PGDU 2025

PostgresPro

Pull-up Optimization in the Postgres Planner

Alena Rybakina

About Me

Developer in Postgres Professional since 2021

Make contribution in PostgreSQL since 2023:

- OR to ANY transformation
- Values to ANY transformation
- Others

Participate in development of extensions:

- AQO
- Replaning

Outline of the Report

1. Pull-up Optimizations in Postgres
2. Types of Pull-up Optimizations in Postgres
3. The Transformation Pipeline
4. Deep Dive: EXISTS Pull-up and Its Core Issue
5. Conclusion and Future Development

PGDU 2025

PostgresPro

Pull-up Optimizations in Postgres

What Is Pull-up Optimization

Pull-up optimization rewrites the query tree by flattening subqueries into the parent query, so they are not executed as nested units.

This exposes higher-level planner optimizations such as global join-order search and construction of Equivalence Classes.

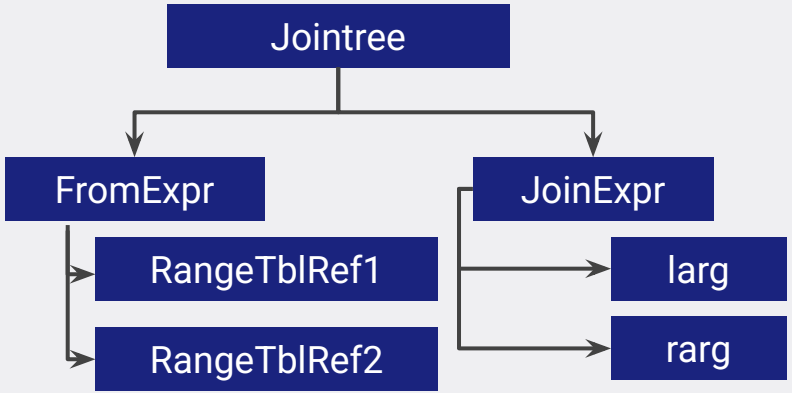
Pull-up Optimizations in Postgres

Pull-up optimization rewrites the query tree by flattening subqueries into the parent query, so they are not executed as nested units.

This exposes higher-level planner optimizations such as global join-order search and construction of Equivalence Classes.

pull_up_subqueries_recurse() walks the jointree and decides what to do based on the node type:

- RangeTblRef: inspect the referenced entry:
  - RTE_SUBQUERY
  - simple UNION ALL or VALUES clauses
  - SubLink quals
- FromExpr: recurse into each child
- JoinExpr: recurse into left (larg) and right (rarg) arguments



Simple VALUES

```
SELECT v.x + 1 AS y FROM (VALUES (42, 'eu')) AS v(x, r) WHERE v.x = 0;
-------------------------------------------------------------------------------------------------
 Values Scan on "*VALUES*"  (actual time=0.007..0.007 rows=0.00 loops=1)
   Filter: (column1 > 0)
```

```
-------------------------------------------------------------------------
 Result  (actual time=0.004..0.005 rows=0.00 loops=1)
   One-Time Filter: false
```

Simple VALUES

```
SELECT v.x + 1 AS y FROM (VALUES (42, 'eu')) AS v(x, r)
WHERE v.x = 0;
```

The subquery is transformed into one-time filter logic — evaluated once since the condition is constant.

```
------------------------------------------------------------------------------
Result (actual time=0.004..0.005 rows=0.00 loops=1)
  One-Time Filter: false
```

Simple VALUES

```
SELECT v.x + 1 AS y FROM (VALUES (42, 'eu')) AS v(x, r)
WHERE v.x = 0;
```

The One-Time Filter is formed from the result of v.x = 0, but due to x = 42, then we need to check 42 = 0, so it is false.

```
------------------------------------------------------------------------------

Result  (actual time=0.004..0.005 rows=0.00 loops=1)
  One-Time Filter: false
```

PGDU 2025

PostgresPro

Types of Pull-up Optimizations in Postgres

Main Types of Pull-up optimizations in Postgres

- Simple VALUES - Flattened directly into the upper query as a RangeTblEntry; a trivial transformation.
- SubLinks - Subqueries embedded inside expressions that can be pulled up if semantically safe:
  - ANY / ALL Subquery Expressions - Transformed into join forms when possible to leverage join planning.
  - EXISTS Subquery Expressions - Transformed into semi-joins; special handling due to correlation semantics.

EXISTS Subquery Transformation

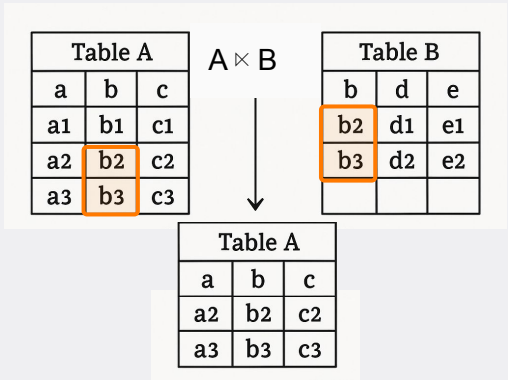
select id from ta where exists (select aval from tb where tb.id = ta.id)

We return id values from ta only when they match id values from tb.
This can be transformed into a Semi-Join.

Initial Plan
(SeqScan with ANY filter)

```
Seq Scan on ta
  Filter: (ANY
           (id = (hashed SubPlan 2).col1))
  SubPlan 2
    ->  Seq Scan on tb
```

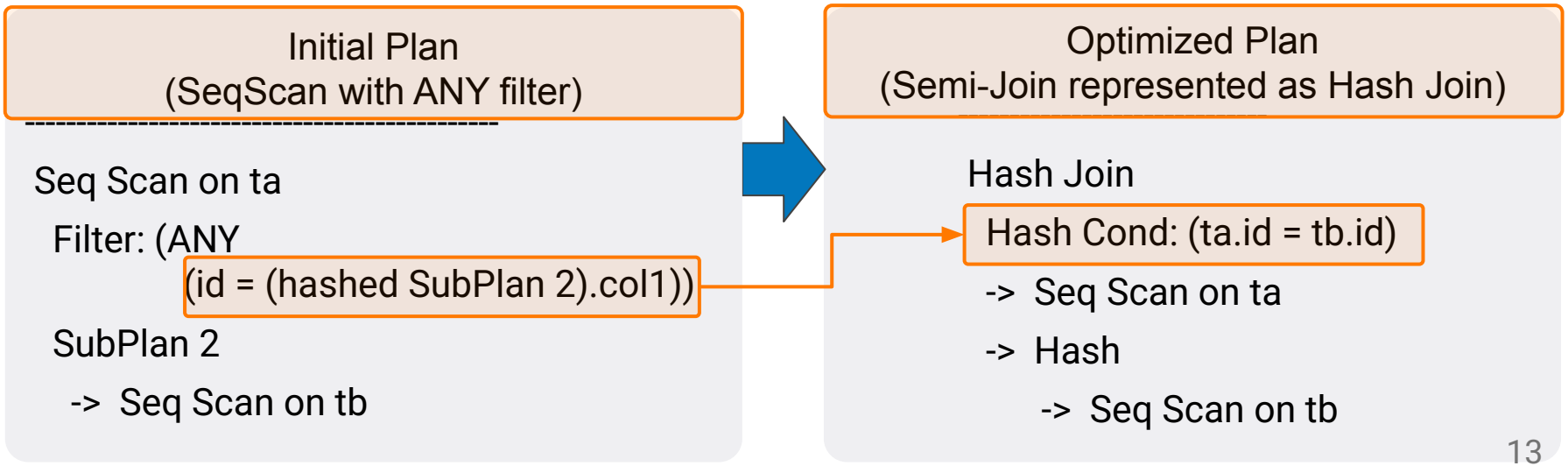
Semi-Join Visualization



EXISTS Subquery Transformation

select id from ta where exists (select aval from tb where tb.id = ta.id)

The planner replaces EXISTS with a Semi-Join for efficiency, avoiding per-row subquery execution and linking ta with tb, transforming the WHERE clause into a join condition.



ANY / IN Subquery Transformation

```
select id from ta where ta.val in (select aval from tb where tb.id = ta.id)
```

The planner replaces the IN with an Inner Join for efficiency, avoiding per-row subquery execution and linking ta with tb, transforming the WHERE and IN clause into a join condition.

Initial Plan
(SeqScan with ANY filter)

```
Seq Scan on ta
  Filter: (ANY (val = (SubPlan 1).col1))
  SubPlan 1
    ->  Index Scan using tb_pkey on tb
          Index Cond: (id = ta.id)
```

Optimized Plan

```
Hash Join
  Hash Cond:
    ((ta.val = tb.aval) AND (ta.id = tb.id))
  ->  Seq Scan on ta
  ->  Hash
        ->  Seq Scan on tb
```

NOT Exists Subquery Transformation

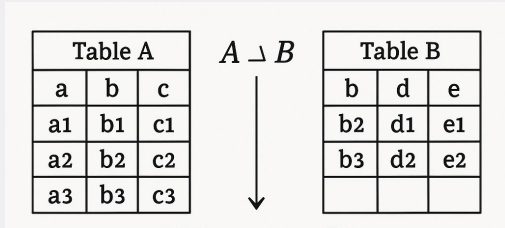
select id from ta where NOT exists (select aval from tb where tb.id = ta.id)

We return id values from ta only when they DON'T match id values from tb.
This can be transformed into an Anti-Join.

Initial Plan
(SeqScan with NOT ANY filter)

```
Seq Scan on ta
  Filter: (NOT (ANY
           (id = (hashed SubPlan 2).col1)))
  SubPlan 2
    ->  Seq Scan on tb
```

Anti-Join Visualization



NOT Exists Subquery Transformation

select id from ta where NOT exists (select aval from tb where tb.id = ta.id)

The planner replaces EXISTS with a Semi-Join for efficiency, avoiding per-row subquery execution and linking ta with tb, transforming the WHERE clause into a join condition.

Initial Plan
(SeqScan with NOT ANY filter)

```
Seq Scan on ta
  Filter: (NOT ANY
    (id = (hashed SubPlan 2).col1))
  SubPlan 2
    ->  Seq Scan on tb
```

Optimized Plan
(Anti-Join represented as Hash Anti Join)

```
Hash Anti Join
  Hash Cond: (ta.id = tb.id)
  ->  Seq Scan on ta
  ->  Hash
        ->  Seq Scan on tb
```

PGDU 2025

PostgresPro

The Transformation Pipeline

How Pull-up Works: Main Planner Pipeline

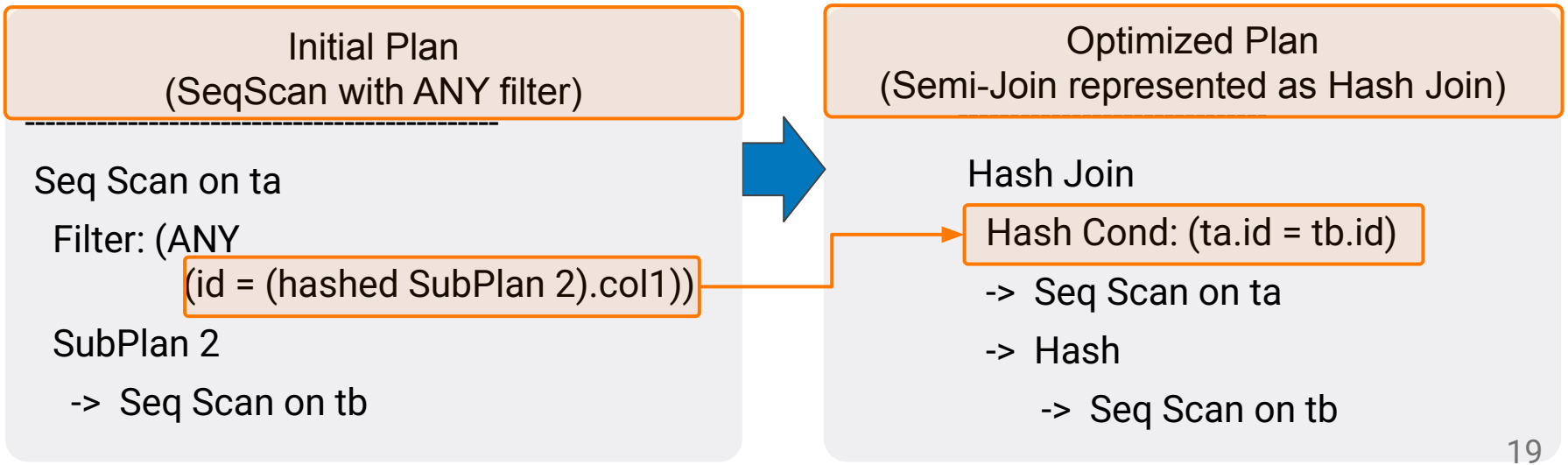
General pipeline of pull-up transformation:

- Applicable if the subquery is simple — i.e., has no grouping, aggregates, SRFs, DISTINCT, ORDER BY, or LIMIT/OFFSET clauses.
- Process subqueries:
  - Preprocess relation RTEs.
  - Pull up any SubLinks; their own children are flattened first.
- Adjust varno references and decrement varlevelsup for outer variables inside the subquery.

EXISTS Subquery Transformation

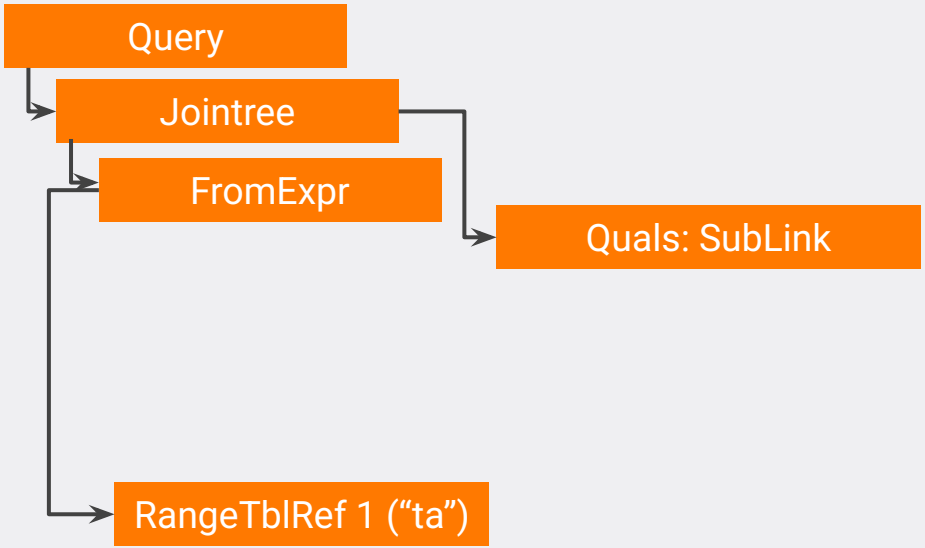
select id from ta where exists (select aval from tb where tb.id = ta.id)

The planner replaces EXISTS with a Semi-Join for efficiency, avoiding per-row subquery execution and linking ta with tb, transforming the WHERE clause into a join condition.



Query Tree - Outer Query Structure

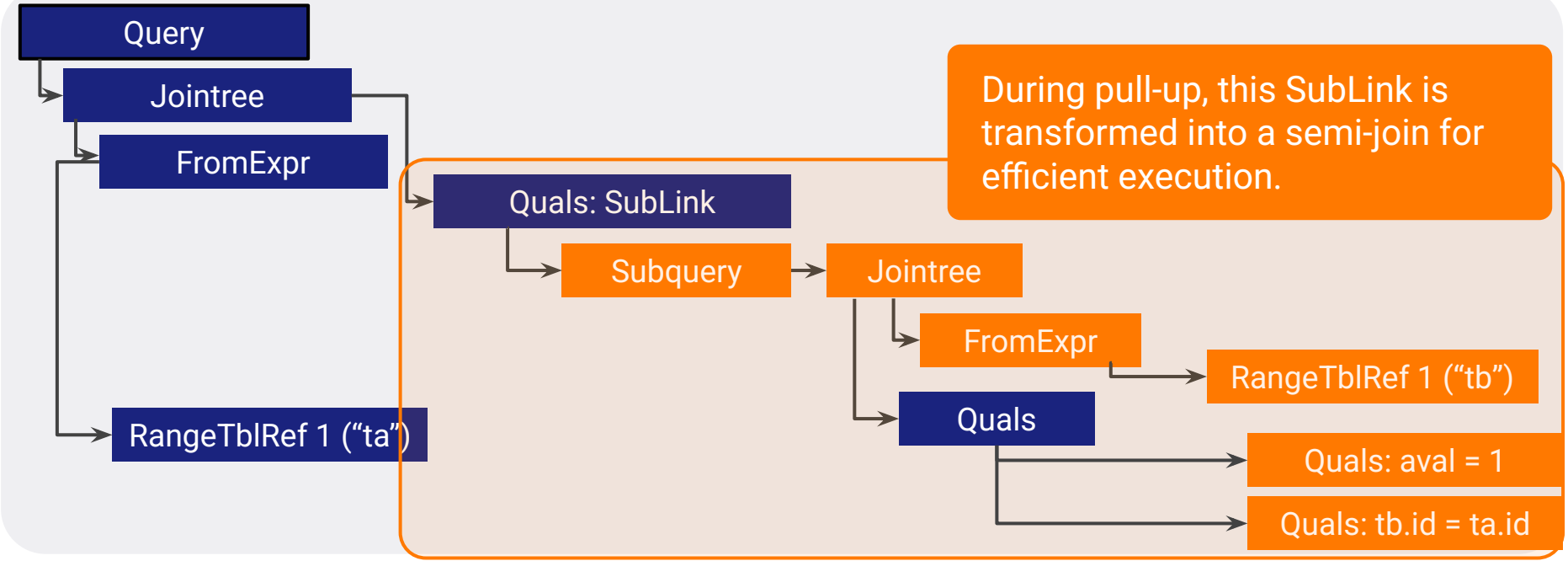
select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



The query contains a single RangeTblEntry "ta" (Index = 1) and one qual of type SubLink.

Query Tree - Subquery (SubLink) Expansion

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



Var Description

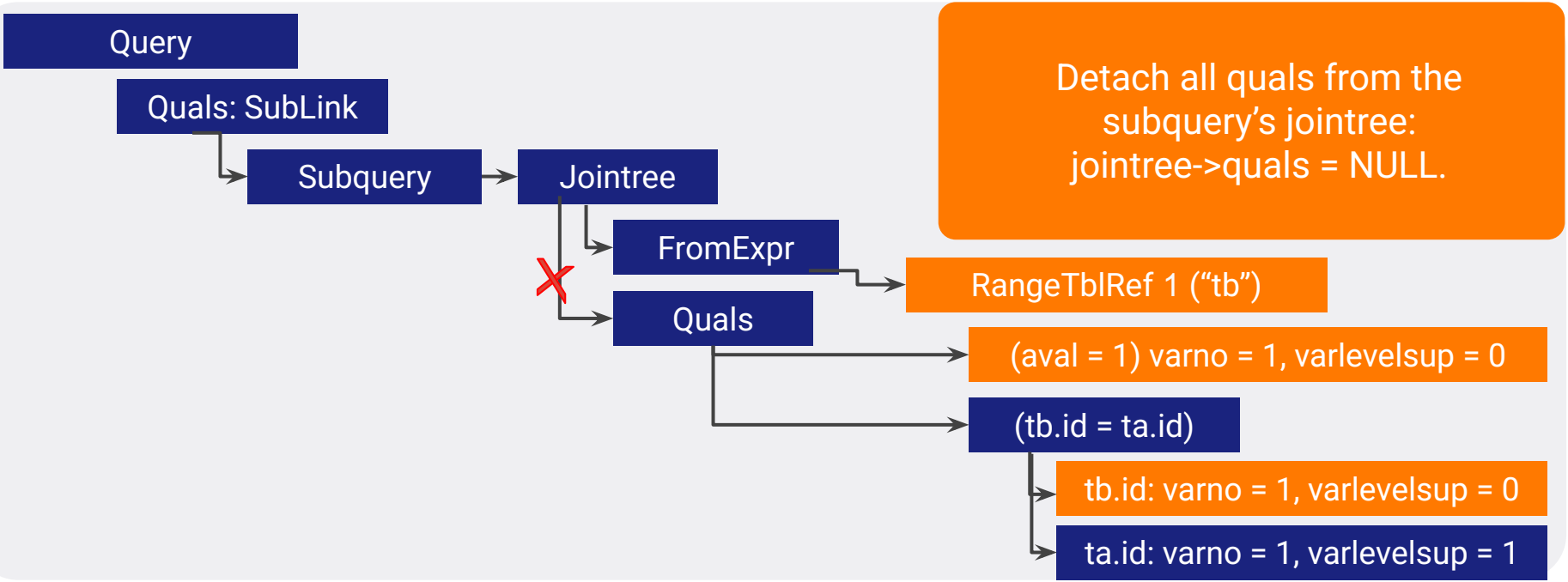
varno: “Which table (or subquery) is this from?”

It’s a Range Table Entry (RTE) index - a 1-based number pointing to the RangeTblEntry in the current Query or SubQuery level.

```
typedef struct Var
{
    Expr   xpr;
    /* expression header */
    Index  varno;
    /* range table index */
    Index  varlevelsup;
    /* levels up from current query */
    Oid    vartype;
    /* column type */
    ...
} Var;
```

Detach Quals From Subquery Jointree

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



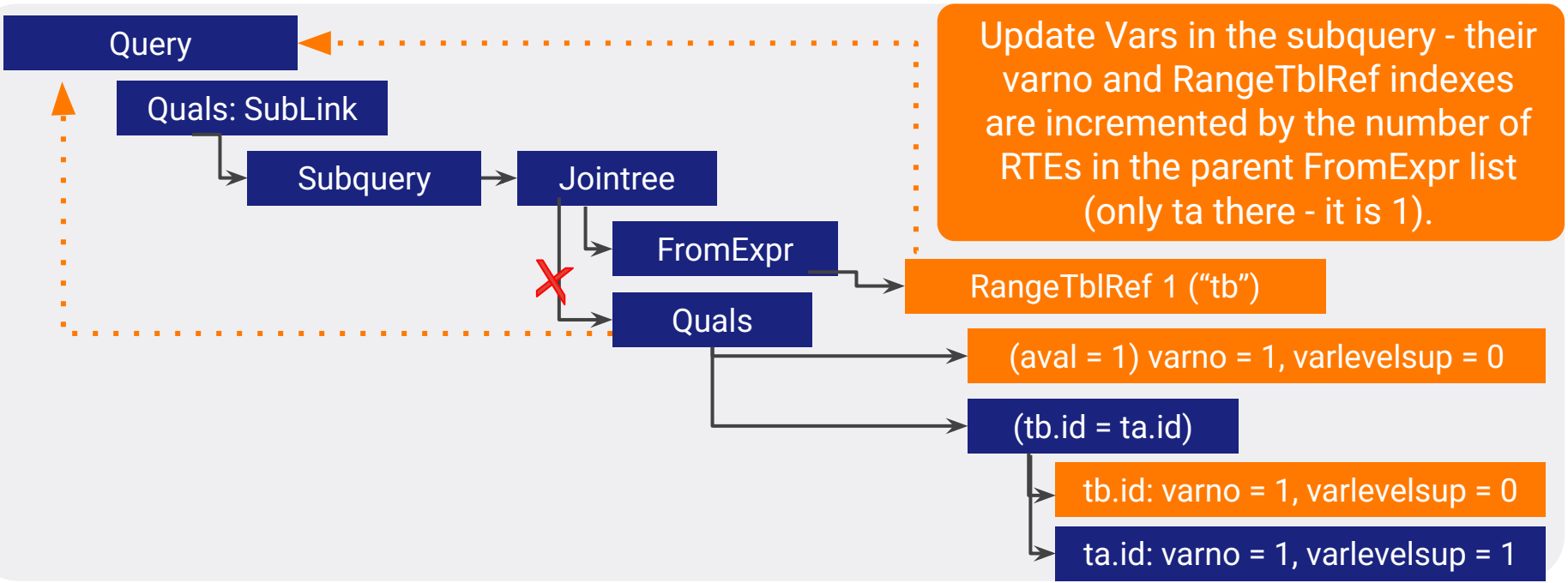
Update Vars in Subquery

If the Var is not an outer reference, update its varno using walker OffsetVarNodes() by the number of tables in the main query.

(01)

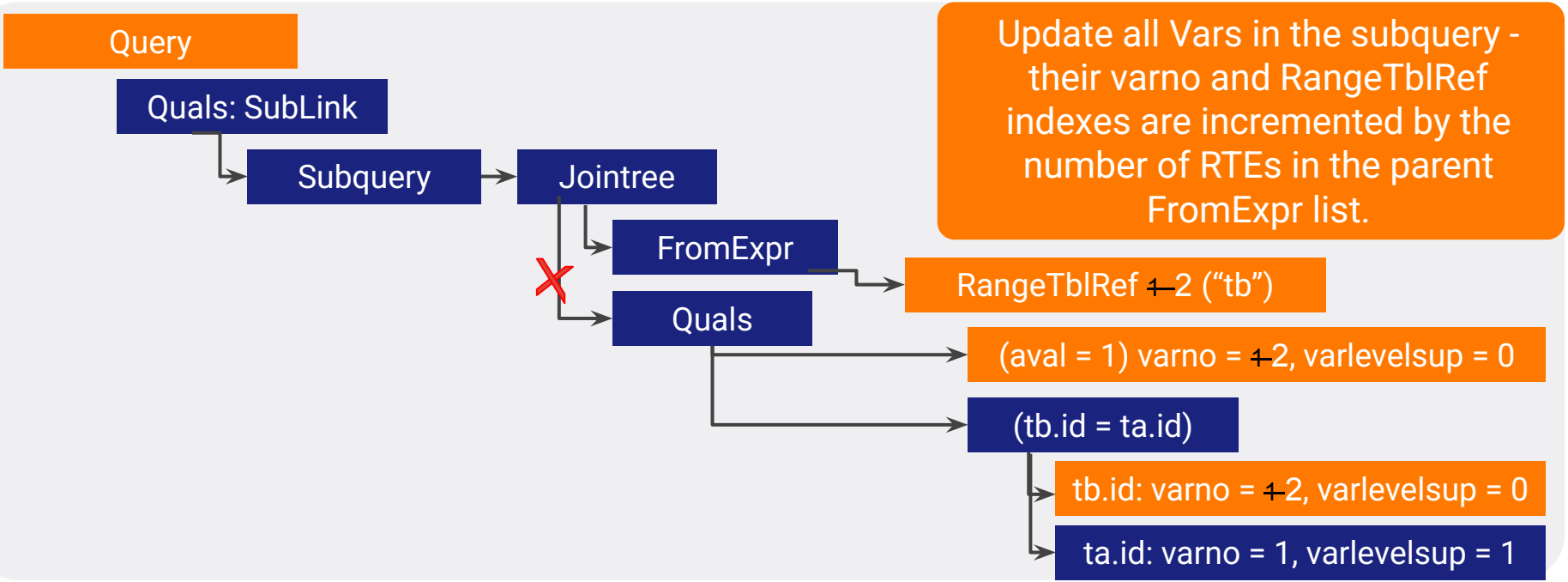
Update Outer Varnos (OffsetVarNodes)

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



Update Outer Varnos (OffsetVarNodes)

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



Update Vars in Subquery

If the Var is not an outer reference, update its varno using walker OffsetVarNodes() by the number of tables in the main query.

(01)

If the Var is an outer reference, update its varlevelsup using walker IncrementVarSublevelsUp() by -1.

27

(02)

Var Description

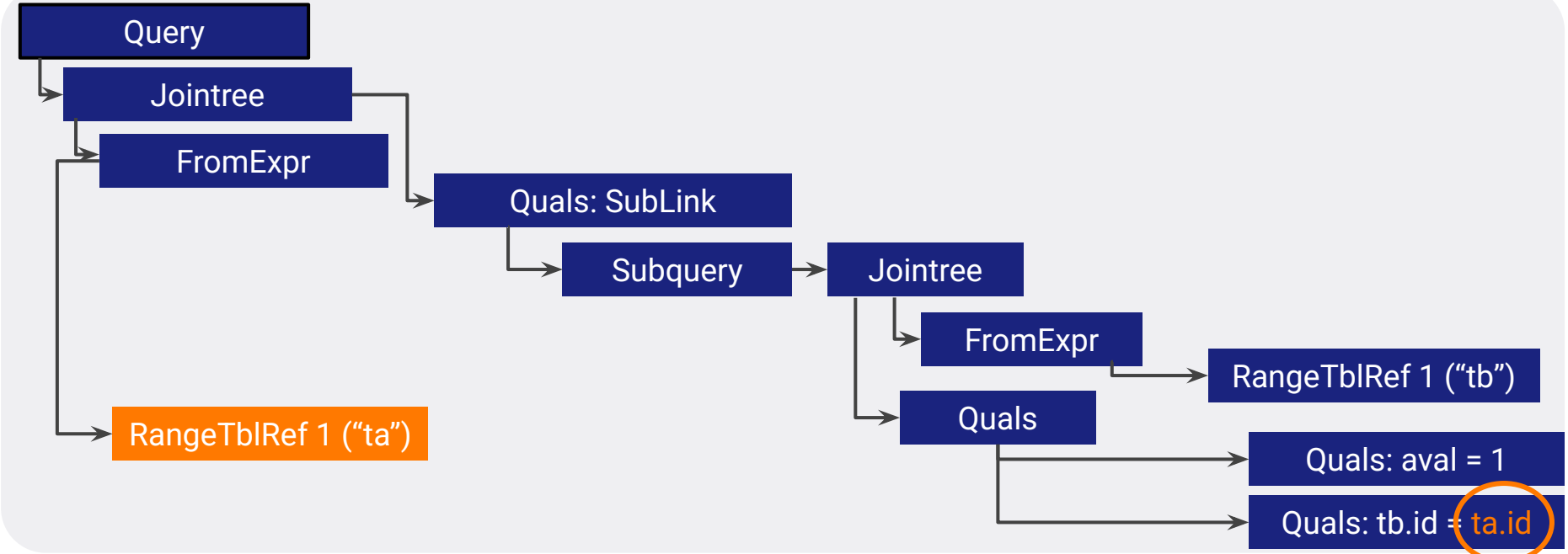
varlevelsup: How many query levels up from the current query to reach the Var's defining query?

It indicates how far up the query tree the referenced variable is located.

```
typedef struct Var
{
    Expr   xpr;
    /* expression header */
    Index  varno;
    /* range table index */
    Index  varlevelsup;
    /* levels up from current query */
    Oid    vartype;
    /* column type */
    ...
} Var;
```

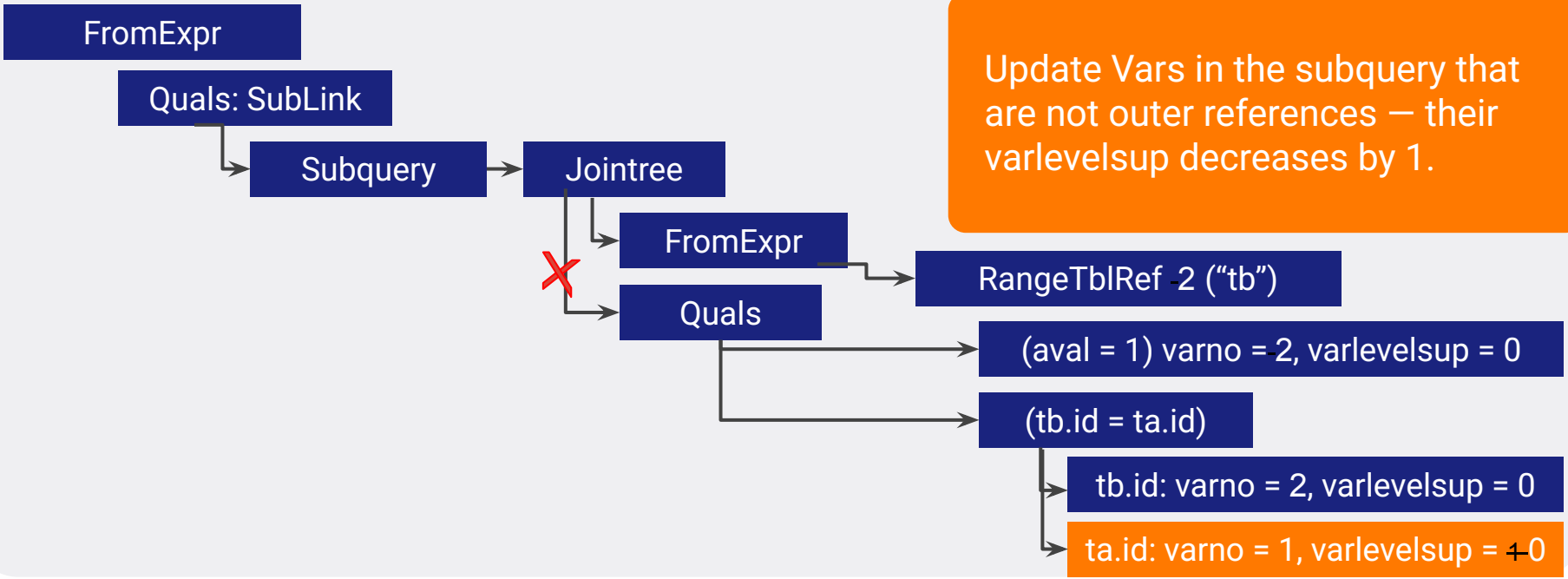
Query Tree - Subquery (SubLink) Expansion

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



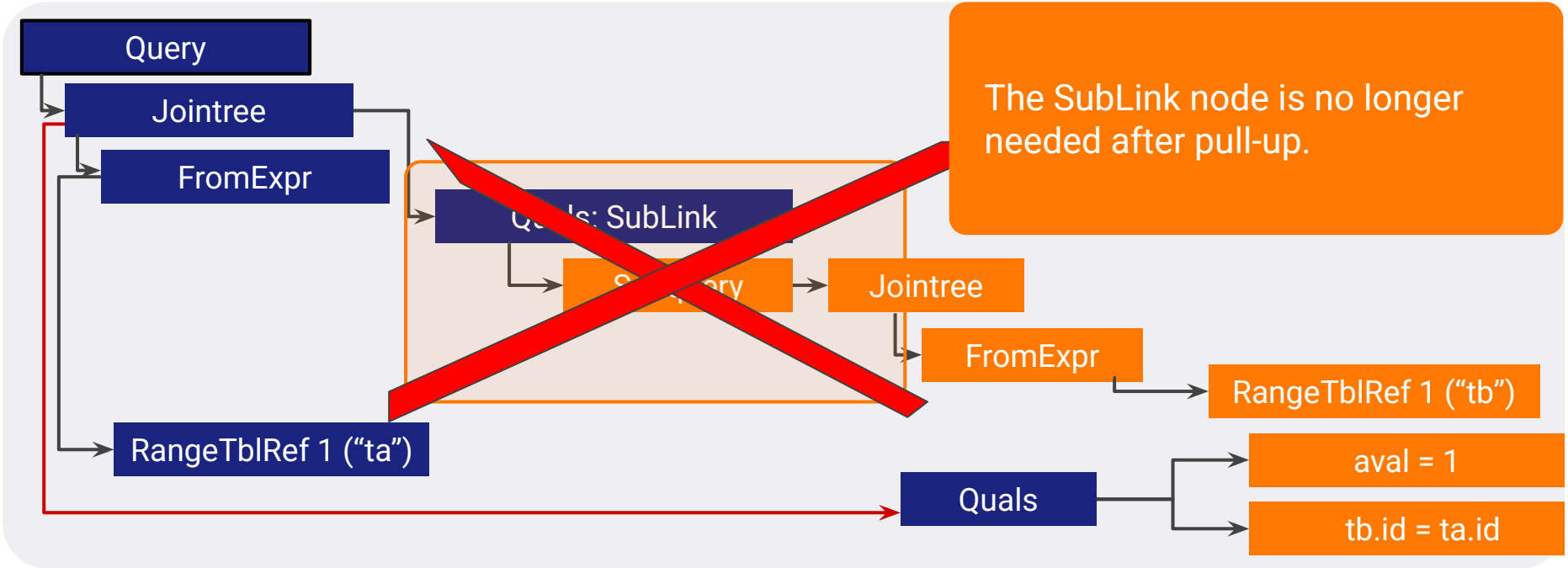
Adjust Var Levels in Subquery

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



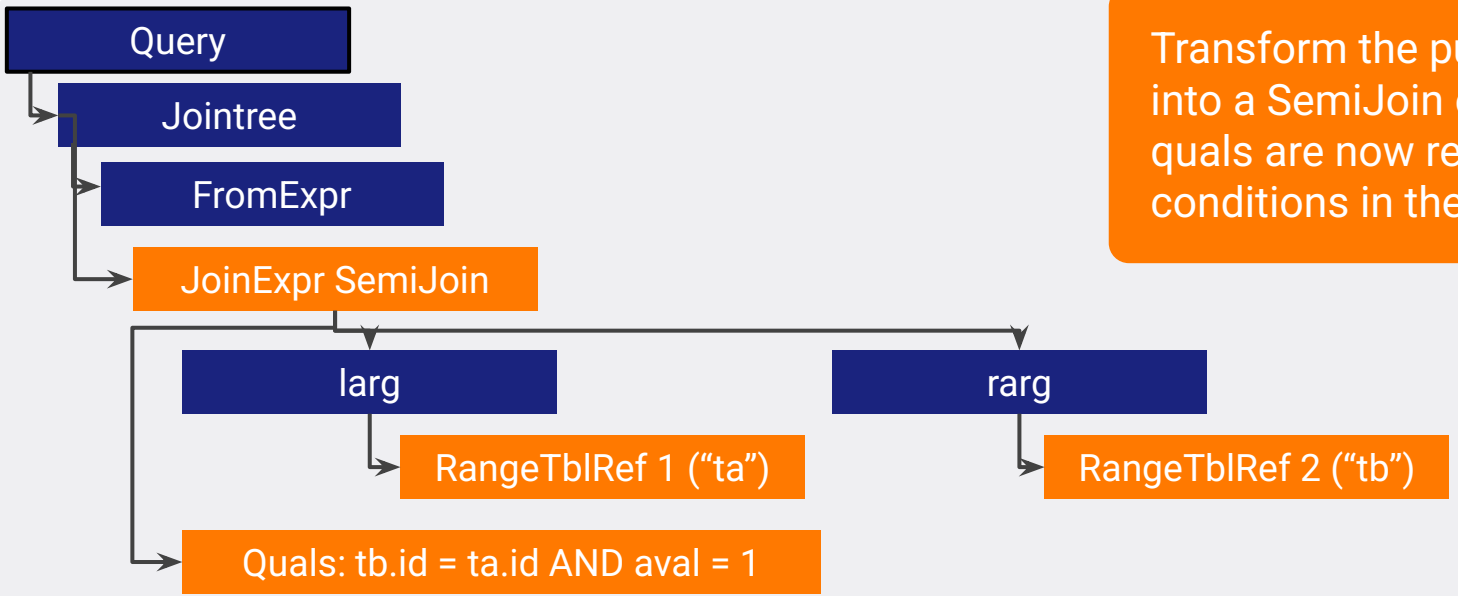
Remove SubLink Node After Pull-Up

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



Convert to SemiJoin

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



Transform the pulled-up subquery into a SemiJoin or AntiJoin. All quals are now represented as join conditions in the parent query.

PGDU 2025

PostgresPro

Deep Dive: EXISTS Pull-up and Its Core Issue

Another EXISTS SubLink Example: Concept Summary

```
ta: id | val        tb: id | aval       tc: id | aid
----+-----         ----+------         ----+-----
  1 |   1             1 |    1           3 |   1
  2 |   2             3 |    1           1 |   1
```

```
SELECT * FROM ta WHERE EXISTS (
SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);
```

```
 id | val
----+-----
  1 |   1
```

Returns all tuples from ta
for which there exists
a pair of rows in tb and tc that
share the same id value
(tb.id = ta.id = tc.id).

Another EXISTS SubLink Example: Query Plan

```
ta: id | val      tb: id | aval      tc: id | aid
   ----+-----        ----+------        ----+-----
    1 |   1          1 |    1          3 |   1
    2 |   2          3 |    1          1 |   1
```

```
SELECT * FROM ta WHERE EXISTS (
  SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);
```

```
 id | val
----+-----
  1 |   1
```

If a JoinExpr appears inside the subquery, pull-up is stopped and a SemiJoin cannot be constructed.

```
SELECT * FROM ta WHERE EXISTS (
  SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);

-------------------------------------------------------
Seq Scan on ta (rows=1.00 loops=1)
  Filter: EXISTS(SubPlan 1)
  SubPlan 1
    ->  Nested Loop  (rows=0.50 loops=2)
          ->  Seq Scan on tc (rows=0.50 loops=2)
                Filter: (ta.id = id)
          ->  Seq Scan on tb (rows=1.00 loops=1)
```

Another EXISTS SubLink Example: Query Plan

```
ta: id | val        tb: id | aval       tc: id | aid
----+-----         ----+------         ----+-----
  1 |   1             1 |    1           3 |   1
  2 |   2             3 |    1           1 |   1
```

```
SELECT * FROM ta WHERE EXISTS (
  SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);
```

```
 id | val
----+-----
  1 |   1
```

The SubPlan must be executed once for each row in ta.

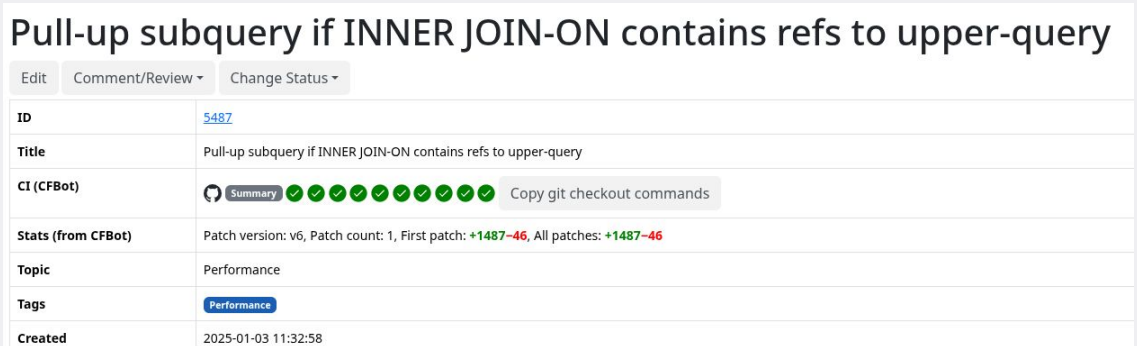
```
SELECT * FROM ta WHERE EXISTS (
  SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);

-------------------------------------------------------
Seq Scan on ta (rows=1.00 loops=1)
  Filter: EXISTS(SubPlan 1)
  SubPlan 1
    ->  Nested Loop  (rows=0.50 loops=2)
          ->  Seq Scan on tc (rows=0.50 loops=2)
                Filter: (ta.id = id)
          ->  Seq Scan on tb (rows=1.00 loops=1)
```

Proposed Patch

Enable Pull-up of EXISTS Subqueries with Inner JOIN Refs to Upper Query

Main benefit: the planner can consider join reordering and earlier filtering.



Pull-up subquery if INNER JOIN-ON contains refs to upper-query

Edit | Comment/Review | Change Status

ID	5487
Title	Pull-up subquery if INNER JOIN-ON contains refs to upper-query
CI (CFBot)	Summary ✓✓✓✓✓✓✓✓✓✓ Copy git checkout commands
Stats (from CFBot)	Patch version: v6, Patch count: 1, First patch: +1487–46, All patches: +1487–46
Topic	Performance
Tags	Performance
Created	2025-01-03 11:32:58

https://commitfest.postgresql.org/patch/5487/

Query Plan After Applying the Patch

```
ta: id | val        tb: id | aval       tc: id | aid
----+-----         ----+------         ----+-----
  1 |   1             1 |    1           3 |   1
  2 |   2             3 |    1           1 |   1
```

```
SELECT * FROM ta WHERE EXISTS (
  SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);
```

```
 id | val
----+-----
  1 |   1
```

No semantic restriction prevents constructing a SemiJoin here.

```
SELECT * FROM ta WHERE EXISTS (
  SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);

-------------------------------------------------------
Nested Loop Semi Join (rows=1.00 loops=1)
  ->  Seq Scan on ta  (rows=2.00 loops=1)
  ->  Nested Loop (rows=0.50 loops=2)
        ->  Index Only Scan using tc_pkey on tc
(rows=0.50 loops=2)
              Index Cond: (id = ta.id)
        ->  Seq Scan on tb (rows=1.00 loops=1)
```

Another EXISTS SubLink Example: Join Expansion

```
ta: id | val      tb: id | aval      tc: id | aid
   ----+-----        ----+------        ----+-----
     1 |   1          1 |    1          3 |   1
     2 |   2          3 |    1          1 |   1
```

```
SELECT * FROM ta WHERE EXISTS (
SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);
```

```
 id | val
----+-----
  1 |   1
```

The join between tb and tc first forms all possible row pairs before applying any filtering conditions.

```
SELECT tb.id as tb_id, tb.aval as tb_aval,
       tc.id as tc_id, tc.aid as ic_aid
FROM tb JOIN tc
     ON true;
tb_id | tb_aval | tc_id | ic_aid
------+---------+-------+--------
    1 |       1 |     3 |      2
    1 |       1 |     1 |      1
    3 |       1 |     3 |      2
    3 |       1 |     1 |      1
```

Another EXISTS SubLink Example: Filtering With ta

```
ta: id | val        tb: id | aval       tc: id | aid
   ----+-----          ----+------         ----+-----
     1 |   1            1 |    1           3 |   1
     2 |   2            3 |    1           1 |   1
```

```
SELECT * FROM ta WHERE EXISTS (
SELECT 1 FROM tb JOIN tc ON ta.id = tc.id);
```

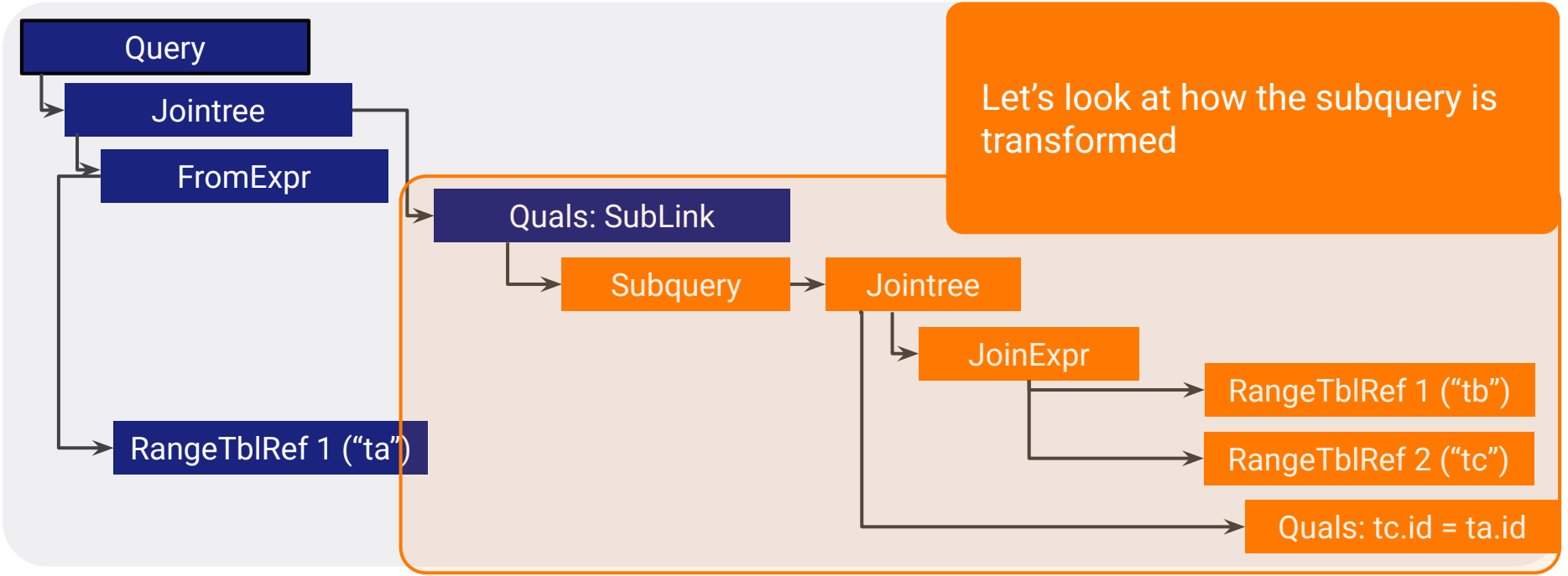
```
 id | val
----+-----
  1 |   1
```

We keep only the combinations where tc.id matches an id from ta.

```
SELECT tb.id as tb_id,
       tb.aval as tb_aval,
       tc.id as tc_id,
       tc.aid as ic_aid
FROM tb
JOIN tc
       ON tc.id in (select id from ta);
 tb_id | tb_aval | tc_id | ic_aid
--------+----------+-------+--------
      1 |        1 |     1 |      1
```

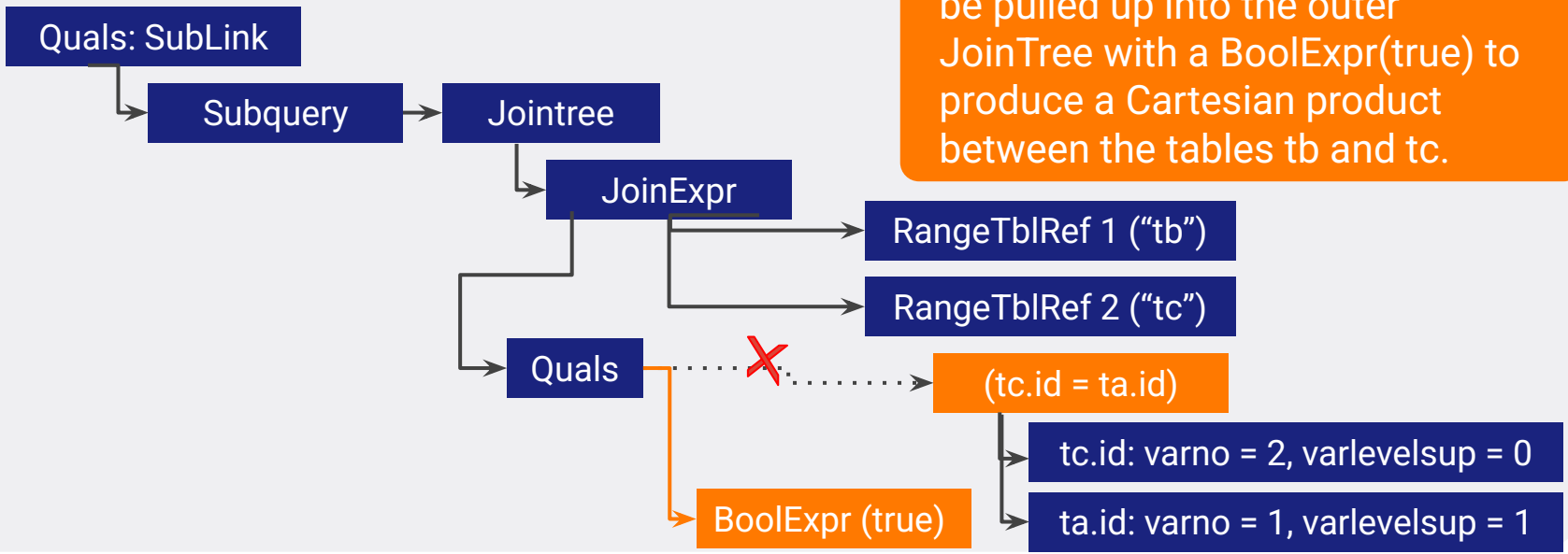
Query Tree

select id from ta where exists (select aval from tb join tc on tc.id = ta.id)



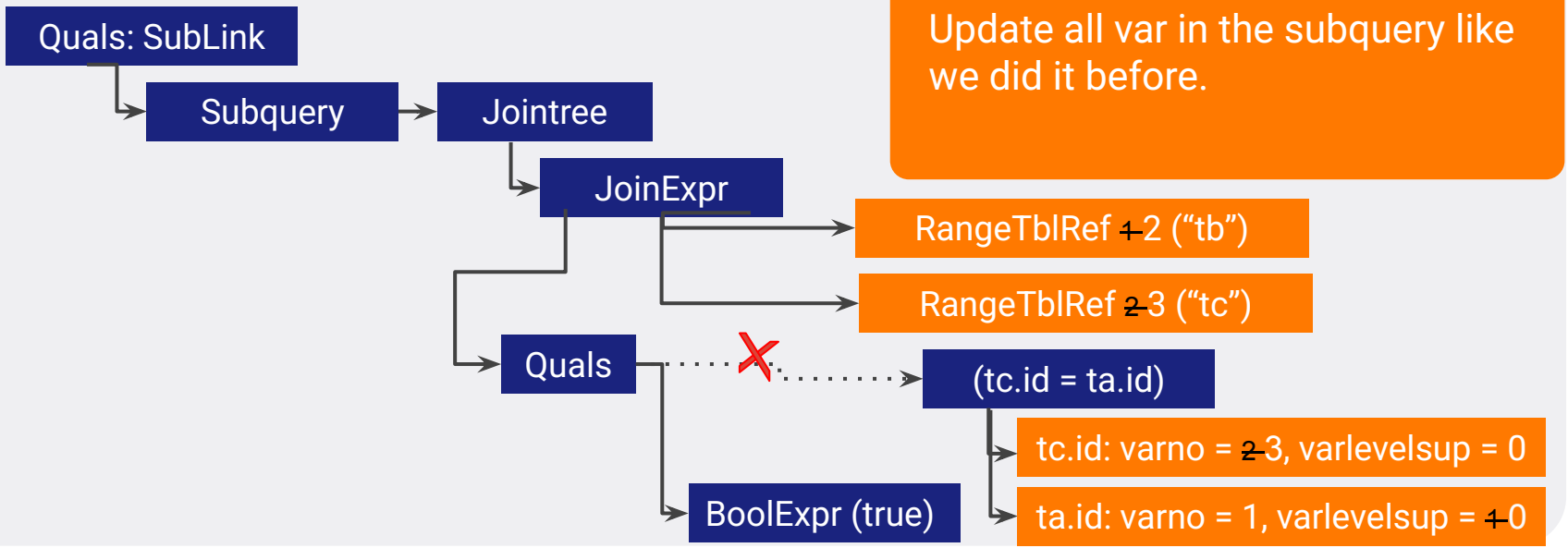
SubQuery Tree: Making Cartesian Product

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



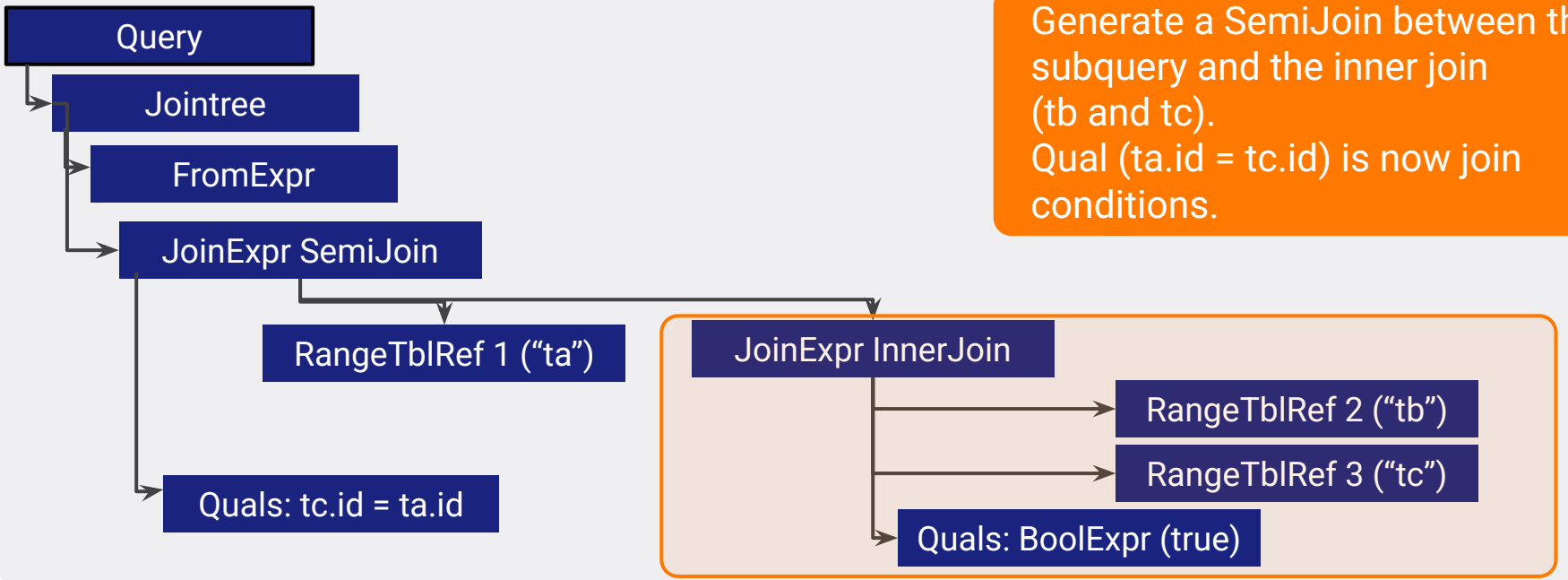
SubQuery Tree: Adjust Vars

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



Final Query Tree

select id from ta where exists (select aval from tb where tb.id = ta.id and tb.aval = 1)



Proposed Patch

Enables pull-up of EXISTS subqueries that contain INNER JOINs. Moves safe quals upward to enable join reordering and reduce redundant subquery scans.

Main development points:

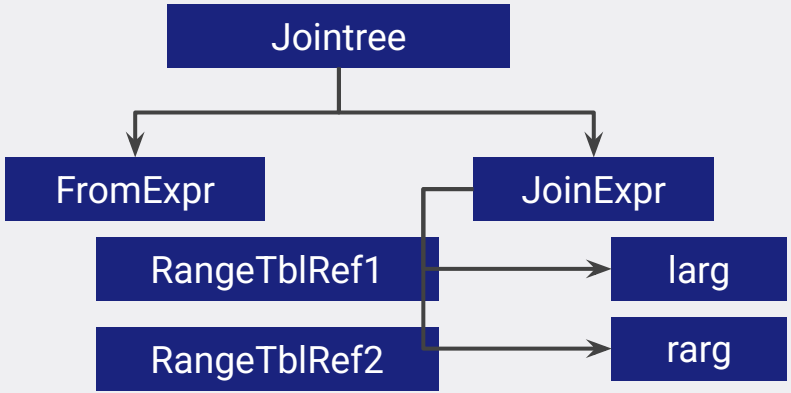
- Introduces a mutator that collects outer quals and replaces them with a BoolExpr(TRUE).
- Stops transformation if:
  - The subquery contains volatile quals;
  - The mutator encounters OUTER JOINs with outer references, since hoisting would break null-preserving behavior.

https://commitfest.postgresql.org/patch/5487/

How the Mutator Works

Recursively walks the subquery join tree, examining FROM and JOIN clauses in the subquery.

hoist_parent_quals_jointree_mutator()



- Gathers join quals with upper reference.
- Replaces affected join quals with TRUE.
- Only INNER JOINs are eligible for qual hoisting.
- If the join contains volatile functions (like random()), or if recursion fails, the transformation stops.
- Stops pull-up transformation if the mutator encounters OUTER JOINs with outer references, since hoisting would break null-preserving behavior.

Test Results on JOB Benchmark

Rewrote selected queries using EXISTS expressions. For example, query 3a.sql:

```
SELECT MIN(t.title) AS movie_title
FROM keyword AS k,
    movie_info AS mi,
    movie_keyword AS mk,
    title AS t
WHERE k.keyword LIKE '%sequel%'
  AND mi.info IN ('Sweden', 'Norway', ...)
  AND t.production_year > 2005
  AND t.id = mi.movie_id
  AND t.id = mk.movie_id
  AND mk.movie_id = mi.movie_id
  AND k.id = mk.keyword_id;
```

```
SELECT MIN(t.title) AS movie_title
FROM movie_info AS mi, movie_keyword AS mk,
title AS t
WHERE
  t.id = mi.movie_id AND t.id = mk.movie_id
  AND mk.movie_id = mi.movie_id
  AND exists (
  select 1 from movie_companies mc
  JOIN keyword AS k on k.id = mk.keyword_id
  WHERE
  mc.note IN ('Sweden',...)
  AND mc.movie_id = mi.movie_id
  AND k.keyword LIKE '%sequel%')
  AND t.production_year > 2005;
```

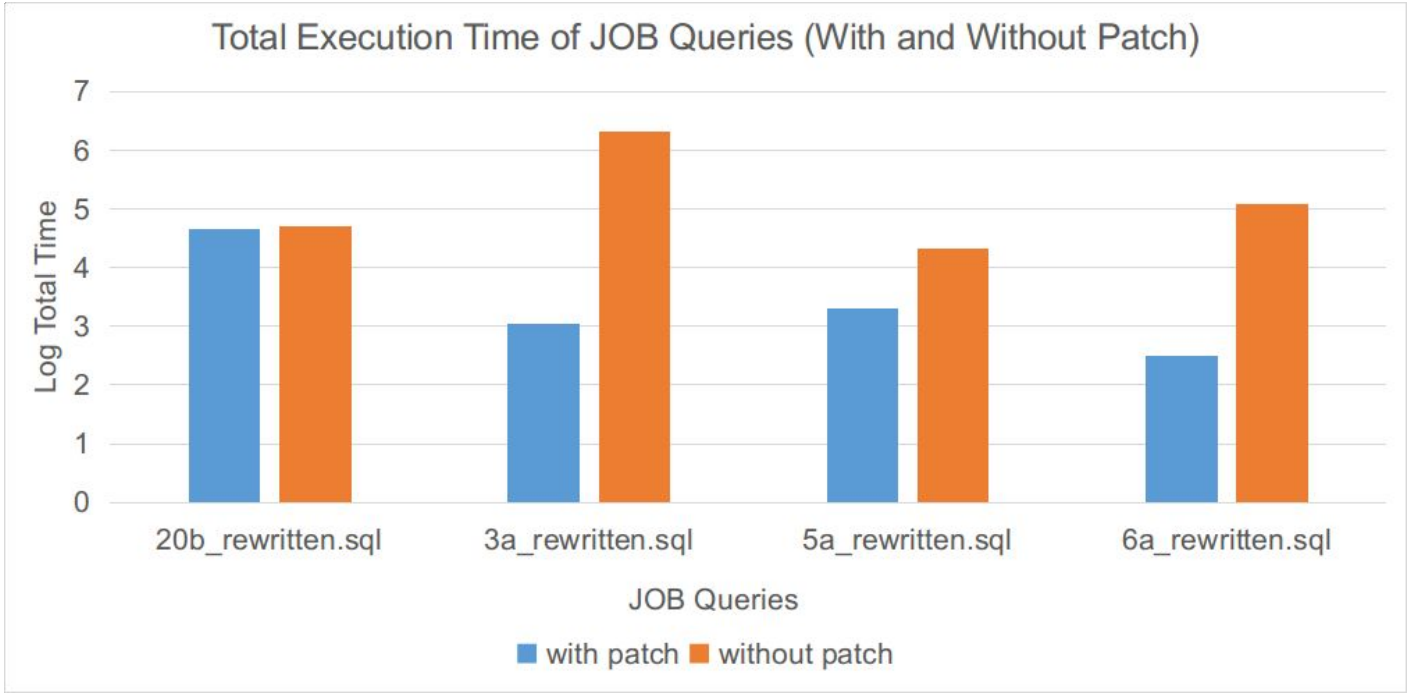
JOB Benchmark Testing Setup

- Branch: test-exists-pull-up
- Rewritten queries located in subselect_test directory
- Queries used for testing: 3a.sql, 5a.sql, 6a.sql, 20b.sql
- Detailed testing instructions provided in the README

https://github.com/Alena0704/jo-bench/tree/test-exists-pull-up

Performance Results on JOB Benchmark

The patch significantly reduces total runtime for queries rewritten with EXISTS sublinks.



One Problem Remaining: Outer Joins Inside Subqueries

Enables pull-up of EXISTS subqueries that contain INNER JOINs. Moves safe quals upward to enable join reordering and reduce redundant subquery scans.

Main development points:

- Introduces a mutator that collects outer quals and replaces them with a BoolExpr(TRUE).
- Stops transformation if:
  - The subquery contains volatile quals;
  - The mutator encounters OUTER JOINs with outer references, since hoisting would break null-preserving behavior.

https://commitfest.postgresql.org/patch/5487/

Let’s Consider it: Exists Sublink with Left Join

The join between tb and tc pairs their rows based on ta.id = tc.id.

For each row in ta, the subquery checks if at least one matching pair exists.

```
ta: id | val
----+-----
 1 |   1
 2 |   2
```

```
tb: id | aval
----+------
 1 |    1
```

```
tc: id | aid
----+-----
 2 |   2
```

```
SELECT * FROM ta WHERE EXISTS (
  SELECT 1 FROM tb LEFT JOIN tc
                          ON ta.id = tc.id);
```

```
id | val
----+-----
 1 |   1
```

Exists Subquery with Left Join

```
SELECT ta.* FROM ta WHERE EXISTS (
    SELECT 1 FROM tb LEFT JOIN tc
                                ON ta.id = tc.id);
```

```
ta: id | val        tb: id | aval      tc: id | aid
   ----+-----          ----+------        ----+-----
     1 |   1            1 |    1          2 |   2
     2 |   2
```

There is a Left Join between tb and tc tables, so we need to consider all tuples from table tb but return only matched with table ta.

```
SELECT tb.id as tb_id,
       tb.aval as tb_aval,
       tc.id as tc_id,
       tc.aid as ic_aid
FROM tb LEFT JOIN tc
ON tc.id in (select id from ta);
```

```
 tb_id | tb_aval | tc_id | ic_aid
--------+---------+-------+---------
     1 |       1 |     2 |       2
```

Exists Subquery with Left Join

```
SELECT ta.* FROM ta WHERE EXISTS (
    SELECT 1 FROM tb LEFT JOIN tc
                                ON ta.id = tc.id);
```

```
ta: id | val
    ----+-----
      1 |   1
      2 |   2
```

```
tb: id | aval
    ----+------
      1 |    1
```

```
tc: id | aid
    ----+-----
      2 |   2
```

The tc.id value is returned because it matches ta.id as well.

```
SELECT tb.id as tb_id,
       tb.aval as tb_aval,
       tc.id as tc_id,
       tc.aid as ic_aid
FROM tb LEFT JOIN tc
ON tc.id in (select id from ta);
```

```
 tb_id | tb_aval | tc_id | ic_aid
---------+-----------+--------+----------
      1 |         1 |      2 |        2
```

Exists Subquery with Left Join

```
SELECT ta.* FROM ta WHERE EXISTS (
    SELECT 1 FROM tb LEFT JOIN tc
                                ON ta.id = tc.id);
```

```
ta: id | val        tb: id | aval      tc: id | aid
    ----+-----          ----+------        ----+-----
      1 |   1            1 |    1          3 |   3
      2 |   2
```

Table tc contains values that don't match ta. When a tc value doesn't match ta, the LEFT JOIN still returns a row with nulls.

```
SELECT tb.id as tb_id,
       tb.aval as tb_aval,
       tc.id as tc_id,
       tc.aid as ic_aid
FROM tb LEFT JOIN tc
ON tc.id in (select id from ta);
```

```
 tb_id | tb_aval | tc_id | ic_aid
---------+-----------+--------+----------
      1 |         1 |        |
```

Exists Subquery with Left Join

```
SELECT ta.* FROM ta WHERE EXISTS (
    SELECT 1 FROM tb LEFT JOIN tc
                                ON ta.id = tc.id);
```

```
ta: id | val
----+-----
 1 |   1
 2 |   2

tb: id | aval
----+------
 3 |    3

tc: id | aid
----+-----
 3 |   3
```

Table tb contains values that don't match ta. The LEFT JOIN still produces a null-extended row - but that tb.id is unrelated to ta.id.

```
SELECT tb.id as tb_id,
       tb.aval as tb_aval,
       tc.id as tc_id,
       tc.aid as ic_aid
FROM tb LEFT JOIN tc
ON tc.id in (select id from ta);
```

```
tb_id | tb_aval | tc_id | ic_aid
--------+-----------+-------+---------
     3 |         3 |       |
```

Exists Subquery with Left Join

The EXISTS condition fails only if no tb.id matches ta.id, or if tb is empty.
tc values don't affect the result completely.

ta: id | val

```
----+-----
 1 |  1
 2 |  2
```

tb: id | aval

```
----+------
 3 |  3
```

tc: id | aid

```
----+-----
 3 |  3
```

```
SELECT * FROM ta WHERE EXISTS (
  SELECT 1 FROM tb LEFT JOIN tc
                          ON ta.id = tc.id);
```

```
id | val
----+-----
```

MySQL: Different Approach

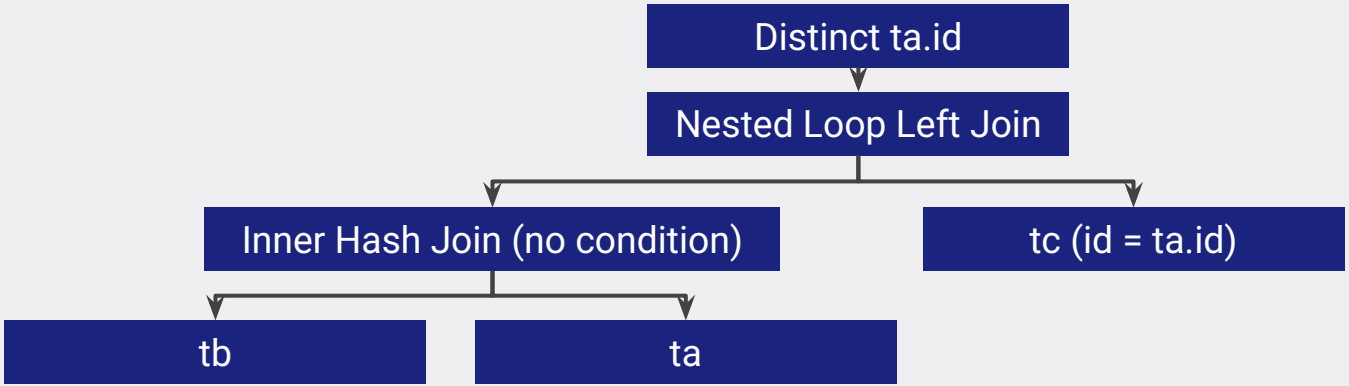
```
SELECT 1 FROM ta WHERE EXISTS (SELECT 1 FROM tb LEFT OUTER JOIN tc
                                                    ON ta.id = tc.id);
```

MySQL seems to pull the referenced table into the subquery, creating a Cartesian-like join between ta and tb and removing duplicate LEFT JOIN results for ta.id.

```
+-------------------------------------------------------------------------------------------------------------------
| -> Remove duplicate ta rows using temporary table (weedout) (rows=2 loops=1)
    -> Nested loop left join (rows=4 loops=1)
       -> Inner hash join (no condition) (rows=4 loops=1)
          -> Covering index scan on tb using PRIMARY  (rows=2 loops=1)
          -> Hash
             -> Table scan on ta (rows=2 loops=1)
       -> Single-row covering index lookup on tc using PRIMARY (id=ta.id)  (rows=0 loops=4)
 |
```

MySQL: Different Approach

```
SELECT 1 FROM ta WHERE EXISTS
                (SELECT 1 FROM tb LEFT OUTER JOIN tc
                                                    ON ta.id = tc.id);
```



Can Postgres Do the Same with Pull-up Optimization?

```
SELECT ta.* FROM ta WHERE EXISTS (
  SELECT 1 FROM tb LEFT JOIN tc
                              ON ta.id = tc.id);
```

```
-----------------------------------------
 Nested Loop Semi Join
   Disabled: true
   ->  Seq Scan on ta
   ->  Seq Scan on tb
```

With the LATERAL path:

- Builds a SEMI/ANTI JoinExpr with NULL quals, converting the subquery into a LATERAL RTE.
- Keeps all outer references within the pulled-up subquery.
- The planner can't generate hash or merge semijoin paths - only Nested Loops are used, scanning the inner per outer row.
- Prevents creation of equivalence classes.
- The planner can't reorder this input across others, which limits global join-order optimization.

Exploring Pull-down in Postgres

```
SELECT ta.* FROM ta WHERE EXISTS
                (SELECT 1 FROM tb LEFT JOIN tc ON ta.id = tc.id);
```

```
-----------------------------------------
 Unique ta.id
   ->  Nested Loop Left Join
         -> Nested Loop
              -> Seq Scan on tb
              -> Seq Scan on ta
         -> Seq Scan tc
              Filter: id = ta.id
```

Only one drawback - the invasive development and more tree walkers, but it looks like we don't have a choice

PGDU 2025

PostgresPro

Conclusion and Future Development

Conclusion And Further Development

- PostgreSQL already supports several forms of pull-up optimization that can eliminate redundant subquery execution and unlock higher-level planner optimizations, such as global join-order search.
- Pull-up optimizations apply only when the subquery is simple.
- The transformation is complex because it must update all elements tied to a subquery.
- Two current limitations remain for EXISTS sublinks:
  - Subqueries with INNER JOINs.
  - Subqueries with OUTER JOINs that reference outer variables.
- The proposed patch resolves the first case but leaves the second as an open problem.
- MySQL’s pull-down optimization could theoretically address this limitation — but its implementation would require highly invasive planner changes.

Feel Free to Review My Patch

Enables pull-up of EXISTS subqueries that contain INNER JOINs. Moves safe quals upward to enable join reordering and reduce redundant subquery scans.

Main development points:

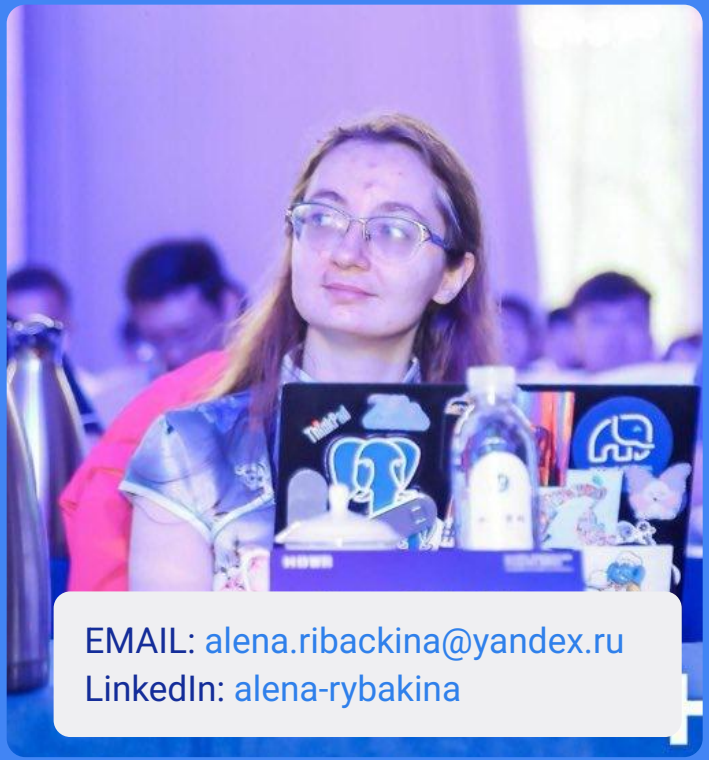
- Introduces a mutator that collects outer quals and replaces them with a BoolExpr(TRUE).
- Stops transformation if:
  - The subquery contains volatile quals;
  - The mutator encounters OUTER JOINs with outer references, since hoisting would break null-preserving behavior.

https://commitfest.postgresql.org/patch/5487/

PGDU 2025

PostgresPro

Thank You for Your Attention!



EMAIL: alena.ribackina@yandex.ru
LinkedIn: alena-rybakina