overhead and cannot be applied blindly — it should be guided by a smart decision strategy.

Where can it be found?

- Switch join is not open source and may only be available in PostgresPro Standard (in August)
- The docker container (alena0704/switch_join) (PostgreSQL 17).

Next Steps for Switch Join

- Develop parallel execution support
- Implement parameterization support
- Develop a more sophisticated approach to determine row count thresholds, including cost-based and empirical methods
- Explore the construction of alternative algorithm switching strategies, such as:
  - Hash Join -> Nested Loop
  - Index Scan -> Seq Scan

PGConf.dev 2025

PostgresPro

Thank You for Your Attention!

Speaker: Alena Rybakina

LinkedIn:

https://www.linkedin.com/in/alena-rybakina

Authors:

Idea: Andrei Lepikhov

Implementation: Alena Rybakina <a.rybakina@postgrespro.ru>,

Andrei Lepikhov<lepihov@gmail.com>