Database Query Compilation: Our Journey

Thomas Neumann and Viktor Leis

Technische Universität München

How do Database Systems Execute Queries?

1. Tuple-at-a-time interpretation:
   MySQL, PostgreSQL (mostly), Microsoft SQL Server (default)
2. Vector-at-a-time interpretation (vectorization):
   DuckDB, Snowflake, MS SQL Server ("columnstore index")
3. Compilation:
   Amazon Redshift, Hyper, Umbra

The Difficulty of Vector-at-a-time Processing

"[...] needs to handle the N-ary nature of the operators. As a result, expressing complex relational operators in a vectorized model is a challenge in itself."
Marcin Zukowski, doctoral dissertation

The Difficulty of Vector-at-a-time Processing

"[...] needs to handle the N-ary nature of the operators. As a result, expressing complex relational operators in a vectorized model is a challenge in itself."

Marcin Zukowski, doctoral dissertation

```
// Input: probe relation with M attributes and K keys, hash-table containing
// N build attributes
// 1. Compute the bucket number for each probe tuple.
// ... Construct bucketV in the same way as in the build phase ...
// 2. Find the positions in the hash table
// 2a. First, find the first element in the linked list for every tuple,
// put it in groupIdV, and also initialize toCheckV with the full
// sequence of input indices (0..n-1).
lookupInitial(groupIdV, toCheckV, bucketV, n);
m = n;
while (m > 0) {
   // 2b. At this stage, toCheckV contains m positions of the input tuples for
   // which the key comparison needs to be performed. For each tuple
   // groupIdV contains the currently analyzed offset in the hash table.
   // We perform a multi-column value check using type-specific
   // check() / recheck() primitives, producing differsV.
   for (i = 0; i < K; i++)
      check[i](differsV, toCheckV, groupIdV, ht.values[i], probe.keys[i], m);
   // 2c. Now, differsV contains 1 for tuples that differ on at least one key,
   // select these out as these need to be further processed
   m = selectMisses(toCheckV, differV, m);
   // 2d. For the differing tuples, find the next offset in the hash table,
   // put it in groupIdV
   findNext(toCheckV, ht.next, groupIdV, m);
}
// 3. Now, groupIdV for every probe tuple contains the offset of the matching
// tuple in the hash table. Use it to project attributes from the hash table.
// (the probe attributes are just propagated)
for (i = 0; i < N; i++)
   gather[i](result.values[M + i], groupIdV, ht.values[i], n);
```

Compilation

Can generate **any** code:

```
// Probe the "probe" relation against the hash table
for (i = 0; i < probe.size; i++) {
   bucket = rehash(hash(probe.values[0][i]), probe.values[1][i]) & mask;
   for (current = ht.bucket[bucket]; current; current = ht.next[current]) {
      if (ht.values[0][current] == probe.values[0][i] &&
          ht.values[1][current] == probe.values[1][i]) {
         ... // match
      }
   }
}
```

Compilation

Can generate **any** code:

```
// Probe the "probe" relation against the hash table
for (i = 0; i < probe.size; i++) {
   bucket = rehash(hash(probe.values[0][i]), probe.values[1][i]) & mask;
   for (current = ht.bucket[bucket]; current; current = ht.next[current]) {
      if (ht.values[0][current] == probe.values[0][i] &&
          ht.values[1][current] == probe.values[1][i]) {
         ... // match
      }
   }
}
```

Simplifies development:

1. prototype new operator by hand-coding it outside the database system
2. benchmark, optimize, refine
3. write code that generates hand-written code

Challenges of Compilation

- Some workloads contain many short queries
- Low latency is crucial for interactive applications
- Machine-generated queries can be huge (e.g., 10MB SQL text)
- High throughput is crucial for long-running queries
- Hard to predict upfront how long a query will take

Compilation to C++ (before 2011)

- We considered this very early on
  - + easy
  - + high throughput
  - – very high compilation latency

Compilation to C++ (before 2011)

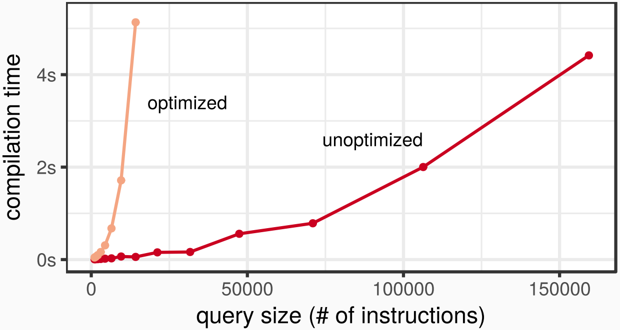
- We considered this very early on
  + easy
  + high throughput
  – very high compilation latency
- Redshift still does it
  - multi-tenant code cache with high hit rates (99%+)
  - nevertheless, about half the end-to-end latency appears to be in compilation
    `https://github.com/amazon-science/redset`

Start of the Journey: Compilation to LLVM IR (2011 - 2017)

- Hyper was one of the first database systems to compile to LLVM IR
- Initially the only execution mode
  - \+ high query throughput
  - \+ fairly good compilation latency for most queries

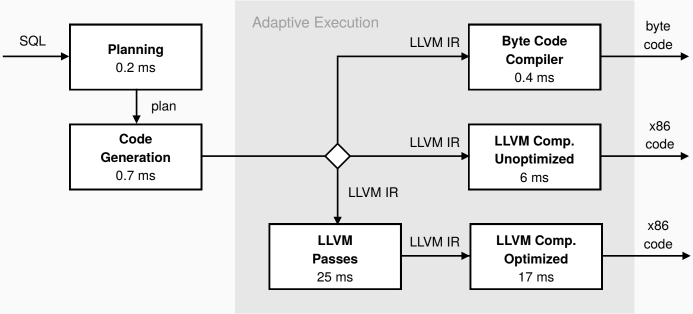
Start of the Journey: Compilation to LLVM IR (2011 - 2017)

- Hyper was one of the first database systems to compile to LLVM IR
- Initially the only execution mode
  - + high query throughput
  - + fairly good compilation latency for most queries
  - – not ideal for workloads with many small heterogeneous queries
  - – cannot handle generated monster queries



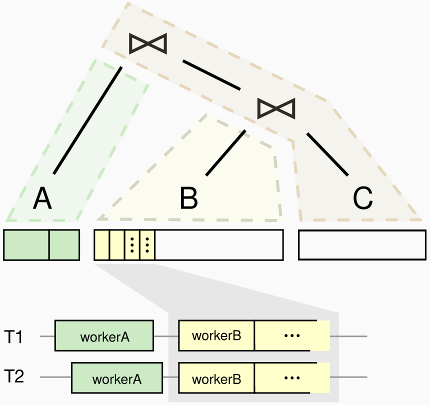
Compilation to Bytecode (2018 - 2020)

- In 2018, we added a bytecode interpreter to Hyper
  - short queries: bytecode interpreter
  - medium queries: LLVM without optimizations
  - long-running queries: LLVM with optimizations



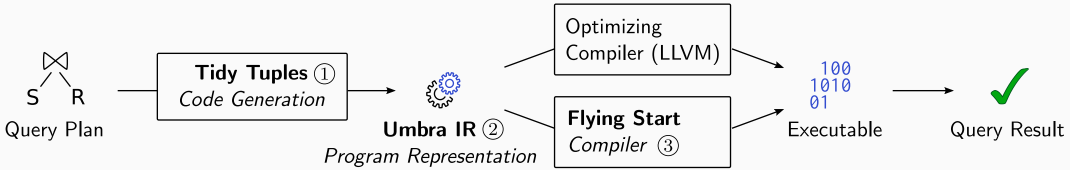
Adaptive Execution (2018 -)

- Start with bytecode interpretation
- Extrapolate runtime and run LLVM compiler in the background if beneficial
- Morsel-driven parallelization provides points for switching modes



Emit Machine Code Directly (2020 -)

- In Umbra, instead of bytecode we directly emit machine code (x86-64/ARM64)
- Guaranteed linear compilation runtime
- Also: database-specific intermediate representation (Umbra IR)
- Lower latency and higher throughput than bytecode
- LLVM is still used for long-running queries for maximum throughput



Invest in Tooling

- Debugging generated code should be pleasant and easy
- Invest some time in tools to get a nice code representation
- Allows for smooth debugging and profiling, interoperability with C++
- Takes some effort, but that is a one-time investment

```
#### %HashJoinTranslator_cpp_805_ = getelementptr int8 %state, i32 1040 (58441)
lea r14, byte ptr [rbx+1040]
#### %TableScanLoleop_cpp_394_ = getelementptr object umbra::RestrictionValues %TableScanLol
lea r15, byte ptr [r8+40]
#### %value = getelementptr object umbra::RestrictionValues %TableScanLoleop_cpp_394_, i32 (
#### store int8* %HashJoinTranslator_cpp_805_, %value (58570, fused with gep)
mov qword ptr [r15], r14
#### %TableScanLoleop_cpp_395_ = getelementptr object umbra::RestrictionValues %TableScanLol
lea r14, byte ptr [r8+40]
#### %upperValue = getelementptr object umbra::RestrictionValues %TableScanLoleop_cpp_395_,
#### store int32 i32 3, %upperValue (58676, fused with gep)
mov dword ptr [r14+16], 3
#### %TableScanLoleop_cpp_440_ = call i32 umbra::RestrictionValues::postprocessRestrictions
mov qword ptr [rsp+56], rdi
mov rdi, rbx
mov rsi, qword ptr [rsp+56]
mov rdx, r8
mov ecx, 2
mov rax, 93825086633954
call rax
#### store int32 %TableScanLoleop_cpp_440_, %TableScanLoleop_cpp_308_ (58809)
mov dword ptr [r13], eax
#### %RelationColumnLogic_cpp_651_ = getelementptr int8 %state, i32 704 (58831)
lea r13, byte ptr [rbx+704]
#### call void umbra::RelationColumn::attachReader (ptr 000050800003C0A0, %RelationColumnLogi
mov rdi, 88510686281888
mov rsi, r13
mov rax, 93825082799078
call rax
```

Summary

- Compilation is the most flexible and efficient way to execute queries
- Learning curve, but pays off in the long run
- Database systems have requirements that differ from traditional compilers
- Requires custom compilation infrastructure and tooling
- Techniques are documented in academic papers:

  - Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. VLDB Endow. 4(9): 539-550 (2011)
  - Thomas Neumann, Viktor Leis: Compiling Database Queries into Machine Code. IEEE Data Eng. Bull. 37(1): 3-11 (2014)
  - Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, Peter A. Boncz: Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. Proc. VLDB Endow. 11(13): 2209-2222 (2018)
  - André Kohn, Viktor Leis, Thomas Neumann: Adaptive Execution of Compiled Queries. ICDE 2018: 197-208
  - Timo Kersten, Thomas Neumann: On another level: how to debug compiling query engines. DBTest@SIGMOD 2020
  - **Timo Kersten, Viktor Leis, Thomas Neumann: Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. VLDB J. 30(5): 883-905 (2021)**
  - Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, Jana Giceva: Bringing Compiling Databases to RISC Architectures. Proc. VLDB Endow. 16(6): 1222-1234 (2023)