Semi-Joins in Postgres

Richard Guo
PostgreSQL Major Contributor & Committer

2026.pgconf.dev

Agenda

1 What is a semi-join?

2 Where semi-joins come from

3 Implementation strategies

4 Improvements in recent releases

What is a semi-join?

Definition

A semi-join returns rows from the **Left-Hand Side (LHS)** where *at least one match* exists in the **Right-Hand Side (RHS)**.

- No duplication of LHS rows.
- RHS columns are not returned.
- Stops scanning RHS after first match.

Relational algebra

$$A \ltimes B$$

semi-join of A with B

Conceptually:
filter A by the existence of matches in B.

EDB POSTGRES AI

Where semi-joins come from

You can't write ***`SEMI JOIN`*** *in SQL — the planner derives it from sublinks.*

Syntax variations

Standard EXISTS

```
SELECT * FROM table_a a
WHERE EXISTS (
  SELECT 1 FROM table_b b
  WHERE a.id = b.a_id
);
```

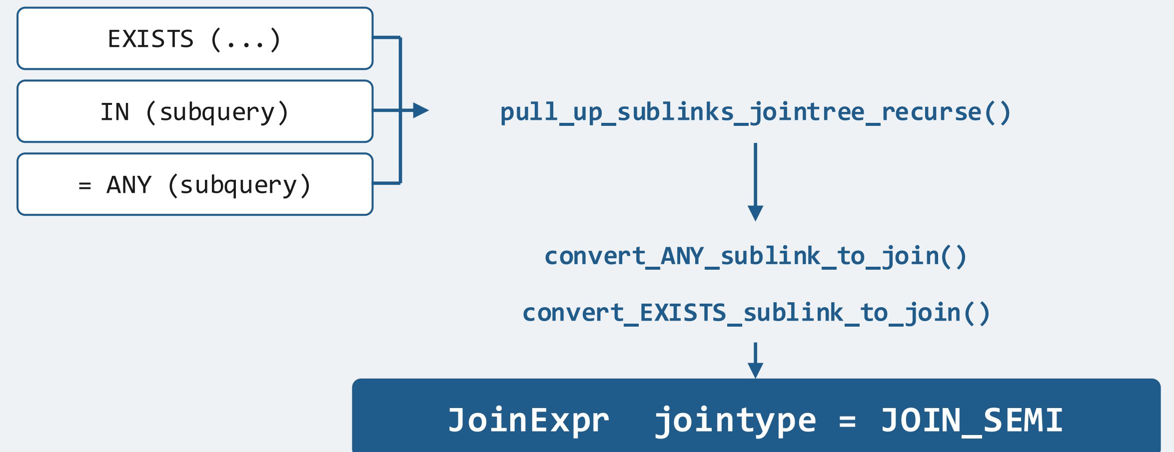
IN Clause

```
SELECT * FROM table_a
WHERE id IN (SELECT a_id FROM table_b);
```

= ANY Operator

```
SELECT * FROM table_a
WHERE id = ANY (SELECT a_id FROM table_b);
```

From SQL to JOIN_SEMI



Caveats

- Pull-up requires the subquery to be safe:
  - no volatile functions in the subquery
  - new JoinExprs must not land on the non-nullable side of an outer join
  - …

Strategy 1: the JOIN_SEMI node

A native execution mode

Supported by all three join algorithms — nested loop, hash, and merge.

Mechanism

- Scan outer relation (LHS).
- Probe inner relation (RHS).
- At first match, emit LHS tuple and advance to next outer tuple.

A small optimization

When the planner can prove the RHS is unique on the join keys, the semi-join is reduced to a plain inner join.
See **reduce_unique_semijoins()**

Example plan

```
Hash Semi Join
  Hash Cond: (a.id = b.a_id)
  →  Seq Scan on table_a
  →  Hash
        →  Seq Scan on table_b
```

Reads naturally as "for each row in A, find any match in B."

EDB POSTGRES AI

Limitation: the cartesian product problem

What if the LHS of the semi-join spans more than one relation?

```
SELECT ... FROM a, b
WHERE (a.x, b.y) IN (SELECT c1, c2 FROM c);
```

The problem

The semi-join's `min_lefthand` is `{A, B}`, so if we insist on doing this as a semijoin we will first have to form the cartesian product of A × B.

The way out

We need a way to avoid the cartesian product. That motivates the second strategy: unique-ify the RHS first, then use a plain inner join.

Strategy 2: unique-ify RHS

Make the RHS distinct, then plain-join

$$A \ltimes C \rightarrow A \bowtie \text{DISTINCT}(C)$$

Once C is unique, a plain inner join behaves exactly like a semi-join (1:0 or 1:1).

Why it's a different game

- DISTINCT(C) is now just another rel.
- Inner joins are commutative and associative.
- Planner is free to join DISTINCT(C) to A first, then B — any order.
- No cartesian product of A × B is required.

Example plan

```
Hash Join
  Hash Cond: (a.id = b.a_id)
  →  Seq Scan on table_a
  →  Hash
        →  HashAggregate
              Group Key: b.a_id
              →  Seq Scan on table_b
```

HashAggregate de-duplicates the RHS; the outer node is a plain Hash Join.

EDB POSTGRES AI

v18: right semi join

Standard Hash Semi Join

Hash table is built on the **inner** side (RHS).

Problem: if RHS is HUGE and LHS is SMALL ...

```
Hash Semi Join
  Hash Cond: (s.id = h.s_id)
  → Seq Scan on small s  (Outer)
  → Hash  (Builds on Huge Table)
        → Seq Scan on huge h
```

Slow build, high memory usage.

New: Hash Right Semi Join

Hash table is built on the **outer** side (LHS).

Fix: hash whichever side is smaller.

```
Hash Right Semi Join
  Hash Cond: (h.s_id = s.id)
  → Seq Scan on huge h  (Outer)
  → Hash  (Builds on Small Table)
        → Seq Scan on small s
```

Fast build, fits in CPU cache.

v18: how it works

The mechanism, in 8 lines of executor code

`src/backend/executor/nodeHashjoin.c` — **`ExecHashJoinImpl()`**

```
+    /*
+     * In a right-semijoin, we only need the first match for each
+     * inner tuple.
+     */
+    if (node->js.jointype == JOIN_RIGHT_SEMI &&
+        HeapTupleHeaderHasMatch(HJTUPLE_MINTUPLE(node->hj_CurTuple)))
+        continue;
+
```

The trick

- **`HEAP_TUPLE_HAS_MATCH`** was already maintained for HJ_FILL_INNER — free to reuse.
- Planner side: swap inputs, mark **`JOIN_RIGHT_SEMI`**, let costing pick whichever side hashes better.

Parallel correctness footnote

RIGHT_SEMI relies on **`HEAP_TUPLE_HAS_MATCH`** to emit each inner tuple at most once. In Parallel Hash Join, two workers probing the same inner tuple can both read the bit as unset and both emit it → duplicate rows. (Bug #19094)

Short-term: Parallel plans disabled for **`JOIN_RIGHT_SEMI`** (commit 257ee7834).

Long-term: Atomic operations on the match flag.

v19: the unique-ification step — problems

The issues

- **Cheapest-total bias:** only the cheapest-total RHS path was considered; missing paths with better sort order.
- **Heuristics:** hash-based vs. sort-based unique-ification was chosen by heuristics instead of `add_path()`.
- **Redundant sorts:** input/output pathkeys ignored — extra Sort nodes in the final plan.

Example: suboptimal plan

```
explain (costs off)
select * from t t1 where t1.a in
  (select a from t t2 where a < 10)
order by t1.a;
```

```
Merge Join
   Merge Cond: (t1.a = t2.a)
   → Index Only Scan using t_a_idx on t t1
   → Sort
         Sort Key: t2.a
         → Unique
               → Sort
                     Sort Key: t2.a
                     → Index Only Scan using t_a_idx on t t2
                           Index Cond: (a < 10)
```

v19: the architectural fix

Pathify the unique-ification step

- **New `RelOptInfo`:** a dedicated relation representing the unique-ified RHS.
- **Multiple paths:** hash-based on cheapest input *and* sort-based on every interesting presorted input.
- **Competition:** all paths compete through standard `add_path()` — the globally optimal one wins.
- **Plan shape:** the unique-ified relation joins with the LHS using a plain inner join.

Side effects

- JOIN_UNIQUE_OUTER and JOIN_UNIQUE_INNER jointypes and special-case code removed (well, almost)
- T_Unique semantics unified with upper DISTINCT clause: adjacent-duplicate removal on presorted input, no UniquePath vs. UpperUniquePath anymore
- UNIQUE_PATH_NOOP related code removed — dead code

EDB POSTGRES AI

Wait, did I just slow the planner down?

```
It may be argued that this patch introduces additional planning
overhead by considering multiple unique-ification paths for the RHS.
While that is true to some extent, I don't think this is a problem.
Please bear with me a moment.

* The additional path generation only occurs in specific semijoin
cases where one input rel is exactly the RHS.  Queries without such
semijoins are not affected.

* This patch only considers the cheapest total path and presorted
paths from the original RHS.  These are typically few in number, and
each has a high likelihood of contributing to a lower overall cost for
the final plan.  I think the cost-benefit trade-off is worthwhile.

* This patch follows the convention in joinpath.c of exploring
alternative join input paths, rather than introducing novel overhead.
For example, when planning (A SEMIJOIN B), the planner considers
multiple paths from B, including its cheapest total path and any paths
with useful sort orders.  There is no clear reason why, in the
analogous case of (A INNERJOIN unique-ified(B)), we should restrict
ourselves to only one path from the unique-ified RHS.

* On the other hand, if we insist on considering only a single path
from the unique-ified RHS, we face a dilemma when the hash-based
implementation has a cheaper total cost, but the sort-based
implementation has a better sort order.  In such cases, what should
our selection criteria be?  Currently, create_unique_path() simply
compares total_cost to choose the cheaper one (even without applying a
fuzz factor), and ignores sort order entirely.  I don't think this
approach makes sense.  It is also inconsistent with the general
pathification framework, where we rely on add_path() to retain the
best path or set of paths based on cost and other metrics, rather than
using such simple heuristics.
```

Case study: 3–5× performance boost

```
create table t (a int, b int);  insert into t select i%10, i%10 from generate_series(1,50000) i;
create index on t (a, b);  analyze t;

select * from t t1, t t2 where (t1.a, t2.b) in (select a, b from t t3) order by t1.a, t2.b;
```

Before v19

```
Incremental Sort
   Sort Key: t1.a, t2.b
   Presorted Key: t1.a
   → Nested Loop
         → Merge Join
               Merge Cond: (t1.a = t3.a)
               → Index Only Scan using t_a_b_idx on t t1
               → Sort
                     Sort Key: t3.a
                     → HashAggregate
                           Group Key: t3.a, t3.b
                           → Seq Scan on t t3
         → Memoize
               Cache Key: t3.b
               Cache Mode: logical
               → Index Only Scan using t_a_b_idx on t t2
                     Index Cond: (b = t3.b)
```

73461.002 ms (CFLAGS='-O3' without cassert)

v19 dev

```
Nested Loop
   → Merge Join
         Merge Cond: (t3.a = t1.a)
         → Unique
               → Index Only Scan using t_a_b_idx on t t3
         → Index Only Scan using t_a_b_idx on t t1
   → Memoize
         Cache Key: t3.b
         Cache Mode: logical
         → Index Only Scan using t_a_b_idx on t t2
               Index Cond: (b = t3.b)
```

21377.494 ms

~3.4× this run

Thank you!