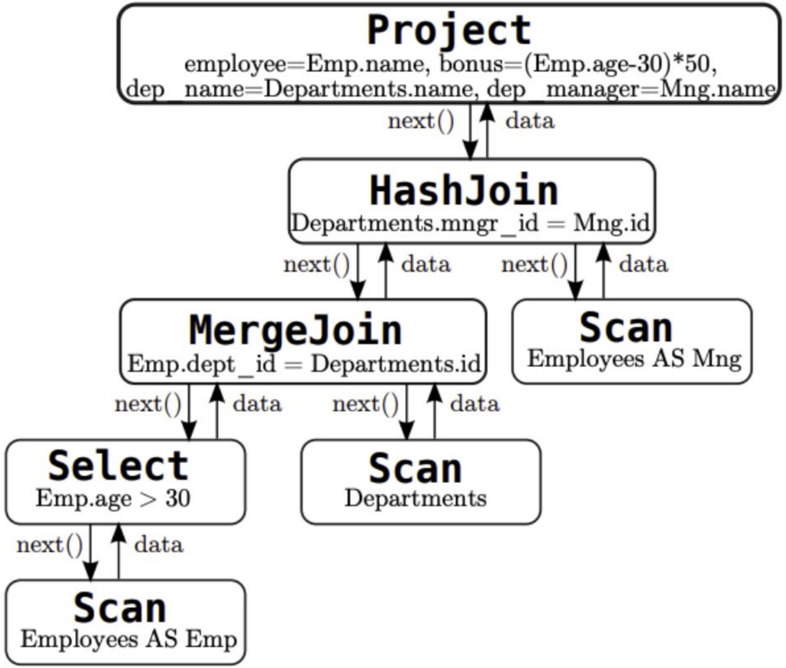
Exploring Just-in-Time Compilation in Relational Database Engines

Table of Contents

Project Recap

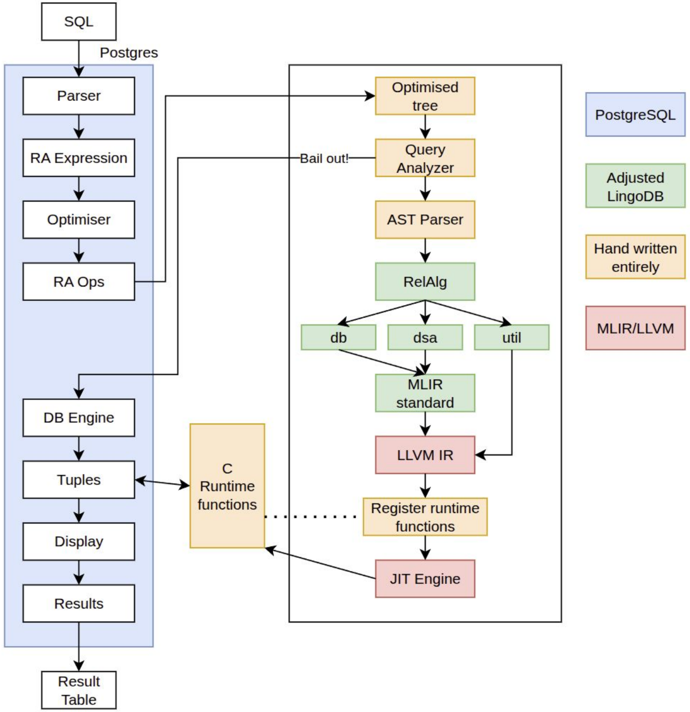


- PostgreSQL is the most popular relational database in the world
- It uses a volcano model for execution
- In theory, changing this to a compiler can make queries 2x faster

Project Recap

- There are 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



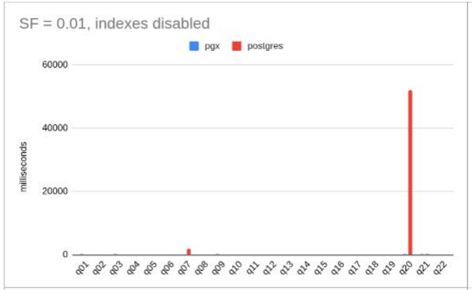
- In the last presentation we initialised our codebase with the structure on the left
- Part C was mostly about implementing the parser for all types of queries
- And to offboard the runtime functions back over to Postgres!

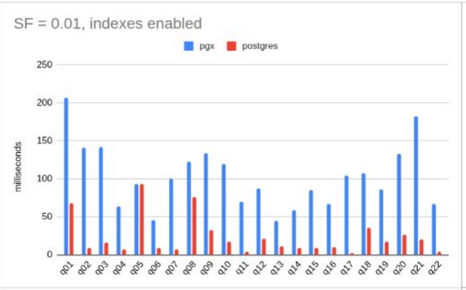
Benchmarks!

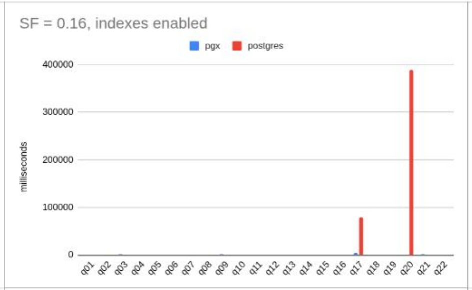
- This is really the only important part here and our main goal
- If we can't show that this beats PostgreSQL significantly, it was pointless
- So let's start there!

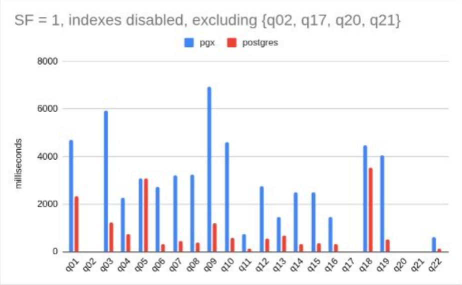
Peek: Benchmarks

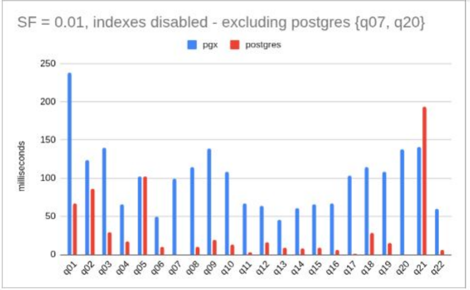
- Keep in mind - pgx does NOT have indexes implemented!
- I’ll walk through these results at the end

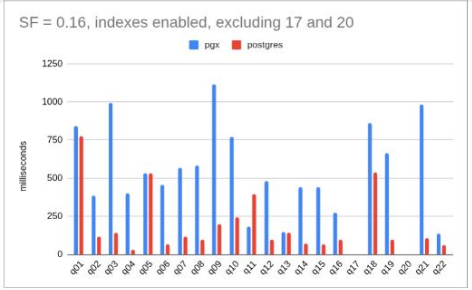






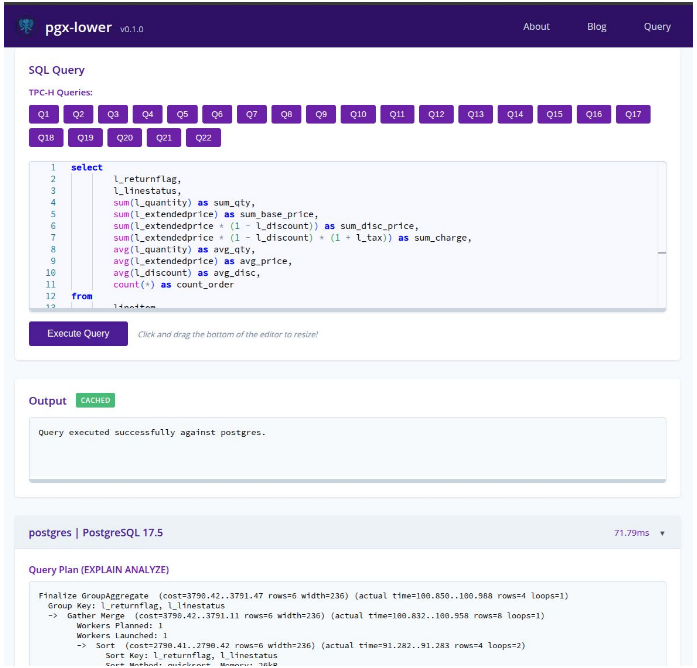






Website

- I also launched a small demo website where you can run queries on your own
- This has a really small dataset of the TPC-H data, and you can see a display of all the lowerings our compiler goes through
- Hopefully it doesn't end up too crowded!



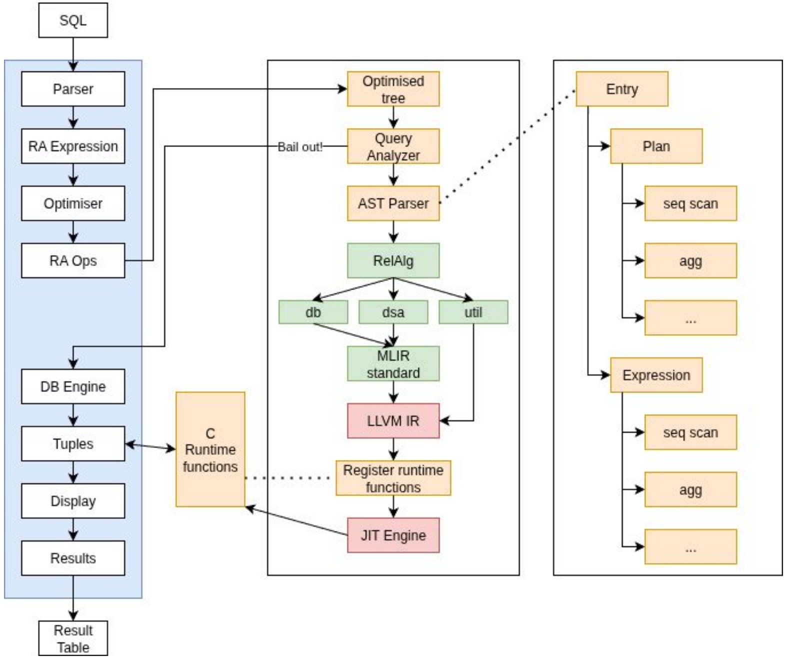
Implementing AST Translation

- When we run a query, we receive a “plan tree” pointer from Postgres
- At a high level there are two types of nodes: plan nodes and execution nodes
- For instance, here on the right is a sample of a tree. I used Postgres’s visualisation tool, but these tree still end up completely massive


```
{AGG
:plan.startup_cost 31.160439593278905
:plan.total_cost 31.19543959327891
:plan.plan_rows 1
:plan.plan_width 272
:plan.parallel_aware false
:plan.parallel_safe true
:plan.async_capable false
:plan.plan_node_id 0
:plan.targetlist (
   {TARGETENTRY
   :expr
      {VAR
      :varno -2
      :varattno 1
      :vartype 1042
      :vartypmod 29
      :varcollid 100
      :varnullingrels (b)
      :varlevelsup 0
      :varnosyn 0
      :varattnosyn 0
      :location -1
      }
   :resno 1
   :resname supp_nation
   :ressortgroupref 1
   :resorigtbl 38449629
   :resorigcol 2
```


```
AGG
└── SORT
    └── NESTLOOP
        ├── NESTLOOP
        │   ├── NESTLOOP
        │   │   ├── NESTLOOP
        │   │   │   ├── NESTLOOP
        │   │   │   │   ├── SEQSCAN (with date filters)
        │   │   │   │   └── INDEXSCAN
        │   │   │   └── INDEXSCAN
        │   │   └── INDEXSCAN
        │   └── INDEXSCAN (nation filter: FRANCE or GERMANY)
        └── INDEXSCAN (nation filter: GERMANY or FRANCE)
```

Implementing AST Translation



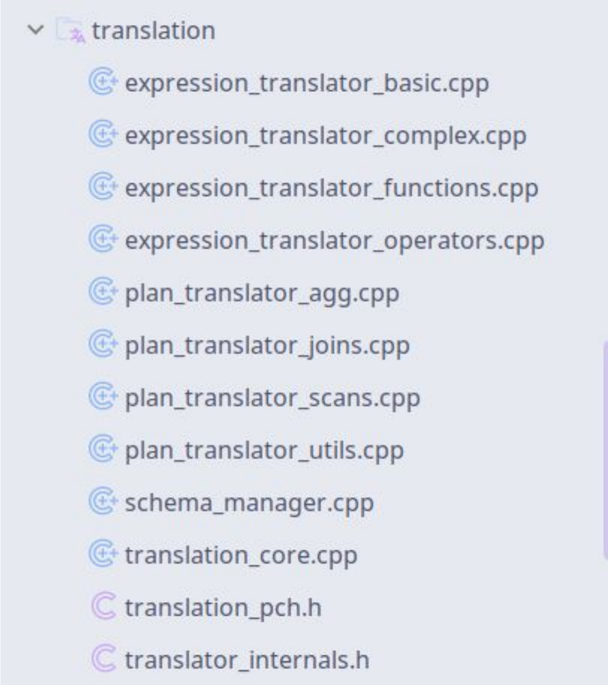
So our translator looks like this on the left: we have a core entry for expressions and a core entry for plans

This goes down into a switch statement depending on the node tag, then we have our logic for each type

Inside of those, they call translation functions again and we have a recursive descent parser!

Implementing AST Translation

- Inside of each function we need to generate our RelAlg. This RelAlg comes from LingoDB's code
- AST translation is probably the majority of the work for this project
- It's a reasonably complex chunk of code, and probably the only more complicated thing is joins



Implementing AST Translation - Aggregation

```
// Plan node translation methods
auto translate_plan_node(QueryCtxT& ctx, Plan* plan) -> TranslationResult;
auto translate_seq_scan(QueryCtxT& ctx, SeqScan* seqScan) -> TranslationResult;
auto translate_index_scan(QueryCtxT& ctx, IndexScan* indexScan) -> TranslationResult;
auto translate_index_only_scan(QueryCtxT& ctx, IndexOnlyScan* indexOnlyScan) -> TranslationResult;
auto translate_bitmap_heap_scan(QueryCtxT& ctx, BitmapHeapScan* bitmapScan) -> TranslationResult;
auto translate_agg(QueryCtxT& ctx, const Agg* agg) -> TranslationResult;
auto translate_sort(QueryCtxT& ctx, const Sort* sort) -> TranslationResult;
auto translate_limit(QueryCtxT& ctx, const Limit* limit) -> TranslationResult;
auto translate_gather(QueryCtxT& ctx, const Gather* gather) -> TranslationResult;
auto translate_gather_merge(QueryCtxT& ctx, const GatherMerge* gatherMerge) -> TranslationResult;
auto translate_merge_join(QueryCtxT& ctx, MergeJoin* mergeJoin) -> TranslationResult;
auto translate_hash_join(QueryCtxT& ctx, HashJoin* hashJoin) -> TranslationResult;
auto translate_hash(QueryCtxT& ctx, const Hash* hash) -> TranslationResult;
auto translate_nest_loop(QueryCtxT& ctx, NestLoop* nestLoop) -> TranslationResult;
auto translate_material(QueryCtxT& ctx, const Material* material) -> TranslationResult;
auto translate_memoize(QueryCtxT& ctx, const Memoize* memoize) -> TranslationResult;
auto translate_subquery_scan(QueryCtxT& ctx, SubqueryScan* subqueryScan) -> TranslationResult;
auto translate_cte_scan(QueryCtxT& ctx, const CteScan* cteScan) -> TranslationResult;
```

auto translate_agg(QueryCtxT& ctx, const Agg* agg) -> TranslationResult;

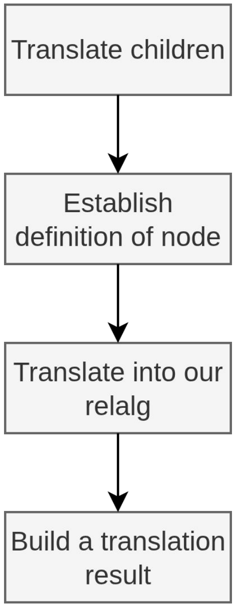
- When I started this I really wanted to make these translations a clean repetitive pattern
- We have the concept of a Translation Result, and at first this was supposed to only flow upwards; you'd ask your children for a translation result, then pass your own one up to your parents
- Then you'd have a TranslationContext and this would flow down to your parents

Implementing AST Translation - Aggregation

- Our aggregation function is quite big, and is split into several sections
- Haha, don’t try to read the code from here
- The photos are just to get an idea of how much is involved in translating it

Implementing AST Translation - Aggregation

- In short, you'll see this pattern for almost all the translation functions
- We start by translating the children,
- Then we establish the definition of the node in our world (which columns should we group by?)
- Then we generate the RelAlg in the builder
- We write our translation results

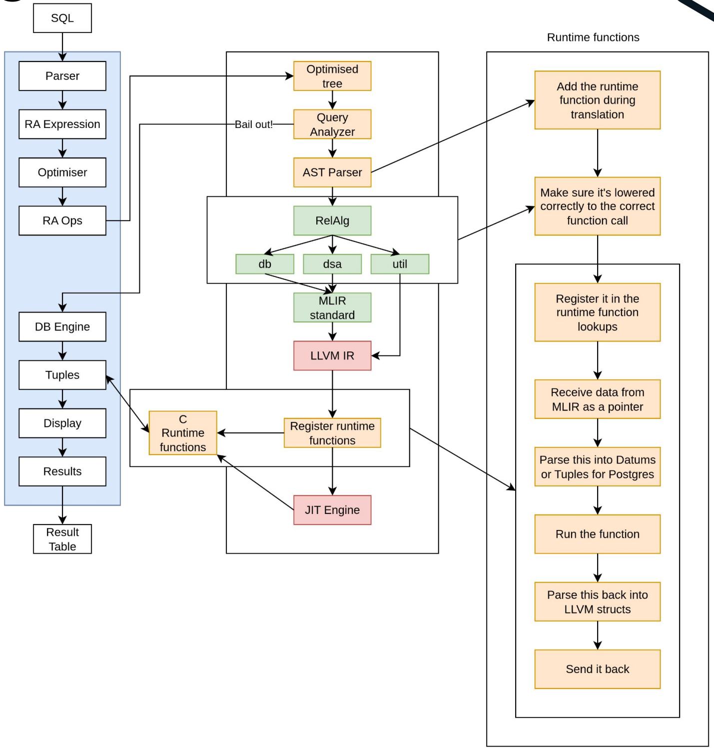


Implementing Runtime Functions

- First off, why do we need runtime functions?
- A big pain-point is that LingoDB is column-oriented and all in memory, while Postgres is row-oriented and on disk
- This means if we run a large benchmark, it will cause our RAM to overflow to our swap, we'll thrash and potentially crash
- Not good.
- So the main things we need runtime functions for is sorting and hash maps. LingoDB simply used std::sort and std::unordered_map
- We'll walk through how I added sort, since that's the most important one.
- The secondary use is for reading/returning tuples. This was already done in part B of the thesis, so I won't go over it too much

Implementing Runtime Functions

- There are a couple of steps to implementing runtime functions, as shown on the right
- We need to translate them into RelAlg calls
- Make sure it's lowered into the correct function call
- We need to register these function calls inside of LingoDB's structure
- Then we're going to need to parse the data we get from LLVM, run the function, parse it back to LLVM results and send it back



Implementing Runtime Functions - Sort - RelAlg

```cpp
auto& columnManager = ctx.builder.getContext()->getOrLoadDialect<mlir::relalg::RelAlgDialect>()->getColumnManager();
std::vector<mlir::Attribute> sortSpecs;
for (int i = 0; i < sort->numCols; i++) {
    const AttrNumber colIdx = sort->sortColIdx[i];
    if (colIdx <= 0 || colIdx >= MAX_COLUMN_INDEX)
        continue;

    auto spec = mlir::relalg::SortSpec::asc;
    if (sort->sortOperators) {
        if (char* oprname = get_opname(sort->sortOperators[i])) {
            spec = (std::string(oprname) == ">" || std::string(oprname) == ">=") ? mlir::relalg::SortSpec::desc
                                                                                  : mlir::relalg::SortSpec::asc;
            pfree(oprname);
        }
    }

    ListCell* lc;
    int idx = 0;
    foreach (lc, sort->plan.targetlist) {
        if (++idx != colIdx)
            continue;

        const TargetEntry* tle = static_cast<TargetEntry*>(lfirst(lc));
        if (IsA(tle->expr, Var)) {
            const Var* var = reinterpret_cast<Var*>(tle->expr);

            if (var->varattno > 0 && var->varattno <= childResult.columns.size()) {
                const auto& column = childResult.columns[var->varattno - 1];
                sortSpecs.push_back( x: mlir::relalg::SortSpecificationAttr::get(
                    ctx.builder.getContext(), attr: columnManager.createRef( scope: column.table_name,  column.column_name), spec));
            }
        }
        break;
    }
}
```

- In the RelAlg translation we iterate over the sort information, then create a “Sort Specification”, and write a RelAlg item with a pointer to the sort specification
- This specification idea is something I’m proud of.
- Postgres has memory contexts, so I register a pointer in the transaction-memory
- Then I can create an arbitrary object, and pass a pointer to it through the RelAlg.

```cpp
auto tupleStreamType = mlir::relalg::TupleStreamType::get( ctx: ctx.builder.getContext());
const auto sortOp = ctx.builder.create<mlir::relalg::SortOp>(
    location: ctx.builder.getUnknownLoc(), [>>] tupleStreamType, childResult.op->getResult( idx: 0), ctx.builder.getArrayAttr( sortSpecs));

TranslationResult result;
```

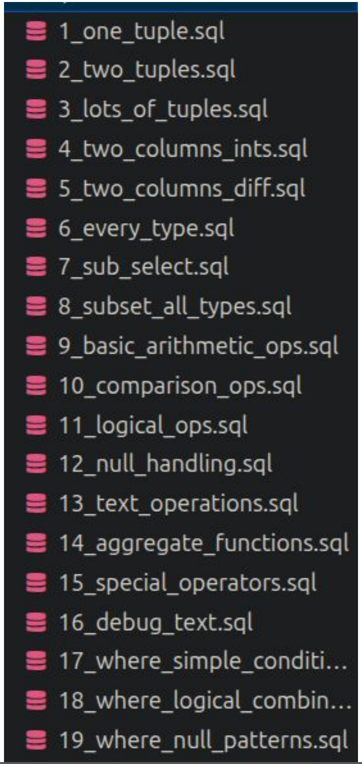
Implementing Runtime Functions - Sort - Lowerings

- This next step is painful
- There isn't really a strong formula for how to do this since we need to dig through LingoDB code
- Below is my diff for the SortOp in the RelAlg -> DB translation
- I rip everything out, and I have my own new idea of an iterator object
- So I need to create a start iterator, then add columns, then a sort command, and a get next tuple interface

```cpp
::mlir::Value createSortPredicate(::mlir::OpBuilder& builder, std::vector<std::pair<::mlir::Value, ::mlir::Value>> sortCrite
  if (pos < sortCriteria.size()) {
    ::mlir::Value lt = builder.create<mlir::db::CmpOp>(sortOp->getLoc(), mlir::db::DBCmpPredicate::lt, sortCriteria[pos].f
    lt = builder.create<mlir::db::DeriveTruth>(sortOp->getLoc(), lt);
    auto ifOp =
      builder.create<mlir::scf::IfOp>(sortOp->getLoc(), mlir::TypeRange{builder.getI1Type()}, lt, true, true);
    {
      auto& thenBlock = ifOp.getThenRegion().front();
      mlir::OpBuilder thenBuilder(&thenBlock, thenBlock.begin());
      thenBuilder.create<mlir::scf::YieldOp>(sortOp->getLoc(), trueVal);
    }
    {
```

```cpp
// createSortPredicate removed - PgSort uses SortSpecification metadata instead of comparator region
virtual void produce(mlir::relalg::TranslatorContext& context, ::mlir::OpBuilder& builder) override {
  auto scope = context.createScope();
  orderedAttributes = mlir::relalg::OrderedAttributes::fromColumns(requiredAttributes);
  auto tupleType = orderedAttributes.getTupleType(builder.getContext());
  // Use GenericIterableType with "pgsort_iterator" to distinguish from regular Vector
  vector = builder.create<mlir::dsa::CreateDS>(sortOp.getLoc(), mlir::dsa::GenericIterableType::get(builder.getContext(), tupl
  children[0]->produce(context,builder);
  // PgSort uses SortSpecification metadata passed through attribute, not comparator region
  // DSA::SortOp is created but the region is unused - performSort() uses the metadata directly
  builder.create<mlir::dsa::SortOp>(sortOp->getLoc(), vector);
  {
```

Implementation - Correctness



- The rest of this part can be summarised with my regression tests
- You can see the order that I implemented things in
- Plus at the end I hit some non-deterministic plan trees which I repeatedly ran the TPC-H set on

Implementation

- Those two jobs were repeated for everything. It's a lot. I won't dig into it.
- Below is the `tokei` command for `src/pgx-lower` and `include/pgx-lower` which doesn't include ALL my code, but most of it

- As a whole this is more like 34,000 lines of code.
- So, let's move onto profiling!

Language	Files	Lines	Code	Comments	Blanks
C	1	166	125	4	37
C Header	20	1996	1436	194	366
CMake	1	7	7	0	0
C++	29	12434	9930	426	2078
Total	56	15412	11498	1227	2687

Language	Files	Lines	Code	Comments	Blanks
C	1	166	125	4	37
C Header	92	5140	4143	257	740
CMake	27	396	345	13	38
C++	115	26590	22406	857	3327
Markdown	6	1065	0	796	269
Total	241	33357	27019	1927	4411

Benchmarking and Profiling - Setup

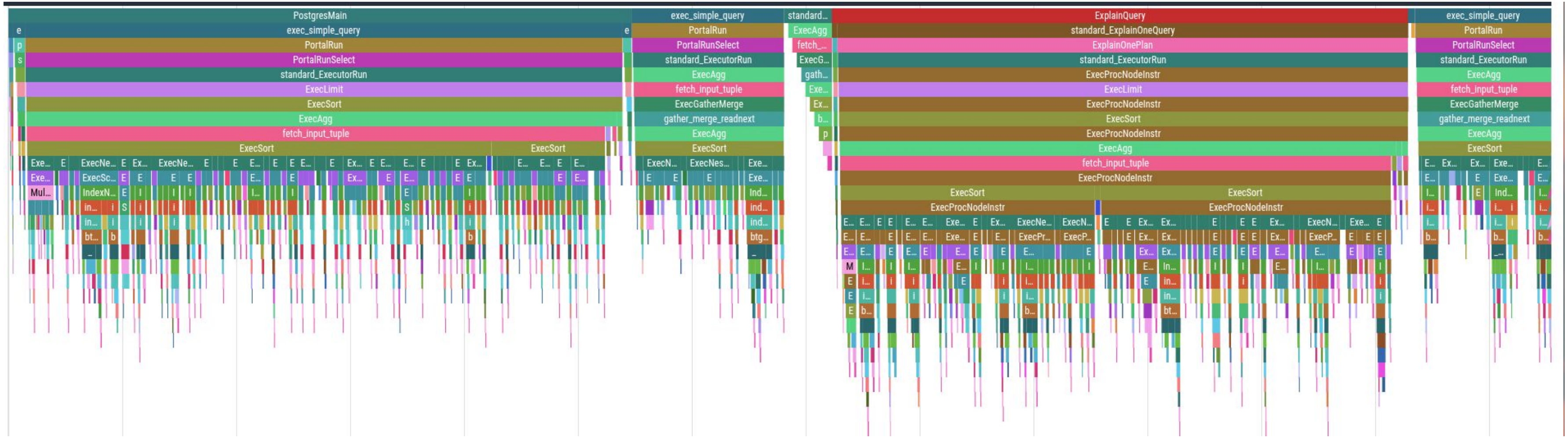
- In part A of this project I did some benchmarking to isolate how much CPU time Postgres uses to prove the concept of this thesis
- However, for profiling now I need something quite more-ish
- A big challenge is if most of my run is in the LLVM, I can’t really insert runtime symbols. At all.
- Which leads me to magic trace!
- https://github.com/janestreet/magic-trace

Benchmarking and Profiling - Setup

- Using magic-trace properly requires an intel cpu, and my main computer doesn't have that
- I dug this ThinkCentre out of my closet, then stacked it on top of my tower of compute
- This is going to be doing the magic trace!

Benchmarking and Profiling - Postgres Flamechart

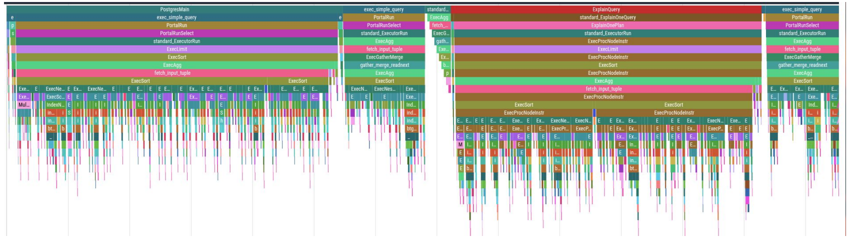
- Query 3 in TPC-H - Runtime of approx 260ms @ SF 0.05

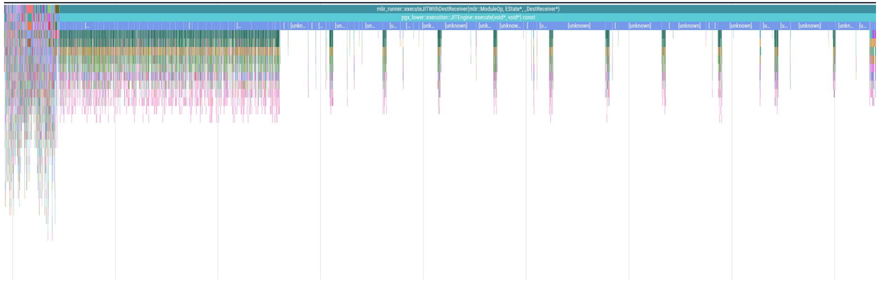


Benchmarking and Profiling - PGX-Lower flame chart

- Q03 - Runtime of approx 4.5 seconds

So what’s going wrong?

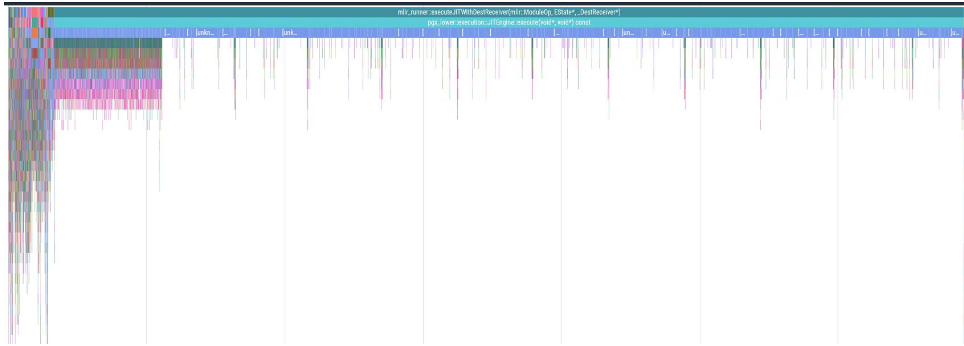




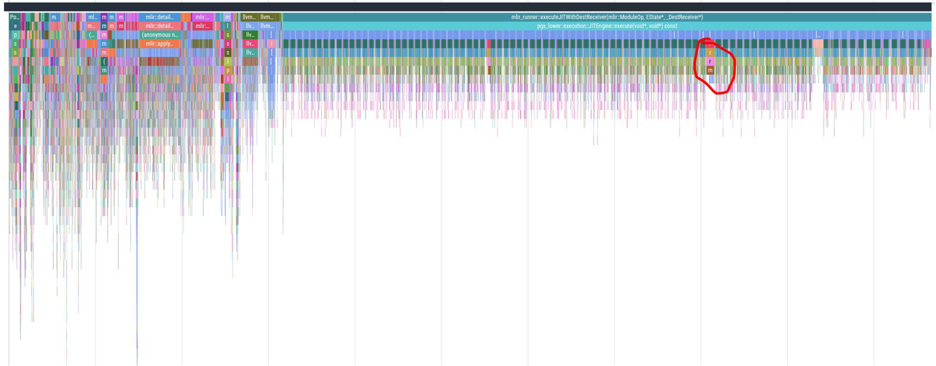
- Firstly, our tuple reads are taking too much time. I did some reading and there’s a critical problem
- When we read a tuple, we look up the data type for a column, and this is recomputed every single lookup
- Also, we read tuples one-by-one, instead of pages
- But the big issue here is our LLVM!

Benchmarking and Profiling - Optimisations

- Either way I fixed up the tuple reading first, and you can see the effect on the chart on the right
- It becomes a chunk faster on the section over to the left
- I also disabled some logs at compile-time and introduce a PGX_HOT_LOG macro that gets removed completely in compile times



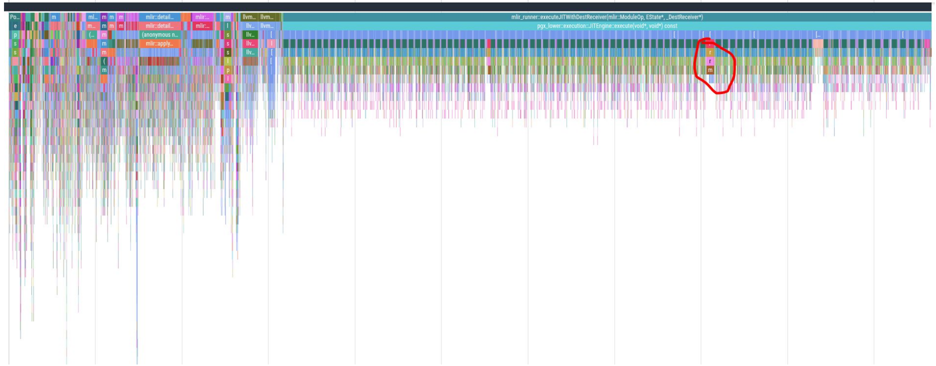
Benchmarking and Profiling - Optimisations



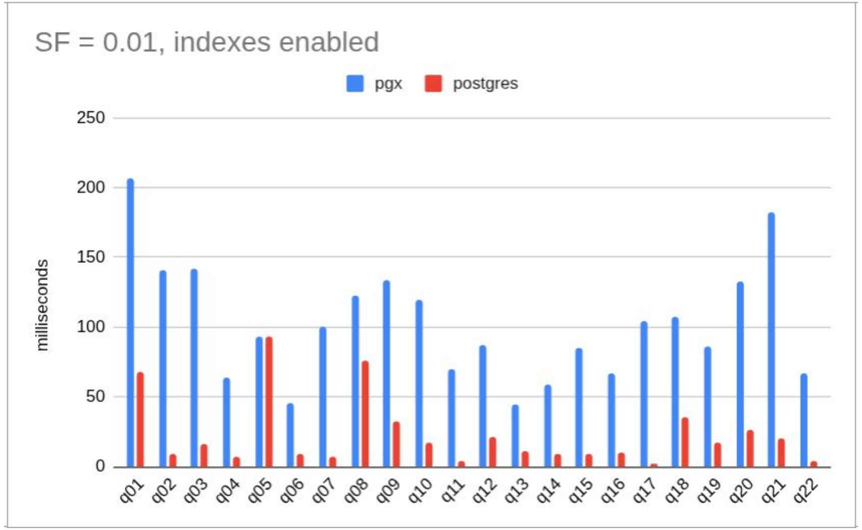
- The next optimisation took me some time to notice
- We were iterating through the entire set during a join and not using the hash join
- So I forced it to listen to postgres's plan tree for this situation, and that little red circle there is the hash join runtime
- This completely condensed the runtime down
- We're still about 400ms. Ten times faster! But still too slow for this particular query

Benchmarking and Profiling - Reflection on benchmarks

- The next improvements become quite rough.
- I need to implement index scans, merge sorts, and various newer features
- However, we can go disable index scans in postgres to level the playing field



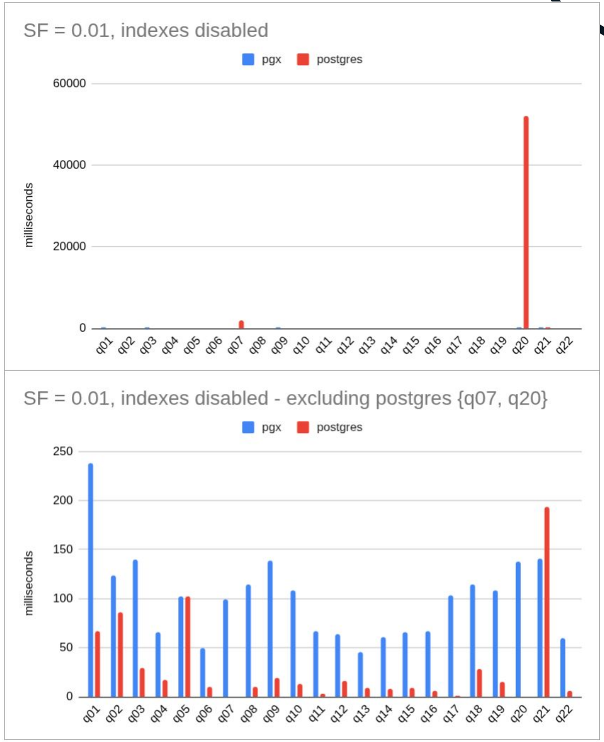
Benchmarking and Profiling - Small scale factor



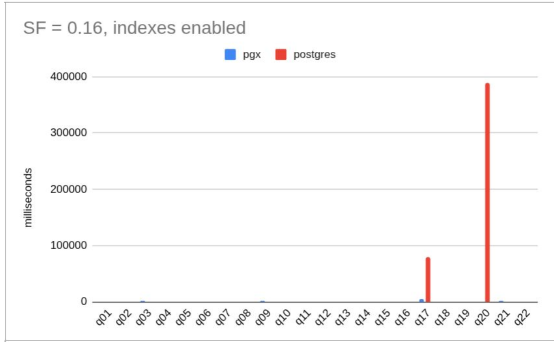
- With indexes enabled in postgres, postgres is a chunk faster at a small scale factor
- This is from the overhead of query compilation as well as postgres having indexes, and us lacking them

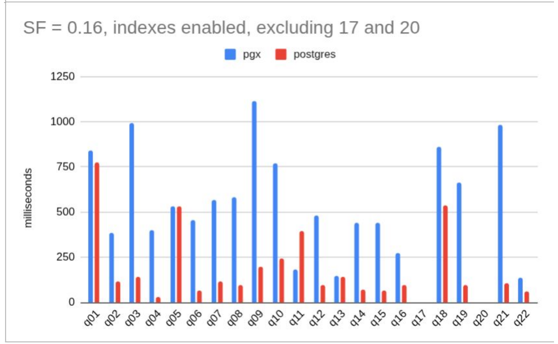
Benchmarking and Profiling - Small scale factor

- If we disable indexes in PostgreSQL, this happens
- You can see certain queries completely skyrocket in latency and jump to a new magnitude
- We're seeing a good chunk of benefit here!



Benchmarking and Profiling - Mid scale factor

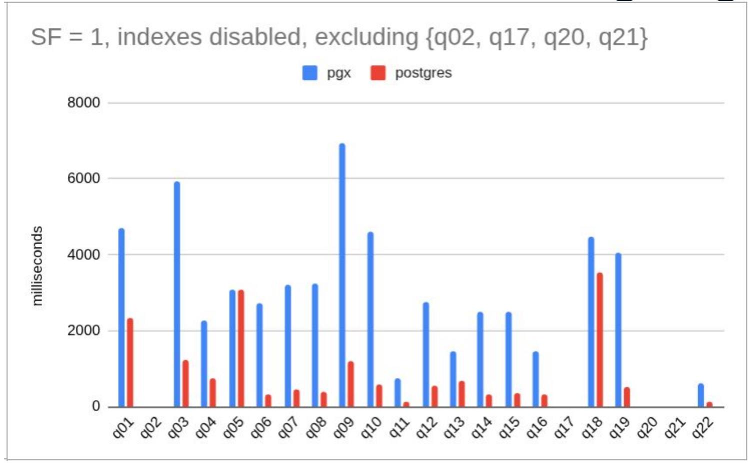




- If we increase our scale factor a bit to 0.16
- We can see even with indexes enabled, those q20 and q17 still skyrocket
- However, overall it seems PostgreSQL is still faster in this range overall
- PostgreSQL can't really function at all without indexes at this scale factor

Benchmarking and Profiling - High scale factor

- For a high scale factor, several queries were taking on the order of hours (q02, q17, q20, q21) so they were disabled
- You can see there's some mixed results, and it depends on what you look at



Benchmarking and Profiling - What would it take?

- What changes would we have to make to get pgx lower to be faster than PostgreSQL?
- We need to implement indexes, more effective caching/buffering, and tune the LLVM compilation, or swap compilers entirely
- In our benchmarks it seems we either read too many tuples or we don't read enough tuples at once.

SF = 0.01, indexes enabled. Query 20. Top: pgx lower 1.4 seconds, bottom: postgres 0.14 seconds

Finishing notes

- I wouldn’t use LLVM/MLIR for this project if I redid it
- I definitely learned a lot about Postgres, its internal functions, and how a compiler functions
- Without LingoDB’s infrastructure I would’ve been quite lost
- LLVM/MLIR is far too heavy-weight for the compiling step
- The main benefit would’ve been those optimisation passes, but inside of Postgres… We should just use the Postgres optimiser!
- It makes sense for a database from scratch like LingoDB
- I would suggest using WebAssembly, like Mutable showed its a perfect fit for these things
- Also I generally dislike C++ for something where correctness is critical

Skipped topics

- A number of topics were skipped here which I'll put into the final report
- Parameter passing / variable contexts
- Using memory contexts from Postgres efficiently
- The pains of the C -> C++ memory boundary
- Logging infrastructure
- Decimal/string hacks
- My attempt to introduce index scans, and what was hard about it
- Nullability pains
- And more!
- Also, I had a timer in CLion throughout this project and we're finishing at 467 hours

Links and thanks for listening!

- Interactive querying
  - https://pgx.zyros.dev/query
  - The blog posts here are incomplete!
- The main repo of the project
  - https://github.com/zyros-dev/pgx-lower
- Report will be written over the next couple of weeks, submitted then also uploaded on pgx.zyros.dev !

- Any questions?