Implementing & Flattening Nested LATERAL joins in DuckDB

Sam Arch, Mayank Baranwal, Arham Chopra
Nested LATERAL joins for DuckDB

FLATERAL

Sam Arch, Mayank Baranwal, Arham Chopra
Outline

1) LATERAL joins and why we need them for UDFs
2) Project Goals
3) Live Demo
4) Approach
5) War Stories
6) Future Work & Takeaways
User-Defined Functions (UDFs)

CREATE OR REPLACE FUNCTION fib(x INT)
RETURNS INT AS
$$
DECLARE
  fib1 INT := 0;
  fib2 INT := 1;
  fib3 INT := 1;
BEGIN
  FOR i IN 1..x LOOP
    fib3 = fib1 + fib2;
    fib1 = fib2;
    fib2 = fib3;
  END LOOP;
  RETURN fib1;
END;
$$ LANGUAGE plpgsql;

SELECT fib(42);
UDFs are Slow

No UDF: Fast
UDF: Slow

Up to 10,000x slower
CREATE OR REPLACE FUNCTION fib(x INT) RETURNS INT AS
$$
DECLARE
  fib1 INT := 0;
  fib2 INT := 1;
  fib3 INT := 1;
BEGIN
  FOR i IN 1..x LOOP
    fib3 = fib1 + fib2;
    fib1 = fib2;
    fib2 = fib3;
  END LOOP;
RETURN fib1;
END;
$$ LANGUAGE plpgsql;
SELECT fib(42);

WITH RECURSIVE run("rec?", "res", "x", "fib1", "fib2", "i") AS
(
  (SELECT True, NULL :: int4, "x", "fib1_1", "fib2_1", "i_1"
  FROM LATERAL (SELECT 0 AS "fib1_1" AS "let0"("fib1_1"),
  LATERAL (SELECT 1 AS "fib2_1" AS "let1"("fib2_1"),
  LATERAL (SELECT 1 AS "i_1" AS "let2"("i_1")))
  UNION ALL
  (SELECT "result".*
  FROM run AS "run"("rec?", "res", "x", "fib1", "fib2", "i"),
  LATERAL (SELECT "ifresult6".*
  FROM LATERAL (SELECT "x" AS "q2_2" AS "let4"("q2_2"),
  LATERAL (SELECT "i*" AS "q2_2" AS "pred3_2" AS "let5"("pred3_2"),
  LATERAL ((SELECT True, NULL :: int4, "x", "fib1_4", "fib2_4", "i_4"
  FROM LATERAL (SELECT "fib1" + "fib2" AS "fib3_4" AS "let7"("fib3_4"),
  LATERAL (SELECT "fib2" AS "fib1_4" AS "let8"("fib1_4"),
  LATERAL (SELECT "fib3_4" AS "fib2_4" AS "let9"("fib2_4"),
  LATERAL (SELECT "i*" + 1 AS "i_4" AS "let10"("i_4")
  WHERE NOT "pred3_2" IS DISTINCT FROM True)
  UNION ALL
  (SELECT False,
  "fib1" AS "result",
  "run"."x",
  "run"."fib1",
  "run"."fib2",
  "run"."i"
  WHERE "pred3_2" IS DISTINCT FROM True)
  ) AS "ifresult6"
  ) AS "result"
  ) AS "run".
  WHERE NOT "run"."rec?"
  )
SELECT "run"."res" AS "res"
FROM run AS "run"
WHERE NOT "run"."rec?"
LATERAL Joins

WITH RECURSIVE run("rec?", "res", "x", "fib1", "fib2", "i") AS 
(
    (SELECT True, NULL :: int4, "x", "fib1_1", "fib2_1", "i_1"
    FROM LATERAL (SELECT 0 AS "fib1_1" AS "let0"("fib1_1"),
    LATERAL (SELECT 1 AS "fib2_1" AS "let1"("fib2_1"),
    LATERAL (SELECT 1 AS "i_1" AS "let2"("i_1"))
    UNION ALL
    (SELECT "result".*
    FROM run AS "run"("rec?", "res", "x", "fib1", "fib2", "i"),
    LATERAL
    (SELECT "ifresult6".*
    FROM LATERAL (SELECT "x" AS "q2_2" AS "let4"("q2_2"),
    LATERAL (SELECT "i" <= "q2_2" AS "pred3_2" AS "let5"("pred3_2"),
    LATERAL
    (SELECT True, NULL :: int4, "x", "fib4", "fib2_4", "i_4"
    FROM LATERAL
    (SELECT "fib1" + "fib2" AS "fib3_4" AS "let7"("fib3_4"),
    LATERAL (SELECT "fib2" AS "fib4_4" AS "let8"("fib4_4"),
    LATERAL (SELECT "fib3_4" AS "fib2_4" AS "let9"("fib2_4"),
    LATERAL (SELECT "i" + 1 AS "i_4" AS "let10"("i_4"))
    WHERE NOT "pred3_2" IS DISTINCT FROM True)
    UNION ALL
    (SELECT False,
    "fib1" AS "result",
    "run"."x",
    "run"."fib1",
    "run"."fib2",
    "run"."i"
    WHERE "pred3_2" IS DISTINCT FROM True)
    ) AS "ifresult6"
    ) AS "result"
    WHERE "run"."rec?"
    )
    )
SELECT "run"."res" AS "res"
FROM run AS "run"
WHERE NOT "run"."rec?"
Why DuckDB?

WITH RECURSIVE run("rec?", "res", "x", "fib1", "fib2", "i") AS
(
    (SELECT True, NULL :: int4, "x", "fib1_1", "fib2_1", "i_1"
    FROM LATERAL (SELECT 0 AS "fib1_1" AS "let0"("fib1_1"),
      LATERAL (SELECT 1 AS "fib2_1" AS "let1"("fib2_1"),
      LATERAL (SELECT 1 AS "i_1" AS "let2"("i_1"))
    UNION ALL
    (SELECT "result".*
    FROM run AS "run"("rec?", "res", "x", "fib1", "fib2", "i"),
      LATERAL
      (SELECT "ifresult6".*
      FROM LATERAL (SELECT "x" AS "q2_2" AS "let4"("q2_2"),
        LATERAL (SELECT "i" <= "q2_2" AS "pred3_2" AS "let5"("pred3_2"),
        LATERAL
        ((SELECT True, NULL :: int4, "x", "fib1_4", "fib2_4", "i_4"
        FROM LATERAL
        (SELECT "fib1" + "fib2" AS "fib3_4" AS "let7"("fib3_4"),
          LATERAL (SELECT "fib2" AS "fib1_4" AS "let8"("fib1_4"),
          LATERAL (SELECT "fib3_4" AS "fib2_4" AS "let9"("fib2_4"),
          LATERAL (SELECT "i" + 1 AS "i_4" AS "let10"("i_4")
        WHERE NOT "pred3_2" IS DISTINCT FROM True)
    UNION ALL
    (SELECT False, "fib1" AS "result",
      "run"."x",
      "run"."fib1",
      "run"."fib2",
      "run"."i"
      WHERE NOT "pred3_2" IS DISTINCT FROM True)
    ) AS "ifresult6"
  ) AS "result"
  WHERE "run"."rec?"
)
)
SELECT "run"."res" AS "res"
FROM run AS "run"
WHERE NOT "run"."rec?"
Project Goals

Implement & flatten nested LATERAL joins

75% - Fixed depth LATERALs (i.e. depth = 2)

100% - Arbitrary depth LATERALs

125% - Full UDF support (Recursive CTEs + LATERALs)
Project Goals

Implement & flatten nested LATERAL joins

75% - Fixed depth LATERALs (i.e. depth = 2)
100% - Arbitrary depth LATERALs
125%* - Full UDF support (Recursive CTEs + LATERALs)

* T&C Apply (explained later)
Live Demo
**Approach**

```sql
SELECT *
FROM
  (SELECT 1) t1(i),
LATERAL (SELECT *
FROM
  (SELECT 2) t2(j),
LATERAL (SELECT i + j) t3(k));
```

---

**Original Codebase**

- **Binder:** Not able to handle correlations in nested laterals
- **Planner:** Plans nested laterals in an incorrect way, causing failures
- **Assumptions in the code on only one level of laterals**
Approach

• Binder:
  ○ Bind laterals recursively
  ○ Track correlations correctly
  ○ Maintain correct depth information
Approach

- **Planner:**
  - Change planning order: top-down
  - Detect correlations correctly
  - Push-down correlations through all the relevant nodes
  - Rewrite column bindings
# Validation

<table>
<thead>
<tr>
<th>Tests</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our Stress Tests (verified against UMBRA)</td>
<td>✔</td>
</tr>
<tr>
<td>PostgreSQL tests for LATERALs and Subqueries</td>
<td>✔</td>
</tr>
<tr>
<td>DuckDB Tests for LATERALs and Subqueries</td>
<td>✔</td>
</tr>
<tr>
<td>DuckDB Regression Tests (for PR)</td>
<td>✔</td>
</tr>
<tr>
<td>DuckDB CI Tests (for PR)</td>
<td>✔</td>
</tr>
<tr>
<td>SQLite Tests for LATERALs (the holy grail)</td>
<td>Don't Exist 🤦‍♂️</td>
</tr>
</tbody>
</table>

Note: Hatching symbol for unresolved issues.
Now for the War Stories

A tale of misery, suffering and joy
Let's do flattening they said
It will be easy they said
...
- Mark Raasveldt
(maybe not really)
DuckDB has one of the most beautiful C++ codebases

The Prologue

Sam

Arham

Mayank

OK!
Reading the Codebase

Especially with close to NO COMMENTS
After weeks of procrastinating and struggling with the code

And after multiple failed attempts
Mark when he tells us "Godspeed"
Debugging Binder Code
Debugging Binder Code

And

Logical

Planner

Code
Debugging Binder Code

And Logical Planner Code

And Physical Planner Code
When 3 million tests pass
But then 2 tests fail
But then 2 tests fail 2 days before the deadline
Revamping the whole logic
Then all tests pass

It’s done.
Future Work

- Put in a Pull Request to DuckDB (refactoring required)
- Benchmark the UDFs in the ProcBench on DuckDB
- Compare DuckDB and Umbra on the ProcBench
- Convince the world to use UDFs 😈
  
  ... So that Sam can get his PhD :)
Takeaways

• The Binder/Rewriter is extremely tricky
  ○ But when it makes sense, it just works

• Extensive Testing & Rapid Prototyping go very far

• Drawing diagrams & (Constructive?) Arguing = Progress

• Real database development is HARD

• This is why DB companies pay us the big bucks
Questions?