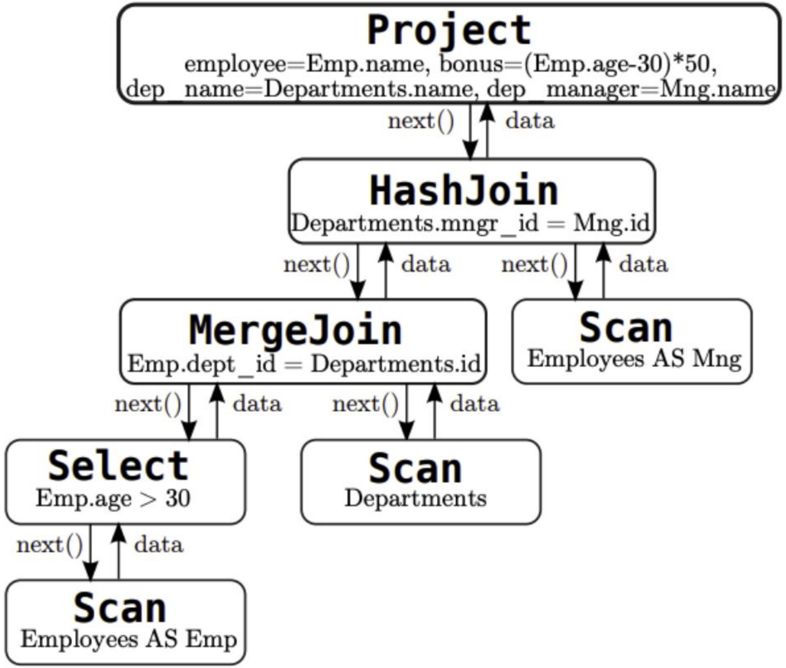
Exploring Just-in-Time Compilation in Relational Database Engines

Table of Contents

Project Recap



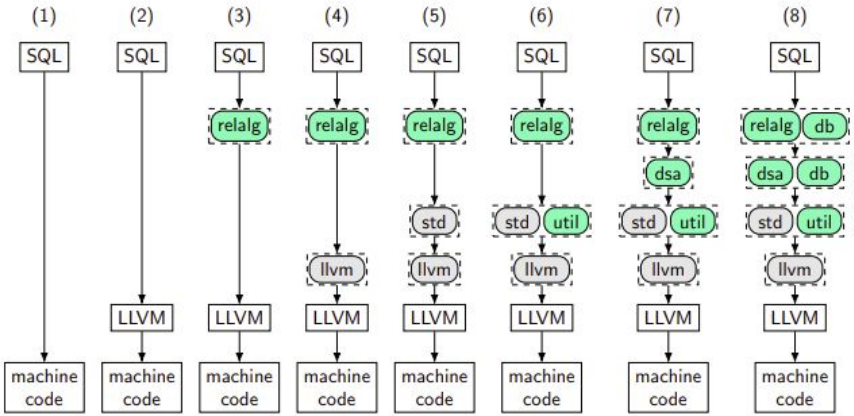
- PostgreSQL is the most popular relational database in the world
- It uses a volcano model for execution
- In theory, changing this to a compiler can make queries 2x faster

Project Recap

- There's two main choices for a compiler; just in time and ahead of time
- Since compiling is part of our latency, JIT is preferred in this context.
- There's a couple of libraries to do that, and we chose MLIR; multi level intermediate representation
- This was justified in part A

Project Recap

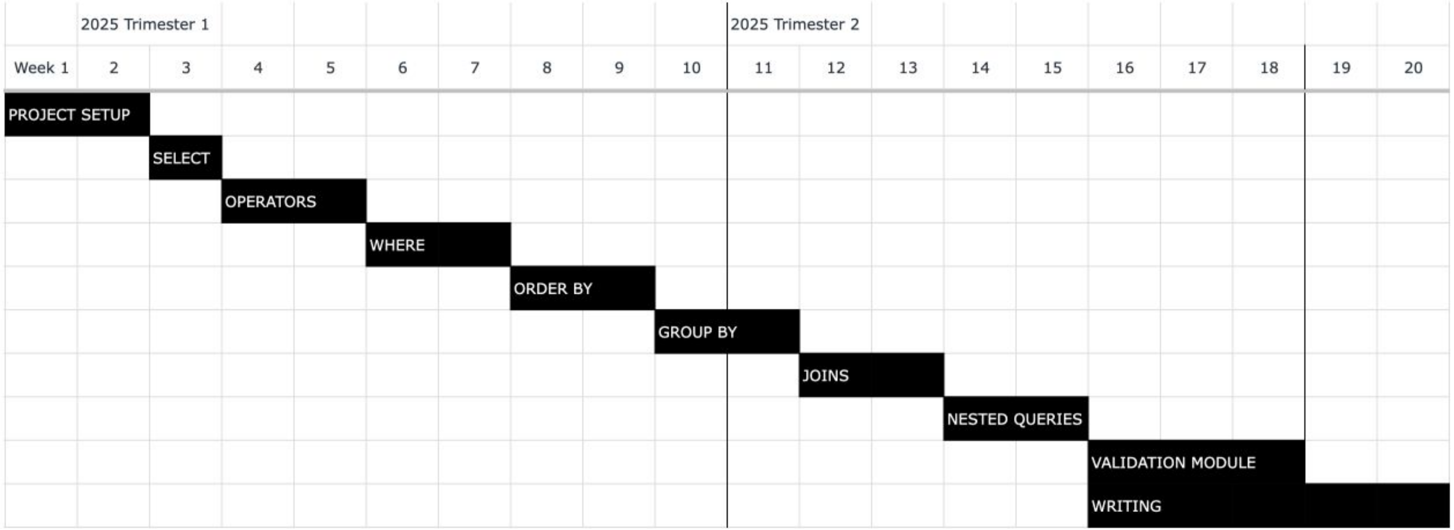
- MLIR breaks compilation into dialect layers and “lowers it”
- On the right is a diagram of LingoDB’s lowerings
- Essentially, we want to take this then apply it to PostgreSQL



https://www.lingo-db.com/

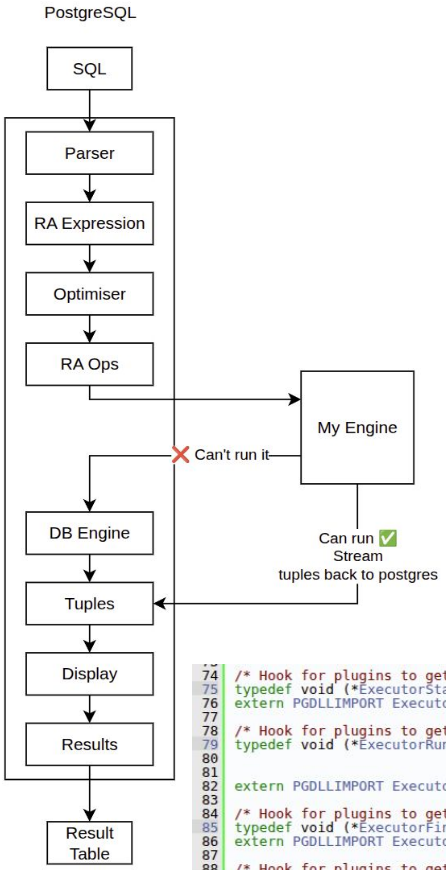
Project Recap

- This is our original project timeline



Project Setup

- How do we even connect to Postgres?
- My two main options are making a fork of postgres, or an extension
- After finding the executor.h, I found that there’s some runtime hooks that I can take over
- I can redirect this towards my own module
- Hooks are here: https://doxygen.postgresql.org/executor_8h_source.html




```c
/* Hook for plugins to get control in ExecutorStart() */
typedef void (*ExecutorStart_hook_type) (QueryDesc *queryDesc, int eflags);
extern PGDLLIMPORT ExecutorStart_hook_type ExecutorStart_hook;

/* Hook for plugins to get control in ExecutorRun() */
typedef void (*ExecutorRun_hook_type) (QueryDesc *queryDesc,
                                       ScanDirection direction,
                                       uint64 count);
extern PGDLLIMPORT ExecutorRun_hook_type ExecutorRun_hook;

/* Hook for plugins to get control in ExecutorFinish() */
typedef void (*ExecutorFinish_hook_type) (QueryDesc *queryDesc);
extern PGDLLIMPORT ExecutorFinish_hook_type ExecutorFinish_hook;

/* Hook for plugins to get control in ExecutorEnd() */
typedef void (*ExecutorEnd_hook_type) (QueryDesc *queryDesc);
extern PGDLLIMPORT ExecutorEnd_hook_type ExecutorEnd_hook;
```

Project Setup

```
add_postgresql_mixed_extension(
        pgx_lower
        VERSION 1.0
        C_SOURCES
            ../src/execution/postgres/executor_c.c
        CPP_SOURCES
            ../src/execution/postgres/my_executor.cpp
            ../src/execution/postgres/executor_c.cpp
```

```
bool try_cpp_executor_direct(const QueryDesc* queryDesc) {
    try {
        // Create an instance of MyCppExecutor and call execute
        MyCppExecutor executor;
        return executor.execute(queryDesc);
    } catch (const std::exception& ex) {
        PGX_ERROR("C++ exception: " + std::string(s: ex.what()));
        log_cpp_backtrace();
        return false;
    } catch (...) {
        PGX_ERROR("Unknown C++ exception occurred!");
        log_cpp_backtrace();
        return false;
    }
}

PG_FUNCTION_INFO_V1(try_cpp_executor);
Datum try_cpp_executor(PG_FUNCTION_ARGS) {
    const auto queryDesc = reinterpret_cast<QueryDesc*>(PG_GETARG_POINTER(0));
    const bool result = try_cpp_executor_direct(queryDesc);
    PG_RETURN_BOOL(result);
}
```

- Now that we know how to connect up, we need to go do it
- Also we need to go install MLIR
- This took me longer than I'd like to admit… at this point I swapped my main computer into Ubuntu 25.04
- I ended up using this as a project template: https://github.com/cppgres/cppgres
- But my project structure quickly changed since I only have one extension and I need MLIR

Project Setup

```
tests
  CMakeFiles
  expected
  sql
    1_one_tuple.sql
    2_two_tuples.sql
    3_lots_of_tuples.sql
    4_two_columns_ints.sql
    5_two_columns_diff.sql
    6_every_type.sql
    7_sub_select.sql
    8_subset_all_types.sql
    8_subset_all_types_simple.sql
    9_basic_arithmetic_ops.sql
    10_comparison_ops.sql
    11_logical_ops.sql
    12_null_handling.sql
    13_text_operations.sql
    14_aggregate_functions.sql
    15_special_operators.sql
    16_debug_text.sql
    17_where_simple_conditions.sql
```

```sql
LOAD 'pgx_lower.so';
SET client_min_messages TO NOTICE;
SELECT 'hello';
DROP TABLE IF EXISTS test;
CREATE TABLE test(id SERIAL);
INSERT INTO test(id) VALUES ( id 42);
SELECT * FROM test;
```

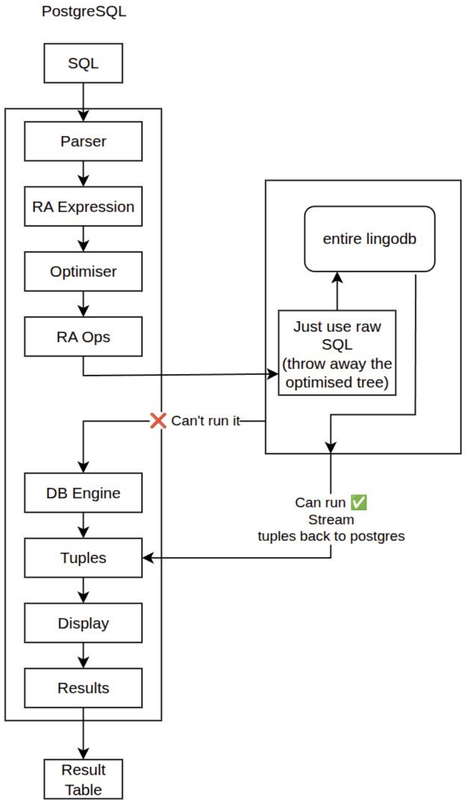
```
LOAD 'pgx_lower.so';
SET client_min_messages TO NOTICE;
SELECT 'hello';
 ?column?
----------
 hello
(1 row)

DROP TABLE IF EXISTS test;
NOTICE:  table "test" does not exist, skipping
CREATE TABLE test(id SERIAL);
INSERT INTO test(id) VALUES (1);
SELECT * FROM test;
NOTICE:  === run_mlir_with_ast_translation: Query info ===
NOTICE:  PlannedStmt ptr: 106078592379840
NOTICE:  planTree ptr: 106078593844312
NOTICE:  planTree->targetlist ptr: 106078593846288
NOTICE:  targetlist length: 1
```

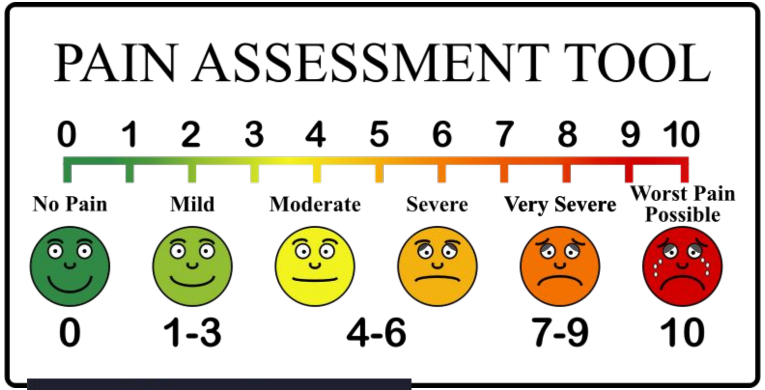
- Part of that project template had a pg_regress component
- This became the spine of the project for telling how far I've progressed
- I set up 28 regression tests
- 1 - 8 includes SELECT, sub selects and type handling
- 9 - 16 includes expression handling
- 16 - 20 includes WHERE statements
- 21 - 24 includes ORDER BY
- 25 - 28 includes GROUP BY
- So if all of these regression tests work, then thesis part B is complete.

Approach #1 - LingoDB as a library

- The idea here was to add LingoDB as a callable library
- I could start off by passing in the raw SQL string, pushing it through their system and getting an output
- That would give me a minimal viable solution to build from
- However, I am throwing away the entire optimised tree
- … But I would shotgun the entire thesis with minimal code…



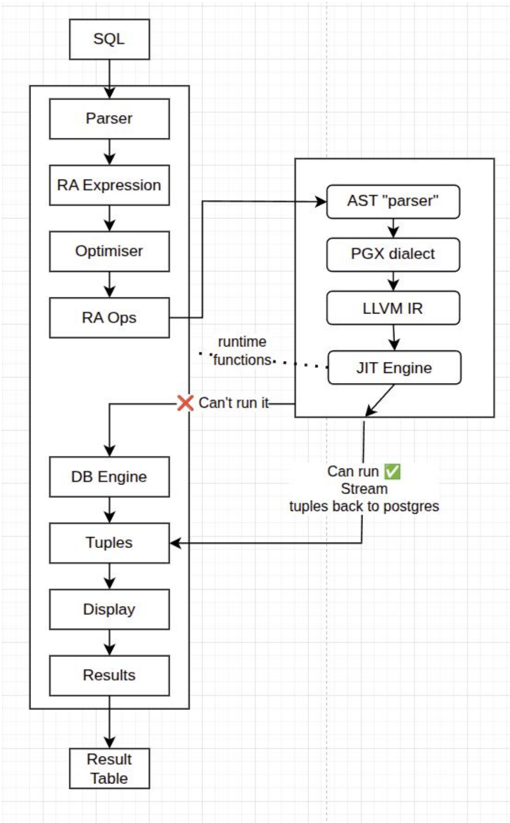
Approach #1 - LingoDB as a library



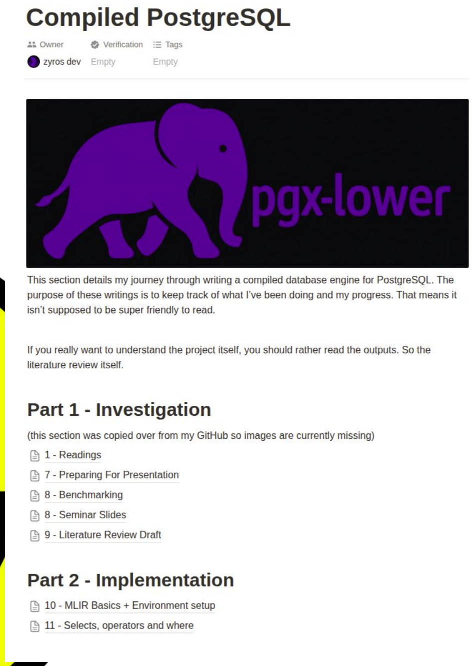
- So… This went quite badly
- LingoDB actually uses libpg_query to parse the SQL
- This hit a LOT of issues when being mingled with Postgres and I spent a lot of time trying to get it to work
- This was actually so much that I ended up completely throwing the repo away
- It was renamed from “psql-mlir-jit” to “pgx-lower”
- To be fair, I would’ve also had to rewrite a lot even if this worked

Approach #2 - From scratch

- This time I decided no black boxes
- I need to get experience with lowerings and how LingoDB interacts internally
- My MVP, on the right, can be a “parser” that is a switch statement, a single dialect, a lowering to LLVM, and a link back to runtime functions



Approach #2 - From scratch



Compiled PostgreSQL

Owner: zyros dev | Verification: Empty | Tags: Empty

This section details my journey through writing a compiled database engine for PostgreSQL. The purpose of these writings is to keep track of what I've been doing and my progress. That means it isn't supposed to be super friendly to read.

If you really want to understand the project itself, you should rather read the outputs. So the literature review itself.

Part 1 - Investigation

(this section was copied over from my GitHub so images are currently missing)

- 1 - Readings
- 7 - Preparing For Presentation
- 8 - Benchmarking
- 8 - Seminar Slides
- 9 - Literature Review Draft

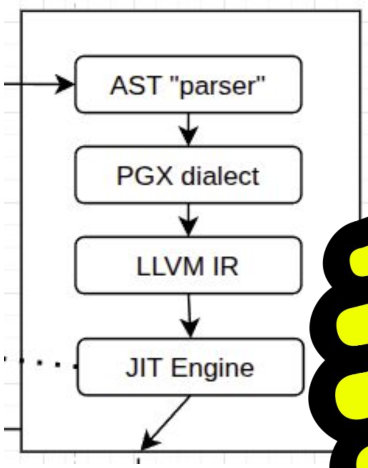
Part 2 - Implementation

- 10 - MLIR Basics + Environment setup
- 11 - Selects, operators and where

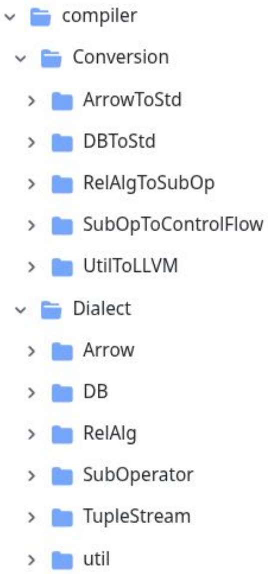
- This actually worked!
- I wrote up a set of Notion pages to document this
- Everything was going great for select statements, some of the aggregation statements, and some of the where statements!
- I was back on the schedule for the trimester
- Then… I realised several problems

Approach #2 - From scratch

- With only a single dialect, the type system is enormously painful. Int32, int64, floats, and so on need to have dedicated operations for addition
- Also my “parser” is just a switch statement that identifies the pattern and routes it to the specific tree type
- So… Now what? Well, we return to LingoDB!



Approach #3 - Leveraging LingoDB's lowerings



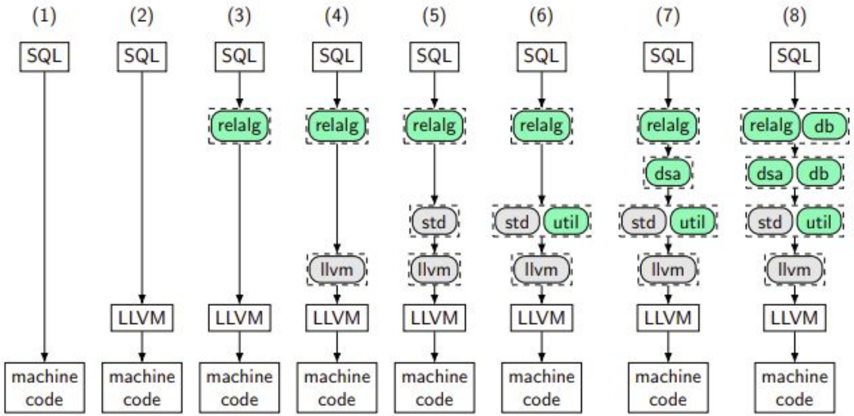
Master branch

2022 paper

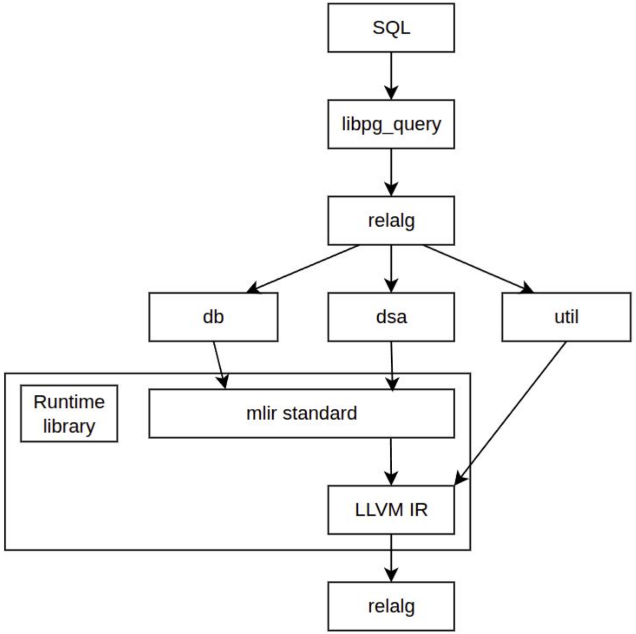
- LingoDB has a good framework of lowerings, so I checked out their master branch, made sure their license says I can do this, then copied in all their dialects and lowerings!
- The idea is I'd rip out my PGX dialect and use their entire thing
- Well… Then after a few days I reverted that, because their master branch is extremely complicated and I rolled back to their 2022 version

Approach #3 - Leveraging LingoDB's lowerings

- Let's go back to that diagram from earlier
- To me, this picture doesn't make that much sense. Or at least I didn't understand what the side-by-side things are
- I thought it just did a lowering of relalg -> db -> dsa -> util -> MLIR standard -> LLVM IR
- But those are mixed dialects…
- The main cause of my confusion was that RelAlgToDB lowering
- Actually, that lowering is RelAlgToMixed



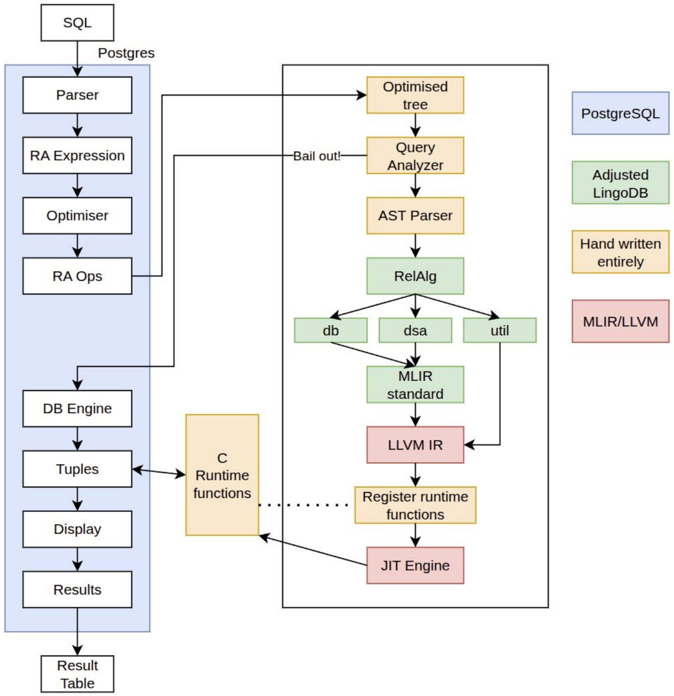
Approach #3 - Leveraging LingoDB's lowerings



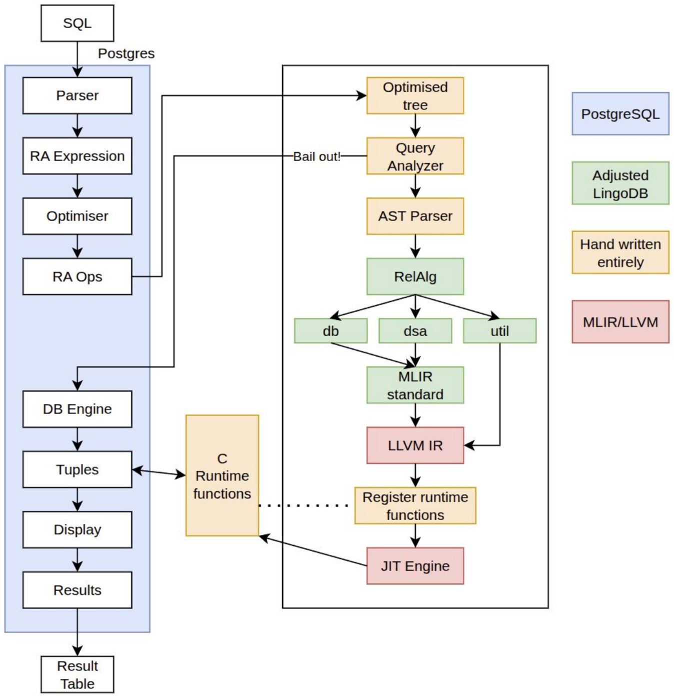
- After reading their code for a while, my understanding of lingodb is closer to this
- They have a parser for the parsed SQL to make their RelAlg dialect, lower it into one "mixed" dialect of db, dsa and util, then lower the db and dsa into mlir standard, finally LLVM IR
- One thing to note is that util -> LLVM IR isn't a separate thing, it's still inside that MLIR standard layer over there
- Also they link to a runtime library for complex operations so that the MLIR can call those functions

Approach #3 - Leveraging LingoDB's lowerings

- So now with LingoDB's lowerings implanted into my library, how is this going to fit?
- On the right, blue is PostgreSQL, green is from LingoDB, and orange are things we're going to have to implement ourselves
- The runtime library is added like that because that's how I did it in Approach #2 and I didn't want to invest time into figuring out LingoDB's method of doing it



Approach #3 - Leveraging LingoDB's lowerings



- You might be wondering why this presentation involved me overhauling my codebase multiple times
- Why haven't I shown any code in this presentation???
- Shouldn't the first part of this thesis have made this architecture clear to me???
- Well, no. This was inevitable due to complexity

Approach #3

- So, let’s look at the size of the work I’ve done here.
- LingoDB’s master branch is roughly 40,000 lines of code, LingoDB’s 2022 paper branch is 16,000 lines of code
- And I’m now up to… 38,000 lines of code?
- Looked a bit further into that and its because there’s still a lot of duplicated files from when I implanted LingoDB
- So, I should probably have a cleanup round soon to lower that
- But yes, this should explain why I abandoned doing this from scratch

LingoDB
main branch

```
(.venv) [08:14:39] xzel@comfy /home/xzel/lingo-db
> tokei include src
===============================================================================
 Language            Files        Lines         Code
===============================================================================
 C                       1          212          160
 C Header              118         6023         5323
 CMake                  34          604          526
 C++                   134        35166        32269
 C++ Header              7         2447         2113
===============================================================================
 Total                 294        44452        40391
===============================================================================
```

LingoDB
22 branch

```
(.venv) [08:16:19] xzel@comfy /home/xzel/lingo-db [0|1]
> tokei lib include/
===============================================================================
 Language            Files        Lines         Code
===============================================================================
 C Header               65         3232         2845
 CMake                  23          323          290
 C++                    88        14493        13475
===============================================================================
 Total                 176        18048        16610
===============================================================================
```

```
> mlir-support
v runtime
  > storage
    ArrowColumn.h
    ArrowView.h
    Buffer.h
    ConcurrentMap.h
    Database.h
    DataSourceIteration.h
    DateRuntime.h
    DecimalRuntime.h
    DumpRuntime.h
    EntryLock.h
    ExecutionContext.h
    ExternalPostgreSQLDatabase.h
    FloatRuntime.h
    GrowingBuffer.h
    HashMultiMap.h
    Hashtable.h
    Heap.h
    helpers.h
    IntegerRuntime.h
    LazyJoinHashtable.h
    LingoDBHashIndex.h
    metadata.h
    MetaDataOnlyDatabase.h
    postgresql_spi_stubs.h
    PostgreSQLBatch.h
    PostgreSQLDatabase.h
    PostgreSQLDataSource.h
    PostgreSQLRuntime.h
    PostgreSQLTable.h
```

pgx-lower

```
(.venv) [08:16:53] xzel@comfy /home/xzel/repos/pgx-lower
> tokei src include
===============================================================================
 Language            Files        Lines         Code
===============================================================================
 C                       1           65           43
 C Header              114        30688        22219
 CMake                  27          449          357
 C++                    94        18622        15501
 Makefile                1          200          102
===============================================================================
 Total                 237        50024        38222
===============================================================================
```

Approach #3

```cpp
class AllocaOpLowering : public OpConversionPattern<mlir::util::AllocaOp> {
   public:
   using OpConversionPattern<mlir::util::AllocaOp>::OpConversionPattern;
   LogicalResult matchAndRewrite(mlir::util::AllocaOp allocOp, OpAdaptor adaptor, ConversionPatternRewriter& rewriter) const override {
      auto loc = allocOp->getLoc();
      auto genericMemrefType = llvm::cast<mlir::util::RefType>(allocOp.getRef().getType());
      Value entries;
      if (allocOp.getSize()) {
         entries = allocOp.getSize();
      } else {
         int64_t staticSize = 1;
         entries = rewriter.create<arith::ConstantOp>(loc, rewriter.getI64Type(), rewriter.getI64IntegerAttr(staticSize));
      }
      DataLayout defaultLayout;
      const DataLayout* layout = &defaultLayout;
      Type elemType = typeConverter->convertType(genericMemrefType.getElementType());
      size_t typeSize = layout->getTypeSize(elemType);
      auto bytesPerEntry = rewriter.create<arith::ConstantOp>(loc, rewriter.getI64Type(), rewriter.getI64IntegerAttr(typeSize));

      Value entriesI64;
      if (entries.getType() == rewriter.getIndexType()) {
         entriesI64 = rewriter.create<mlir::arith::IndexCastOp>(loc, rewriter.getI64Type(), entries);
      } else {
         entriesI64 = entries;
      }

      Value sizeInBytes = rewriter.create<mlir::arith::MulIOp>(loc, rewriter.getI64Type(), entriesI64, bytesPerEntry);

      auto elemPtrType = mlir::LLVM::LLVMPointerType::get(rewriter.getContext());
      ::mlir::Value allocatedElementPtr = rewriter.create<LLVM::AllocaOp>(loc, elemPtrType, rewriter.getI8Type(), sizeInBytes, 0);
      rewriter.replaceOp(allocOp, allocatedElementPtr);

      return success();
   }
};
class AllocOpLowering : public OpConversionPattern<mlir::util::AllocOp> {
   public:
   using OpConversionPattern<mlir::util::AllocOp>::OpConversionPattern;
   LogicalResult matchAndRewrite(mlir::util::AllocOp allocOp, OpAdaptor adaptor, ConversionPatternRewriter& rewriter) const override {
      auto loc = allocOp->getLoc();

      auto genericMemrefType = llvm::cast<mlir::util::RefType>(allocOp.getRef().getType());
      Value entries;
      if (allocOp.getSize()) {
         entries = allocOp.getSize();
      } else {
         int64_t staticSize = 1;
         entries = rewriter.create<arith::ConstantOp>(loc, rewriter.getI64Type(), rewriter.getI64IntegerAttr(staticSize));
      }

      auto bytesPerEntry = rewriter.create<mlir::util::SizeOfOp>(loc, rewriter.getI64Type(), genericMemrefType.getElementType());
      Value sizeInBytes = rewriter.create<mlir::arith::MulIOp>(loc, rewriter.getI64Type(), entries, bytesPerEntry);
      Value sizeInBytesI64 = rewriter.create<mlir::arith::IndexCastOp>(loc, rewriter.getI64Type(), sizeInBytes);

      auto mallocFuncResult = LLVM::lookupOrCreateMallocFn(allocOp->getParentOfType<ModuleOp>(), rewriter.getI64Type());
      if (failed(mallocFuncResult)) {
         return failure();
      }
      LLVM::LLVMFuncOp mallocFunc = *mallocFuncResult;
      auto callOp = rewriter.create<LLVM::CallOp>(loc,
```

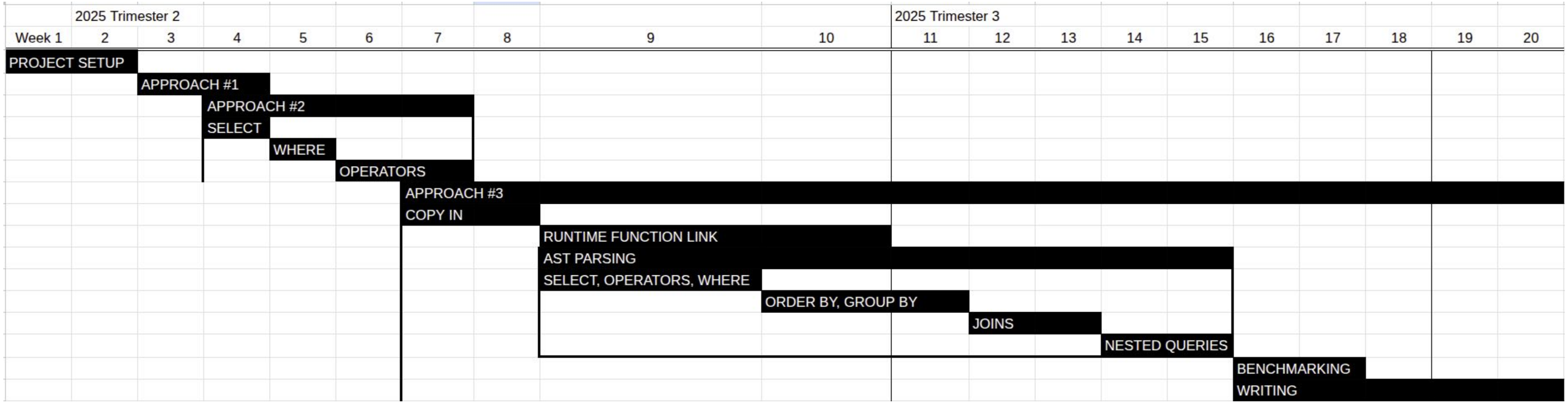
- Some of LingoDB’s lowerings are… hard to read…
- That picture on the left is **two** lowering operators from UtilToLLVM
- So I could show you the code, but it’s a bit much for a presentation

What I learned

- There's a reason someone didn't simply do this: It's time consuming
- Also, its extremely difficult to validate the final product maintains ACID compliance to keep the database safe, so it could ruin PostgreSQL for practical purposes
- There's numerous pains I didn't go into here:
  - MLIR's threading model going against Postgres's
  - Postgres's memory contexts conflict with ours

Project Next Steps

- Since the operators are implanted, it progresses through each type of query faster
- We still managed to finish all the types of parsing that we wanted by the end of part B, but we ALSO have all the lowerings and dialects done



Expected Deliverables From here

- The codebase: https://github.com/zyros-dev/pgx-lower
- This should be… approachable to install by the end of part C. Currently, installing that is quite painful. Hardcoded paths and so on.
- A report at the end of part C explaining components properly

Thank you!