DuckPL: A Procedural Language in DuckDB

Bringing PL/pgSQL to DuckDB

30th of January 2026 DuckDB Developer Meeting #1

Denis Hirn

✉ denis.hirn@uni-tuebingen.de
University of Tübingen
kryonix

Denis Hirn? kryonix?

Long-time DuckDB contributor since early 2020:

- Recursive/Materialized CTEs
- Some re-architecting of query decorrelation
- Various bug fixes and optimizations

Research focus:

- Database systems
  - query optimization,
  - execution engine design, and
  - User-Defined Function optimization, aka. **how to get rid of them**
- Compilers and programming languages
- Bridging the gap between both fields

State of DuckDB UDFs

Supported ✅

```sql
CREATE MACRO add(a, b) AS a + b;
```

Not Supported ❌

```sql
CREATE MACRO sequence(n) AS
  IF n < 0 THEN
    do some stuff
  ELSE
    do other stuff
  END IF;
  do some more
  CREATE TABLE something AS (column type);
  RETURN something;
```

`MACRO`s:

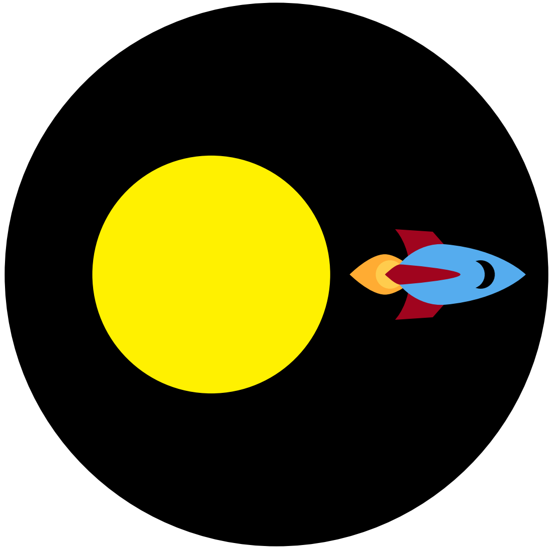
- simple textual replacements
- no full-fledged procedural language for writing user-defined functions (UDFs) yet

`Python`, `R`, and other languages can be used to write UDFs, but:

- Require an external runtime
- Breaks the **“Single-File, Zero-Dependency Database”** promise

Introducing DuckPL!

Procedural **`PL/SQL`**, Native to DuckDB.



Procedural PL/SQL for DuckDB!

Procedural Logic (DuckPL)

```sql
CREATE FUNCTION collatz(y BIGINT) RETURNS BIGINT AS $$
DECLARE
  steps BIGINT := 0;
  x BIGINT := y;
BEGIN
  WHILE x > 1 LOOP
    IF x % 2 = 0 THEN
      x := x / 2;
    ELSE
      x := 3 * x + 1;
    END IF;
    steps := steps + 1;
  END LOOP;
  RETURN steps;
END;
$$;
SELECT collatz(5);
```

🥳 *Easy!*

Recursive CTE (Pure SQL)

```sql
SELECT
  (WITH RECURSIVE collatz_cte(x, steps) AS (
      SELECT 5 AS x, 0 AS steps
        UNION ALL
      SELECT
        CASE WHEN x % 2 = 0
              THEN x / 2
              ELSE 3 * x + 1
        END AS x,
        steps + 1 AS steps
      FROM collatz_cte
      WHERE x > 1
    )
    SELECT steps
    FROM collatz_cte
    WHERE x = 1
  ) AS collatz;
```

😰 *Not so easy*…*

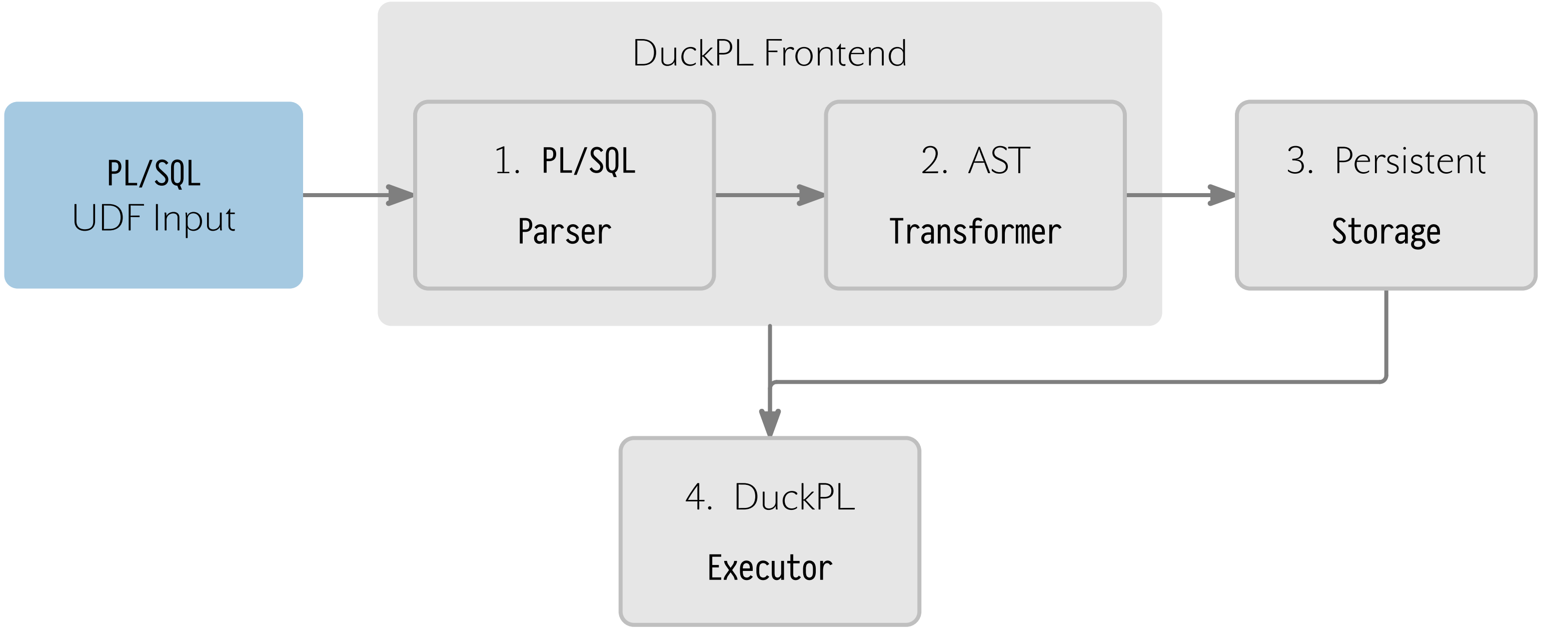
*For average users

No external runtimes required Imperative programming right inside the system

PL/pgSQL compatibility Migration of existing codebases to DuckDB made ***easy***

Demo Time!

Implementing DuckPL: Key Components & Architecture



Parser Extension Custom parser for `CREATE FUNCTION` statements
Operator Extension Custom `Bind` and `Plan` for `CREATE` operations

Parsing: The Missing Pieces

```
CREATE FUNCTION collatz(y BIGINT)
RETURNS BIGINT
AS $$
DECLARE
  steps BIGINT := 0;
  x BIGINT := y;
BEGIN
  WHILE x > 1 LOOP
    IF x % 2 = 0 THEN
      x := x / 2;
    ELSE
      x := 3 * x + 1;
    END IF;
    steps := steps + 1;
  END LOOP;
  RETURN steps;
END;
$$;
```

1. **Incomplete `CREATE FUNCTION` parsing support**: DuckDB's `SQL` parser lacks grammar rules to parse `AS $$ ... $$` functions.
2. **No Language Parser**: The `PL/pgSQL` body is a generic string literal and must be parsed separately.

😨

Consequence We need to parse **both** the `CREATE FUNCTION` statement **and** the `PL/pgSQL` function body.

PL/pgSQL Parsing: The libpg_query Approach

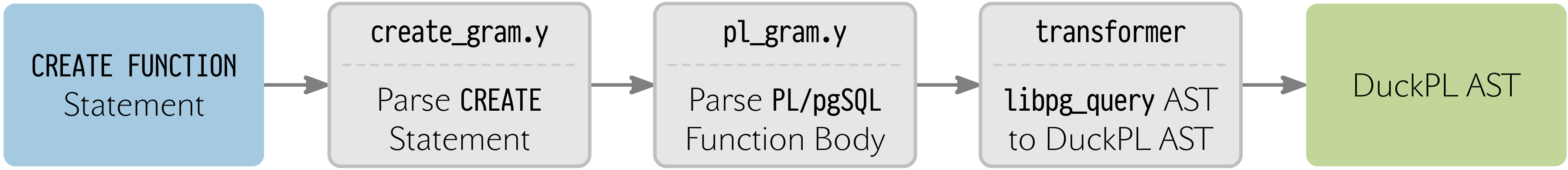
Phase 1: Parsing

- DuckPL needs **two** parsers:
  1. for the CREATE FUNCTION statement
  2. for the PL/pgSQL function body.
- We can reuse the existing libpg_query parser for both!
  - leverage missing pieces for CREATE FUNCTION parsing
  - reuse the existing PL/pgSQL parser as-is

Phase 2: AST Transformation

- Implement transformer from libpg_query AST to DuckPL AST

This mirrors **exactly** how DuckDB's original SQL parser was built in 2018!



CREATE FUNCTION Statement
create_gram.y
Parse CREATE Statement
pl_gram.y
Parse PL/pgSQL Function Body
transformer
libpg_query AST to DuckPL AST
DuckPL AST

Soon™: Rip this apart and use a PEG-based parser!

Interlude: DuckPL AST — The Universal Internal Representation

```
duckpl
├── src
│   ├── include
│   │   ├── ast
│   │   │   ├── expressions
│   │   │   │   ├── pl_fetch_cursor.hpp
│   │   │   │   └── pl_inline_sql.hpp
│   │   │   ├── statements
│   │   │   │   ├── pl_assignment.hpp
│   │   │   │   ├── pl_block.hpp
│   │   │   │   ├── pl_break.hpp
│   │   │   │   ├── pl_declare.hpp
│   │   │   │   ├── pl_emit.hpp
│   │   │   │   ├── pl_execute.hpp
│   │   │   │   ├── pl_if.hpp
│   │   │   │   ├── pl_loop.hpp
│   │   │   │   ├── pl_open.hpp
│   │   │   │   ├── pl_print.hpp
│   │   │   │   └── pl_stop.hpp
│   │   │   ├── pl_cursor.hpp
│   │   │   ├── pl_expression.hpp
│   │   │   └── pl_statement.hpp
│   │   ├── parser
│   │   └── storage
│   └── parser
└── third_party
```

DuckPL IR: **minimalist** and **syntax-agnostic** to support multiple source languages.

- Complex constructs like `FOR`/`WHILE` loops desugar to `LOOP` + `IF` + `BREAK`
  - Eliminates `FOR`, `WHILE`, `CURSOR`, `ARRAY` loops, *etc.*
- No `CASE` statements: Everything simplifies to an `IF` statement.

Source Language (`PL/pgSQL`)

```
WHILE counter < 10 LOOP
    counter := counter + 1;
END LOOP;
RETURN counter;
```

↦

Internal DuckPL IR AST

```
loop {
    if (counter >= 10) { break; }
    let counter = counter + 1;
}
emit counter;
stop;
```

Simplifies Interpreter Reduces complexity of control flow handling

Future Language Support A new procedural language (`PL/Python`, `PL/Duck`) requires just **Transformer** ↦ DuckPL AST

Simplifies Compilation Easier to compile DuckPL AST to `SQL`

Storage: Persisting and Registering DuckPL UDFs

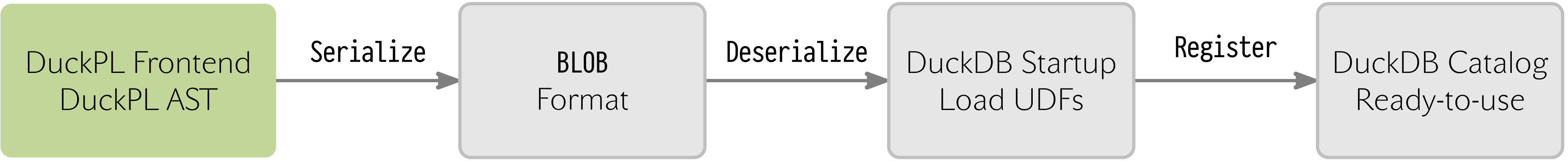
Persistent Storage

- UDFs are stored in **duckpl_functions** table.
- AST is serialized and stored as a **BLOB**.
- Avoids unnecessary re-parsing.
- Similar to DuckLake macros.

Registration on Startup

- Load stored UDFs from **duckpl_functions**.
- Deserialize AST and register in the catalog.
- UDFs are immediately available without parsing.

```sql
CREATE TABLE duckpl_functions (
  function_id BIGINT PRIMARY KEY,
  function_uuid UUID,
  function_num_args INT,
  function_arg_names TEXT[],
  function_arg_types TEXT[],
  function_return_types TEXT[],
  function_returns_set BOOLEAN,
  function_name TEXT NOT NULL,
  function_src TEXT,
  function_body BLOB);
```



Execution: The Stack-Driven Interpreter

Simple ***tree-walk*** interpreter, but avoids recursive calls by using an **explicit stack of frames**:

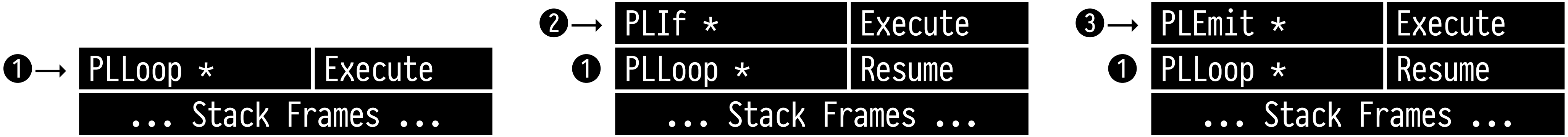
State Management Execution can be **paused** and **resumed** at any point

No C++ recursion No stack depth limits, no risk of stack overflows

```
❶→ LOOP {
       IF NOT x > 1 THEN
         BREAK;
       END IF;
       IF x % 2 = 0 THEN
       ...
     }
...
```

```
❶   LOOP {
❷→     IF NOT x > 1 THEN
         BREAK;
       END IF;
       RETURN NEXT x;
       ...
     }
...
```

```
❶   LOOP {
       IF NOT x > 1 THEN
         BREAK;
       END IF;
❸→     RETURN NEXT x;
       ...
     }
...
```



This design allows DuckPL to **stream** results efficiently without buffering everything in memory.

Execution: Streaming Output Like an Operator

```
CREATE FUNCTION infinite()
RETURNS SETOF BIGINT AS $$
DECLARE
  i BIGINT := 0;
BEGIN
  LOOP
    i := (i + 1) % 1000;
    RETURN NEXT i;
  END LOOP;
END
$$;

-- Create 10 DataChunks:
SELECT *
FROM infinite()
LIMIT 10 * 2048;
```

PostgreSQL: Buffers **all results** before returning **anything**

- Will never return anything from **`infinite()`** function
- Leads to memory ballooning 💣
- Cannot be interrupted (*e.g.*, via `LIMIT`)

💥

DuckPL's interpreter is **Fully streaming**:

Memory Efficiency No unnecessary buffering of results

- `RETURN NEXT` statements write result into output chunk

Interruptibility When chunk is full, return `OperatorResultType::HAVE_MORE_OUTPUT`

- Interpreter **pauses** and **resumes** when requested:
  Enabled by explicit stack of frames design
- Allows **early termination** (*e.g.*, via `LIMIT`)

Architectural Fit Follows same idea as physical operators in DuckDB 😊

Execution: Expression Fast-Path

```sql
CREATE FUNCTION collatz(y BIGINT)
RETURNS BIGINT AS $$
DECLARE
  steps BIGINT := 0;
  x BIGINT := y;
BEGIN
  WHILE x > 1 LOOP
    IF x % 2 = 0 THEN
      x := x / 2;
    ELSE
      x := 3 * x + 1;
    END IF;
    steps := steps + 1;
  END LOOP;
  RETURN steps;
END;
$$;
SELECT collatz(5);
```

The Slow Way:

Method Wrapping every expression in a `SELECT <...>` statement

Bottleneck Triggers the **Full SQL Pipeline** (Binding, Optimization, Execution) for ***every single expression*** 😵
This becomes ***super slow*** 🐌 without optimization!

The Fast Path:

💡 Use `ExpressionExecutor` for simple expressions

Not supported.. But how we do it anyway 😉:

1. **Prepare** a dummy `SELECT x > 1` statement
2. **Extract** the expression `x > 1` from prepared statement
3. **Cache** an `ExpressionExecutor` instantiated with `x > 1`
4. **Execute** against a `DataChunk` containing local variables

Result: 🎉 It's *fast* 💨🐌 now (we've seen speedups of 30×).

* This is vastly simplified; We have to do a lot more work to prepare the expression properly to make it cacheable.

DuckPL Feature Support

Supported ✅

- Scalar/Table-valued UDFs
- Variables and Assignments
- All Data Types
- Composite Types types like `lineitem`
- Control Flow
  - `IF`
  - `LOOP`, `WHILE`, and `FOR` loops
  - `BREAK` and `CONTINUE`
  - `RETURN` and `RETURN NEXT`
- **Cursors** (`FETCH INTO`)
- Debugging (`RAISE INFO`)

Planned 🚧

- Aggregate/Window UDFs
- Exception handling
- Transactions `COMMIT`, `ROLLBACK`
- **UDF Optimizer**

Compilation to Pure SQL

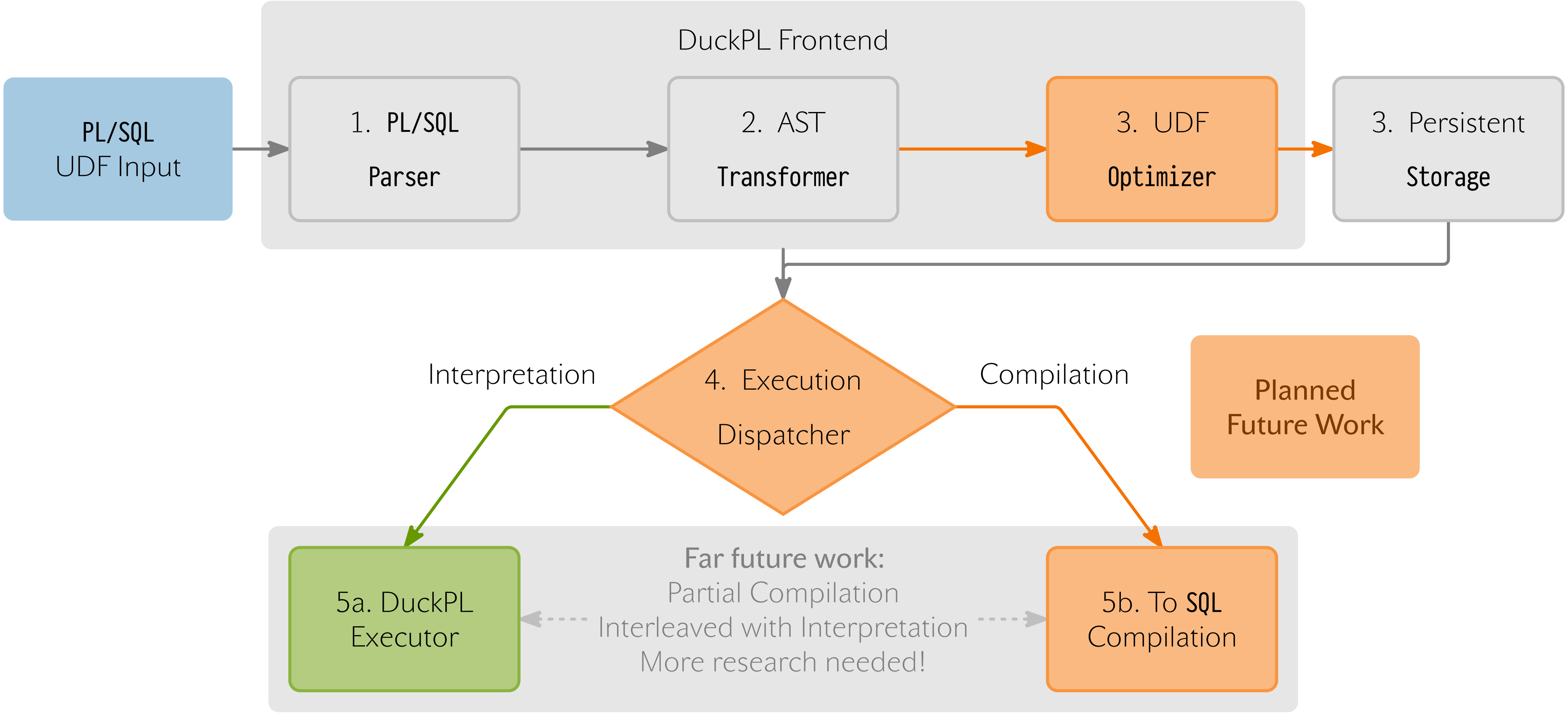
- Massively improve performance
- Leverage DuckDB's execution engine
- Use `WITH RECURSIVE` for complex control flow

Not for now or never ❌*

- Dynamic `SQL` (Use `query(...)`)
- Advanced cursor features like `SCROLL` and `MOVE`
- Triggers

* PRs welcome! 🚀 As soon as DuckPL becomes open source.

DuckPL Architecture: The Path to Hybrid Execution



Future Work — The Vision for DuckPL

Interactive Mode Allow DuckPL statements directly in the `CLI`, for a `REPL`-like experience:

```
❯ duckdb
D LET y = 0 :: BIGINT;
D FOR i IN 1..10:
    LET x = (SELECT RANDOM());
    IF x > 0.5:
        LET y = y + 1;
D PRINT y;
5
D █
```

Modern Syntax Follow **friendly `SQL`** idea for PL syntax

- Add a secondary, lightweight syntax
- Add `PL/Python` frontend

Next-Gen Parser Move to PEG based **`PL/pegSQL`** for better DuckDB integration

Compiled Execution Integrate our **UDF compilation** research into DuckPL

Production Readiness Improve **error messages** and **debugging support**

Vectorized Interpretation Implement **vectorized interpretation**

Advanced Features Support **table-valued variables**

There is **so much** more to do!

Conclusion

Compatibility First

Bring `PL/pgSQL` functionality to the DuckDB ecosystem.

The Win:

- Compatibility layer for existing `PL/pgSQL` codebases.
- Minimal learning curve for Postgres users.
- Works with existing tools and scripts immediately.

Smart Execution

Built using **tried-and-tested** techniques from DuckDB's history.

The Win:

- Stack-driven interpretation: enables streaming (no memory ballooning).
- No external runtimes.
- Ship your database ↔ ship your code. No dependencies.

The Vision

Designed with advanced optimization techniques in mind:
Apply **Automatic UDF Compilation and Inlining*** research.

The Win:

Native Speed Massive improvements through compilation to `SQL`.

Hybrid Execution Interleaved interpretation for best performance and full feature set.

* Which I extensively worked on during my PhD—so I'm biased 😉

🚀 DuckPL will be open-sourced soon!

DuckPL: A Procedural Language in DuckDB

Bringing PL/pgSQL to DuckDB

30th of January 2026 DuckDB Developer Meeting #1

Denis Hirn

✉ denis.hirn@uni-tuebingen.de
University of Tübingen
kryonix