

# Adaptive Execution of Compiled Queries

André Kohn, Viktor Leis, Thomas Neumann

*Technische Universität München*

{kohna, leis, neumann}@in.tum.de

**Abstract**—Compiling queries to machine code is a very efficient way for executing queries. One often overlooked problem with compilation is the time it takes to generate machine code. Even with fast compilation frameworks like LLVM, generating machine code for complex queries often takes hundreds of milliseconds. Such durations can be a major disadvantage for workloads that execute many complex, but quick queries. To solve this problem, we propose an adaptive execution framework, which dynamically switches from interpretation to compilation. We also propose a fast bytecode interpreter for LLVM, which can execute queries without costly translation to machine code and dramatically reduces the query latency. Adaptive execution is fine-grained, and can execute code paths of the same query using different execution modes. Our evaluation shows that this approach achieves optimal performance in a wide variety of settings—low latency for small data sets and maximum throughput for large data sizes.

## I. INTRODUCTION

Compiling queries to machine code has become a very popular method for executing queries. Compilation is used by a large and growing number of commercial systems (e.g., Hekaton [1], [2], MemSQL [3], Spark [4], and Impala [5]) as well as research projects (e.g., HIQUE [6], HyPer [7], DBToaster [8], Tupleware [9], [10], LegoBase [11], ViDa [12], Vodoo [13], Weld [14], Peloton [15], [16]). The main advantage of compilation is, of course, efficiency. By generating code for a given query, compilation avoids the interpretation overhead of traditional execution engines and thereby achieves much higher performance.

One obvious drawback of generating machine code is compilation time. Consider, for example, the following meta data query:

```
SELECT c.oid, c.relname, n.nspname
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhparent
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE i.inhrelid = 16490 ORDER BY inhseqno
```

This query touches only a very small number of tuples, which means that its execution time is negligible (less than 1 millisecond in HyPer). However, before HyPer can execute this query, it needs to compile it to machine code. With optimizations enabled, LLVM takes 54ms to compile this query. In other words, compilation takes 50 times longer than execution. Assuming a workload where similar queries are executed frequently, 98% of the time will be wasted on compilation. And this query is still fairly small; compilation times can be much higher for larger queries. Compilation of the largest TPC-DS query, for example, takes close to 1 second.

Of course, for large data sizes, compilation does pay off as the resulting code is much more efficient than interpretation.

In this work, we focus on database systems that compile queries to LLVM IR (“Intermediate Representation”), which is afterwards compiled to machine code by the LLVM compiler backend. This approach offers the same machine code quality as compiling to C/C++, while reducing compilation time by an order of magnitude [7]. The compilation times of the LLVM compiler may be low enough for some workloads, for example those consisting of long-running ad hoc queries or pre-compiled stored procedures. For other applications, however, long compilation times are a major problem.

The example query shown above is one of the queries sent by the PostgreSQL administration tool *pgAdmin*. On startup, *pgAdmin* sends dozens of complex queries (up to 22 joins), all of which access only very small meta data tables. Compiling these queries causes perceptible and unnecessary delays. Caching the machine code (e.g., after stripping out constants) might improve subsequent executions, but would not improve the initial user experience. More generally, because the human perception threshold is less than a second, the additional latency caused by compilation can lead to a worse user experience for interactive applications. Finally, business intelligence tools occasionally generate extremely large queries (e.g., 1 MB of SQL text), which de facto cannot be compiled with standard compilers.

For the workloads just mentioned, the user experience of a compilation-based engine can be *worse* than that of a traditional, interpretation-based engine (e.g., Volcano-style execution). Thus, depending on the query, one would sometimes prefer to have a compilation-based engine and sometimes an interpretation-based engine. Implementing two query engines in the same system, however, would involve disproportionate efforts and may cause subtle bugs due to minor semantic differences. In this work, we instead propose an adaptive execution framework that is principally based on a single compilation-based query engine, yet integrates interpretation techniques that reduce the query latency. The key components of our design are a (i) fast bytecode interpreter specialized for database queries, (ii) a method of accurately tracking query progress, and (iii) a way to dynamically switch between interpretation and compilation. Without relying on the notoriously inaccurate cost estimates of query optimizers, this dynamic approach enables the best of both worlds: Low latency for small queries and high throughput for long-running queries.

Our adaptive execution framework is directly applicable to many compilation-based systems. Furthermore, our approach

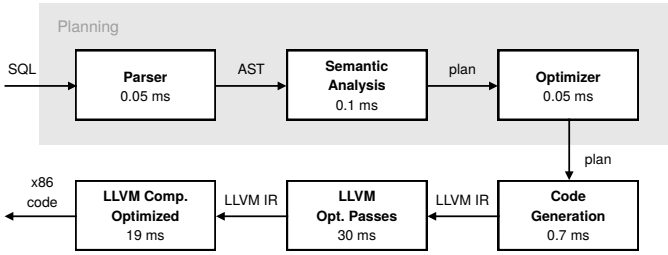


Fig. 1. Architecture of compilation-based query engines.

is non-invasive, i.e., the query engine itself does not have to be rewritten. Our system always generates LLVM IR code for the incoming query but does not immediately compile it to machine code. Rather, it adaptively ensures that the query is executed as fast as possible using runtime feedback and without relying on cost estimates from the query optimizer.

## II. QUERY EXECUTION VIA COMPILATION

Executing a SQL query in a relational database system involves a complex multi-step process that is illustrated in Fig. 1. The SQL text is first parsed into an abstract syntax tree (“Parser”). The AST is translated into an unoptimized query plan (“Semantic Analysis”) that is optimized afterwards (“Optimizer”). Traditional engines directly execute this query plan (e.g., using Volcano-style iteration). Compilation-based engines, in contrast, translate the optimized query plan into some imperative, low-level, machine-independent language (“Code Generation”) that is optimized again (“LLVM Opt. Passes”) and finally compiled to machine code (“LLVM Comp. Optimized”). Some compilation-based systems have multiple intermediate languages between the relational algebra and the low-level imperative representation (LLVM IR in our case). This does not really affect our discussion, as machine code generation generally takes longer than these additional phases. In the following, we describe the major challenges faced by compilation-based query engines that compile to machine code.

### A. Latency vs. Throughput Tradeoff

This paper is based on HyPer, which executes queries by compiling them to the LLVM IR. LLVM is a widely-used open source compiler framework for ahead-of-time and just-in-time compilation. Fig. 1 shows the execution times of each stage for TPC-H query 1 in our system using LLVM. The numbers show that most time is spent in the final two LLVM compiler phases (“LLVM Opt. Passes” and “LLVM Comp. Optimized”), while the preceding code generation, query optimization, and analysis phases are negligible. Therefore, to optimize overall latency, we need to focus on making machine code generation cheaper (or avoid it completely).

Compilation time and execution time differ depending on the compiler and optimization settings used. Fig. 2 shows the compilation and execution time of TPC-H query 1 on scale factor 1 under different settings<sup>1</sup>. As the figure shows, LLVM has a similar throughput as the handwritten C++ query while

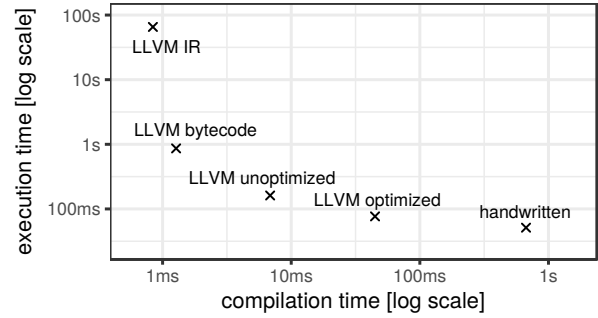


Fig. 2. Single-threaded query compilation and execution time for different execution modes on TPC-H query 1 on scale factor 1.

requiring a much lower compilation time<sup>2</sup>. Disabling all LLVM optimizations results in significantly lower compilation times at the cost of (slightly) higher execution times. The figure also shows the built-in LLVM interpreter (“LLVM IR”), which directly interprets the LLVM IR module, and our bytecode-based interpreter (“LLVM bytecode”), which we describe in Section IV. These numbers show that only interpreters can achieve very low latency but, unsurprisingly, this is achieved by sacrificing throughput.

Fig. 2 clearly shows that there is a tradeoff between latency and throughput. For long-running queries, compilation to machine code with a maximum optimization level is often preferable, while for quick queries an interpreter would be best. Unoptimized machine code lies in between and offers a good tradeoff between these two extremes. Depending on the complexity of the SQL query and the amount of data accessed, different execution schemes are optimal. In this work, we therefore propose to dynamically adapt the query execution by switching between a LLVM bytecode interpreter and the LLVM compiler with optional optimization passes.

Another relevant aspect is that not all code paths of a query are equally important. A query consisting of an in-memory hash join between a very small build relation and a large probe relation, might be best executed by interpreting the hash table build code and compiling the hash probe code. Thus, for different parts of the query, different execution modes can be ideal. Let us also note that compilers are single-threaded, while modern query engines are generally multi-threaded. Thus, while compilation is ongoing, all but one CPU cores are idle, whereas an interpreter could start utilizing all available cores much earlier.

### B. Compiling Large Queries

A compilation time of 59ms for TPC-H query 1 may still be considered low enough for some applications. However, query 1 is still fairly small in terms of its resulting code size and larger queries take much longer to compile. The compilation time of the largest TPC-H and the largest TPC-DS query are 146ms and 911ms respectively.

Furthermore, we observed for very large, machine-generated queries (e.g., from business intelligence tools), that the com-

<sup>1</sup>The experimental setup is described in Section V.

<sup>2</sup>Note that the handwritten version does not implement overflow checks, which explains its slightly faster runtime.

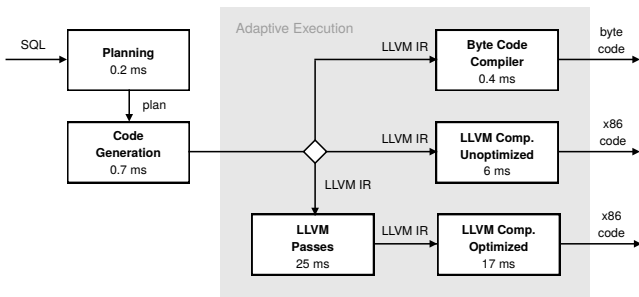


Fig. 3. Execution modes and their compilation times.

pilation times grow super-linearly with the query size. The compilation times may thus become significantly longer and queries may not finish at all (cf. Section V-E). While such queries are not common, any industrial-strength system must be able to execute them—in particular, since traditional database systems do not have this problem.

### III. ADAPTIVE EXECUTION

We argue that compilation-based engines should support the 3 execution modes shown in Fig. 3. Bytecode interpretation enables very low latency for quick queries, unoptimized machine code is a good tradeoff for medium-sized queries, and optimized machine code achieves peak throughput for long-running queries. A system that supports these modes, can provide an optimal user experience if it chooses the right mode for a given query.

One possible way to decide between the different execution modes is to rely on the cost estimates computed by the query optimizer. Cardinality estimates as well as cost models are often inaccurate [17], [18], which may result in unnecessary compilations or long-running queries being executed in the interpreter. The effects of a wrong decision can be severe as the bytecode interpreter can be slower by an order of magnitude and compilation can easily take hundreds of milliseconds. Furthermore, the compilation itself is single-threaded so compiling up-front leaves all remaining threads idle until the compilation is finished.

Our adaptive execution approach is dynamic, as we avoid any up-front decision about the execution mode. Instead, we always start executing every query using the bytecode interpreter and all available threads. We then monitor the execution progress to decide whether (unoptimized or optimized) compilation would be beneficial. If this is the case, we start compiling on a background thread, while the other threads continue the interpreted execution. Once compilation is finished, all threads quickly switch to the compiled machine code. Because all execution modes semantically execute the same instructions on the same data structures, no work is lost when switching between execution modes and the machine code can pick up where the interpreter left off.

Our approach is fine-grained. The tracking and the decision to compile is not done for the entire query, but for a specific query pipeline (e.g., an expensive hash table probe). Therefore, different pipelines might be executed using different execution

modes. This can be better than any static up-front decision because optimized compilation is done only for very expensive parts of the query. As we show later, in a multi-threaded setting, it is also often beneficial to execute the same pipeline by consecutively running all 3 modes.

To implement our dynamic approach we need a number of mechanisms, which we describe in the following three sections. First, it must be possible to track the progress of a pipeline. Second, there must be a way to switch the execution from bytecode to compiled execution without losing any work. Finally, we need a model for deciding whether it is beneficial to switch to compilation.

#### A. Tracking Query Progress

Fig. 4 illustrates the basic code structure for an example query plan and the code structure that our compiler generates. The entry point is the `queryStart` function, which, when called, executes the query. `queryStart` is mainly responsible for calling C++ initialization code and for launching the different pipelines of the query. It can be fairly large in size, but since this code is executed only once, it never pays off to compile it. The actual data-dependent parts of the query are always performed in the `worker` functions, each of which computes the result of one pipeline. The query plan in Fig. 4 consists of 3 pipelines that are processed by the 3 `worker` functions `workerA`, `workerB`, and `workerC`.

This code structure has been chosen with multi-core parallelization in mind. Each `worker` function requires two arguments: the `state` (e.g., intermediate query processing hash tables) and a `morsel`, which determines the range of values to process. Intra-query parallelization is implemented by running multiple `worker` threads on the same `worker` function, but with different (non-overlapping) morsels (e.g., of a relation). In our parallelization framework, the threads use work stealing and the range for each invocation is fairly small (e.g., 10,000 tuples), which avoids thread imbalances. This morsel-wise (or block-wise) execution has been identified as fast model for intra-operator parallelism in main-memory databases [19], [20] and is also just the right granularity at which the progress of a query can be monitored. After each morsel, `worker` threads consult a work-stealing data structure anyway. At this point, we added some extra monitoring code and timing information to keep track of how many morsels have been processed per pipeline. Additionally, we also record the total amount of work whenever a pipeline starts (e.g., the size of the relation or hash table being scanned). This information allows us to monitor the query progress.

#### B. Switching Between Execution Modes

With adaptive execution, a morsel represents the smallest unit of work that can be processed by the query engine. Besides being useful to track the progress of a query, these morsels also emerge as the crucial mechanism for dynamically switching between different execution modes. Processing a single morsel involves reading a specified range of values and operating on a shared state like a hash table. Providing both—range

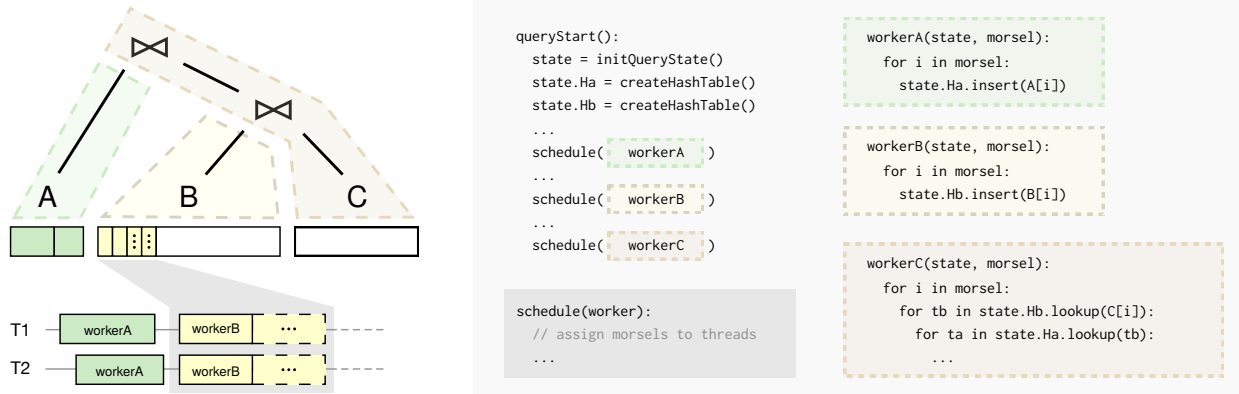


Fig. 4. Illustration of query plan translation to pseudo code. `queryStart` is the main function. Each of the three query pipelines is translated into a `worker` function. The lower left corner shows that the work of each pipeline is split into small morsels that are dynamically scheduled onto threads.

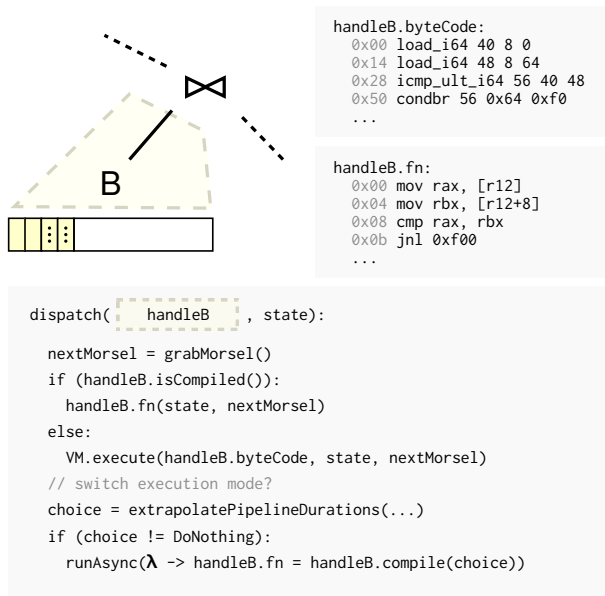


Fig. 5. Switching on-the-fly from interpretation to execution. The dispatch code is run for every morsel.

and shared state—as input parameters to a `worker` function simplifies the generated code and makes the processing of morsels independent from each other.

This independence enables us to choose the execution mode for an individual `worker` arbitrarily. It becomes semantically equivalent to process either all morsels, every second morsel, or no morsel with the bytecode interpreter and process the remainder with a compiled `worker` function. We can further compile a single `worker` function multiple times with different optimization levels to improve the throughput of the function step-by-step.

Fig. 5 illustrates the integration of this concept into the morsel-driven parallelization framework. Instead of identifying a `worker` function by its memory address, we introduce an additional `handle` indirection. This object stores multiple variants of the same function. For every single morsel, we then choose the fastest available representation which could

either be bytecode or an address to compiled machine code. Consequently, to change the execution mode, one only needs to set a function pointer in this `handle` object. Once set, all remaining morsels will be processed using the new variant enabling seamless transitions between execution modes.

### C. Choosing Execution Modes

We have already shown that a higher query throughput comes at the cost of a higher query latency. We therefore always start the execution of each `worker` function with the low-latency bytecode interpreter and compile it only if the need becomes evident. This, however, raises the question of how to determine when compilation is beneficial. To make this decision, we continuously evaluate the following options for every pipeline:

- 1) proceed with the current execution mode
- 2) compile the `worker` function to machine code without compiler optimizations (unoptimized)
- 3) compile the `worker` function to machine code with compiler optimizations (optimized)

Without compilation, the remaining time is entirely based on the current processing speed (i.e., the speed of the bytecode interpreter). We track this speed for every worker thread individually by calculating the local tuple processing rate whenever we finish a morsel. The total pipeline duration can then easily be extrapolated based on the remaining tuples in the pipeline, which is always known at that point in time, and the number of active worker threads. We can further refine this extrapolation by using a dynamically growing morsel size, yielding a higher number of sample points.

With compilation, the remaining time of the pipeline also depends on the expected compilation time as well as an estimate of how much faster the compiled code would be. Another aspect that needs to be incorporated is that, while compilation is ongoing, the execution of the pipeline can continue in a multi-threaded setting (using the bytecode or optimized code). We thus have to compute the tuples that can be processed on the remaining threads during the compilation and extrapolate the time needed for the remainder afterwards.

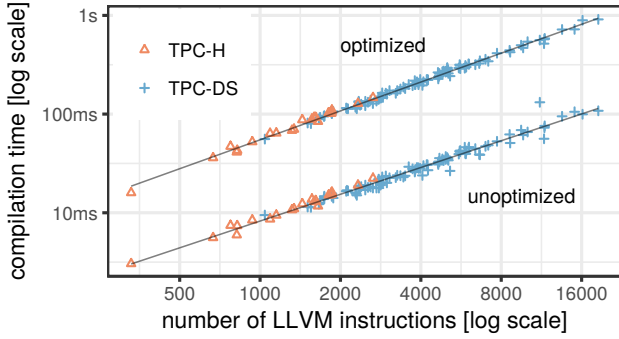


Fig. 6. LLVM compilation time for (un-)optimized machine code for TPC-H and TPC-DS queries.

```
// f: worker function
// n: remaining tuples
// w: active worker threads
extrapolatePipelineDurations(f, n, w):
  r0 = avg(rate in threadRates)
  r1 = r0 * speedup1(f); c1 = ctime1(f)
  r2 = r0 * speedup2(f); c2 = ctime2(f)
  t0 = n / r0 / w
  t1 = c1 + max(n - (w-1)*r0*c1, 0) / r1 / w
  t2 = c2 + max(n - (w-1)*r0*c2, 0) / r2 / w
  switch min(t0, t1, t2):
    case t0: return DoNothing
    case t1: return Unoptimized
    case t2: return Optimized
```

Fig. 7. Extrapolation of the pipeline durations.

The compilation time and the speed-up of a worker function depend on the generated query plan and are determined empirically in our system. As Fig. 6 shows, the number of LLVM instructions of a query correlates very well with its compilation time for all TPC-H and TPC-DS queries. The generated plans contain between 300 and 19,000 instructions for which we observe a near-linear compilation time. For the speed-ups between different execution modes, we use empirical data (cf. Section V-D). While it is generally difficult to forecast accurate query speed-ups and compilation times, adaptive execution only requires rough extrapolations.

Fig. 7 shows the pseudo code for comparing the different execution modes. In order to reduce the synchronization overhead, the extrapolation is only performed by a single worker thread. We delay the first evaluation by 1 millisecond to increase the accuracy of the estimates and reevaluate after every processed morsel thereafter. After every morsel, the thread computes the average processing speed of all threads and compares the remaining processing time of the execution modes. If the transition to a new execution mode appears beneficial, the thread compiles the `worker` function and resets all processing rates. This allows one to eventually transition to the fastest execution mode for every pipeline.

#### IV. FAST BYTECODE INTERPRETATION

As Fig. 2 shows, generating machine code takes a non-trivial amount of time—even without compiler optimizations. An interpreter is therefore a crucial part of our design.

LLVM is a compiler framework that has been designed to generate machine code, but it also contains an interpreter. This interpreter directly executes the LLVM IR without any additional compilation step. Thus, systems that compile to LLVM IR could execute queries using this interpreter. However, as can be seen in Fig. 2, the built-in interpreter is extremely slow (over 800 times slower than the corresponding machine code). The reason is that LLVM IR was designed as a versatile and generic format for implementing optimization passes. Its pointer-based in-memory representation allows easy code transformations but is highly cache unfriendly. Furthermore, the execution of an instruction involves a costly runtime dispatch as there is only a single instruction (e.g., integer addition) for all operand widths (e.g., 8, 16, 32, 64 bits).

To make interpretation a viable strategy, we therefore translate the native LLVM IR into an optimized bytecode format for a virtual machine (VM) that can be interpreted much more efficiently. We have to address two key challenges here: First, processing the bytecode should be as cheap as possible in order to minimize the interpretation overhead. Second, an efficient translation into this bytecode has to be possible. The latter is particularly difficult as many standard compiler techniques like liveness analysis have a super-linear worst-case behavior, yielding unacceptable translation times for very large queries. And finally, the VM must behave 100% identical to native machine code as we want to seamlessly switch between interpreted VM code and native machine code. We therefore developed a virtual machine that mostly follows the LLVM instruction set, but aims for cheap interpretation and offers additional functionality for common constructs.

##### A. Virtual Machine

Our virtual machine is a *register machine*. When calling an interpreted function, we allocate a register file that holds all values computed during function allocation. This allocation happens on the stack if possible, falling back to heap allocation if the register file is too large. For now, we can pretend that every value computed in the LLVM IR has one fixed position in that register file. As we will see in Section IV-C, it is actually undesirable to map values to registers like that, but for now we just assume that all values exist somewhere in the register file. The first two entries in the register file are initialized to 0 and 1, respectively, such that these constants are always readily available in registers.

The instruction set of the VM is *fixed length, statically typed*, and in most places mimics the LLVM IR instruction set. For example, the small LLVM function

```
define i32 @add(i32, i32) {
  %3 = add i32 %1, %0
  ret i32 %3
}
```

will be translated into a very similar VM fragment:

```
add_i32 24 16 20
return_i32 24
```

The `add_i32` instruction loads the two function arguments from

```

while (true) {
  switch ((++ip)->op) {
  case Op::add_i32: *((int32_t*)(regs + ip->a1))=*((int32_t*)(regs + ip->a2)) + *((int32_t*)(regs + ip->a3)); break;
  case Op::add_i64: *((int64_t*)(regs + ip->a1))=*((int64_t*)(regs + ip->a2)) + *((int64_t*)(regs + ip->a3)); break;
  case Op::call_void_i32: (void(*)(int32_t))(ip->lit)(*(int32_t*)(regs + ip->a1)); break;
  ... // around 500 more instructions
  }
}

```

Fig. 8. VM code fragment implementing the interpreter loop. `ip` points to the current instruction and `reg` points to the memory storing the registers.

```

compute liveness and order blocks
for each block b:
  allocate registers for values that become
  live in b
  for each instruction i in b:
    if i is not subsumed:
      translate i into VM opcodes
  propagate values in  $\phi$  nodes
  release register for values that ended in b

```

Fig. 9. Translation of LLVM IR into VM code.

the registers 16 and 20 (which are byte offsets into the register file), and writes the result back into register 24. The `return_i32` instruction returns that value to the caller. Note that there is not always a 1:1 correspondence between LLVM instructions and VM instructions, like in this example. First, the LLVM instructions are annotated with types, while the VM instructions have the type baked into to the opcode itself. For example the LLVM `add` is expanded into different add instructions during translation depending upon the argument types. And second, we sometimes collapse multiple LLVM instructions into one VM instruction to handle frequently occurring instruction sequences (cf. Section IV-F).

We use a fixed length encoding for the opcodes to improve the decoding speed. This increases the memory footprint of the translated function relative to native machine code, but it is still much more compact than the original pointer-heavy LLVM IR. As Fig. 8 shows, the VM code itself then consists of a large switch statement that evaluates all supported instructions.

In total, the VM handles about 500 instruction/type combinations, each consisting of a single and fairly simple line of C++ in the VM code. The bytecode interpreter code is about 800 lines of code, which is surprisingly small for a component that allows us to interpret arbitrary query plans without modifying the query code generation. This is important for the maintainability of the system as it would be highly unattractive to maintain completely separate code paths for both—native code and interpreted execution.

## B. Translating into VM Code

The translation of LLVM IR code into VM code is shown in Fig. 9. It starts by computing the liveness information for register allocation, which is by far the most challenging step of the translation, and therefore discussed below in detail. Afterwards, we know when a value becomes alive within the control flow and when it dies.

With that information, the transformation itself is simple. Note that the transformation exploits the fact the LLVM programs are in Single Static Assignment (SSA) form, i.e., a

value is produced exactly once, and never changes during the lifetime of the program.

We iterate over all basic blocks of the program in the order that the liveness computation has determined. For every block, we then check whether values become alive even though the producing instruction is not contained in the block itself (this is rare, but can happen with a complex control flow). If so, we immediately allocate a register for these values. The instructions within the block are then translated into VM opcodes one by one, except for cases where subsequent instructions are *subsumed* by previous instructions, for example when folding a sequence of instructions into one VM opcode (cf. Section IV-F). At the end of the block we copy values into the  $\phi$  nodes of successor blocks if needed (i.e., if the successor block uses  $\phi$  nodes to unify different values in SSA representation), and release registers for all values where the lifetime has ended. Just as the allocation mentioned before, this is also an exception caused by the control flow, as we will discuss below. For the vast majority of cases we allocate registers on demand and release them when the last user of that value is gone. In summary, we consider block boundaries only when the control flow forces us to extend the lifetime of a value.

After this translation step, the VM program is ready for execution. It performs exactly the same work as the native code would, including all function calls and all memory writes, which is important for the switch between interpretation and compilation. There are some engineering details here to make that substitution possible. Calls to interpreted code, for example, need to be patched during the translation to accept an additional parameter (the VM program). However, that is similar to standard compiler techniques for nested functions and does not introduce too much complexity. Our translator has about 2,400 lines of code most of which are dedicated to the register allocation. As the translation operates almost entirely on the well defined LLVM IR language, the additional engineering effort is not too high.

## C. Register Allocation

As mentioned before, there is only one step during the translation into VM code that is algorithmically challenging, and that is the register allocation, i.e., the mapping of LLVM values to register slots. Our problem differs slightly from traditional register allocation, as we only use virtual registers and therefore could allocate a (nearly) arbitrarily large number of them. However, we clearly do not want to do this: The register file is accessed very frequently during interpretation, and therefore should always be in the L1 cache. A large register file wastes precious L1 cache entries.

Our register allocation problem is therefore the following:

- 1) assign a register slot to every LLVM value in the program
- 2) make sure that a register is only shared between different values if their lifetimes do not overlap
- 3) minimize the total number of registers
- 4) translate very large programs efficiently

In principle, register allocation is a well understood problem in compiler construction [21]. In order to do register allocation we need liveness information, i.e., we have to know for each basic block which values are alive and which are dead. However, computing this liveness information has super-linear runtime in the number of basic blocks, which can make these algorithms prohibitively expensive for large functions. And unfortunately, some of our queries do compile into very large functions with thousands of basic blocks and tens of thousands of values. This is very different from handwritten programs, which tend to consist of small functions. Register allocators try to avoid the expensive liveness computation by splitting the life-ranges via spilling to memory [22]. But this is not really an option for us (in contrast to regular machine code), as we would then have to find a mechanism to minimize the spill region, as that would have to be cache-resident, too. Some JIT systems therefore restrict the register allocation to values within a single basic block (which is easy) or consider only a fixed number of neighboring basic blocks. This approach is computationally simple but can lead to a poor register allocation.

We developed a new linear-time register algorithm that recognizes and utilizes loop structures to quickly approximate the optimal register allocation. Finding the optimal register allocation for a program in SSA form with an unbounded number of registers is super-linear. Instead, our register allocator may sometimes needlessly extend the lifetime of a variable within the bounds of the innermost loop that contains all uses of that value. In practice, however, this only occurs with complex control flows, has imperceptible effects and serves as reasonable trade off for the linear worst-case behavior.

Having a linear runtime algorithm is very important for the adaptive execution framework. As we will see in Section V-E, the regular LLVM compiler is de facto unable to compile some very complicated queries due to the super-linear algorithms used. Indeed, we have encountered machine-generated queries where the largest function consists of 300,000 values and thousands of basic blocks. An algorithm with super-linear runtime for such functions thus leads to unacceptable compile times (hours or even days).

To give an impression of different register allocation strategies, we report the size of the register file for different allocation strategies for the relatively large TPC-DS query 55: If we just allocate values to registers without reuse we need 36 KB, which is larger than our L1 cache. Using a greedy assignment strategy instead where we consider a fixed window of basic blocks for the lifetime, we need 21 KB. This is better and sufficient for some JIT compilers, but still quite large. The algorithm that we present below reduces this number to 6 KB, which is much more reasonable.

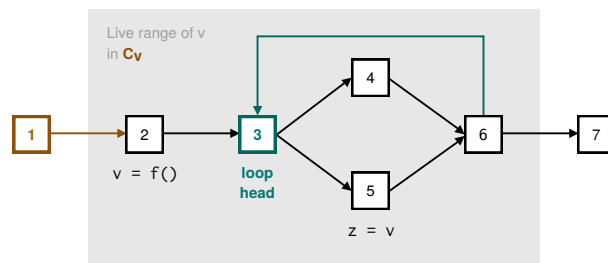


Fig. 10. Computing the liveness of a variable  $x$ . The vertices are basic blocks, which are connected by control flow edges (i.e., branch instructions).

```
// compute the liveness of values in function F
ComputeLiveness(F):
// find loop structures in F
label all basic blocks in F in reverse postorder
compute the dominator tree D for each basic block
label all nodes in D with pre-/postorder numbers
mark the first basic block in F as loop head
for each jump edge j: B → B':
    if B' is ancestor of B in D:
        mark B' as loop head
for each basic block B:
    associate B with the next dominating loop head
for each loop:
    compute the first and last block of the loop
    compute the next dominating loop head
    label loop with nesting depth
// use the loop information to compute lifetimes
for each value v in F
    B_v = set of basic blocks containing
           definition and users of v
    C_v = innermost loop containing all blocks in B_v
    L_v = empty lifetime interval
for each B in B_v:
    if C_v is innermost loop for B:
        extend L_v with B
    else:
        extend L_v with outermost loop below C_v
        that contains B
```

Fig. 11. Linear-time algorithm for liveness computation.

#### D. Linear-Time Liveness Computation

Our algorithm is based upon two key concepts: 1) we compute that liveness of a value as a live-range with a start block and an end block. The traditional method of computing the liveness for each block individually inherently has  $O(n^2)$  runtime. And 2) we keep the live-range of each value as tight as possible by labeling the blocks according to the control flow and by explicitly handling loops. This is illustrated in Fig. 10: The basic blocks in this figure are labeled in *reverse postorder*, which matches the control flow order. The value  $x$  is created in block 2 and consumed in block 5. Naively one could think that the lifetime of  $x$  is therefore the interval  $[2,5]$ , but this is incorrect: Block 5 is part of a loop that starts in 3 and which involves the blocks  $[3,6]$ . Any of these blocks can reach block 5. Therefore, we extend the lifetime to include the *containing loop* of the reader, which results in the life range in  $[2,6]$ .

The full algorithm is shown in Fig. 11. It operates in two phases: In the first phase, it identifies all loops that occur in the function and associates each basic block with the innermost enclosing loop. With this information we can compute the lifetime of a value by identifying all basic blocks that contain definition or uses of a value and lifting these blocks to the level of the innermost loop that contains all blocks. Conceptually,

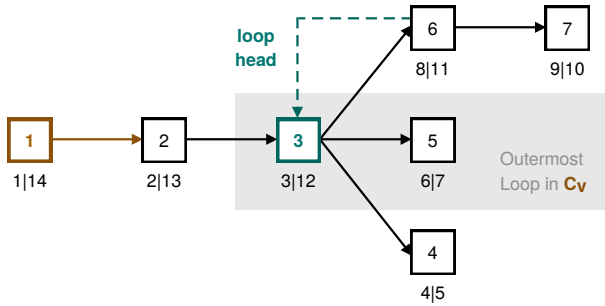


Fig. 12. Dominator tree annotated with pre-/post-order.

a value is alive from its definition to its last user, including all blocks that might be traversed along the way due to loop constructs.

We now look at the algorithm in more detail. It starts by labeling (and ordering) all basic blocks in *reverse postorder*, i.e., a block is placed after all its incoming blocks. Ignoring loops this directly corresponds to the control flow order, and for a human would be the “natural” way to order the blocks in a programming language. This order is required for the next algorithm step, and has the added advantage of making sure that the block labels are meaningful regarding the control flow. Using this labeling, we can compute the dominator tree  $D$  efficiently [23], [24], which, for each basic block, tells us the closest basic block that must have been executed before. For lookup purposes we label all nodes in  $D$  with pre-/post-order numbers [25]. This labeling allows us to determine ancestor/descendant relationships in  $O(1)$ . The dominator tree for our running example is shown in Fig. 12. Using the pre/postorder numbers we can immediately see that, e.g., block 2 transitively dominates block 6, as the interval  $[8,11]$  of block 6 is contained in the interval  $[2,13]$  of block 2.

All this infrastructure is used to identify loops. To avoid edge cases for blocks outside of loops, we pretend that the whole function body is part of one large loop, and we mark the first block of the function as the *loop head* (i.e., the entry point of the loop). Now we look at all jumps between pairs of blocks  $B$  and  $B'$ . If  $B'$  is an ancestor of  $B$  in the dominator tree  $D$ , we have found a loop, and we mark  $B'$  as the loop head. In our example block 6 jumps to block 3, which dominates 6, and thus block 3 is a loop head, i.e., the entry point of a loop. After identifying all loops, we associate each block with their innermost containing loop, represented by the nearest dominating loop head. We use a disjoint set data structure with *path compression* here to make this computation fast. We remember the first and the last block of a loop (according to the block labels), and the loop in which it is nested. In our example the loop starting at block 3 contains the block 3–6, and is contained in the top-level (pseudo) loop starting at block 1. Finally, we compute the nesting depth for each loop.

While this computation is involved, and uses several non-trivial algorithms, the overall complexity of each step is linear. Indeed, most of the complexity stems from the fact that we want to guarantee linear runtime: We could, for example, leave out the pre/post-order labeling or the path-compression, but we

would get super-linear runtime in subsequent steps. The same is true for the choice of the dominator tree algorithms.

Using this loop information, the liveness computation for each individual value  $v$  becomes simple. We identify the set  $U_v$  of all blocks that contain either the definition or the uses of  $v$ . If the containing loop was the same for all these blocks, the lifetime would simply be the span from the first block to the last block, according to the reverse postorder labeling. In the general case, we identify the least common loop  $C_v$  that contains all blocks from  $U_v$ . We extend the lifetime of  $v$  to include the blocks from  $U_v$  within  $C_v$  that are not located within nested loops. For every other block  $b$  in  $U_v$ , we extend the lifetime of  $v$  to include all blocks of the outermost loop within  $C_v$  containing  $b$ . In our example the containing loop  $C_v$  for value  $v$  is the whole function, the definition of  $v$  in block 2 is immediately in that loop, but the use of  $v$  in block 5 is one loop level deeper. As a result, the lifetime of  $v$  is the interval  $[2,6]$ . This whole computation is very cheap due to lookup structures we have prepared while analyzing the loops in the first phase.

Note that some care is required for LLVM’s  $\phi$  nodes: The  $\phi$  nodes are used for the Single-Static-Assignment (SSA) form, and they pick a value depending upon the incoming edge that has led to the basic block with the  $\phi$  node. For the purpose of lifetime computations, the arguments of  $\phi$  are “read” at end of the corresponding incoming block, and the  $\phi$  node is “written” immediately afterwards in the same block, and then “read” in the block that contains the  $\phi$  node. This is not particularly difficult to implement, but one has to keep that in mind when computing the liveness for  $\phi$  nodes.

### E. Interoperability

We interpret the original LLVM IR using our virtual machine. Therefore, our bytecode interpreter behaves equivalently to generated machine code (except for speed differences, of course). This is important, because it allows us to seamlessly switch between interpretation and machine code, without modifying the rest of the system.

The interoperability between bytecode and machine code, however, raises a problem. While a function pointer suffices to run machine code, we need to interpret the bytecode with the virtual machine. So instead of a direct function call we need to call additional dispatch code (cf. Fig. 5) and pass to it the function’s bytecode as an additional argument. We could then differentiate both signatures by tagging the pointer and dynamically call the respective function, but that would be quite invasive and would introduce unnecessary branches. Instead, we always pass an extra pointer argument to the function even though it is redundant in the machine code case. This allows us to transparently switch from interpreted to compiled code by replacing the function pointer and inject the additional argument.

The reverse direction is simpler as we can call existing C++ code from both, generated machine code and from our VM. We just have to make sure that a suitable call instruction is available in our VM for every existing function signature.



Referring to Fig. 8, the opcode `Op::call_void_i32` is required to call C++ functions with a single 32 bit integer parameter and no return value. As we know all exported C++ functions, we can identify missing opcodes at compile time.

#### F. Optimizations

While being possible, it is sometimes inadvisable to translate LLVM instructions independent from each other. One example for this is overflow checking. Any arithmetic that occurs within a query is checked for overflows in order to report overflow errors to the user. With LLVM, this check boils down to 4 instructions that are always executed in sequence. With the bytecode interpreter, our translator recognizes this sequence, and replaces it with a single VM bytecode that performs all four steps at once. This greatly reduces the number of instructions for some queries and decreases their execution time.

Another frequently occurring pattern is the `GetElementPtr` (i.e., pointer arithmetic) instruction followed by a load or store. These sequences are also recognized during the translation and merged into one VM opcode to reduce the instruction count.

In general, it would make sense to translate a large corpus of queries, and to check for frequently occurring sequences of instructions in order to replace them by macro instructions. One candidate for that could, for example, be NULL handling, which also tends to create similar instruction sequences. In future work, we will expand this mechanism to recognize more of these constructs.

### V. EVALUATION

In this section, we experimentally compare the adaptive query execution framework discussed in Section III with different statically chosen execution modes. We also devote special attention on the bytecode interpreter introduced in Section IV to answer the question whether query interpretation adds additional value to compilation-based databases.

Our experiments are performed in HyPer, a database system that directly generates LLVM IR, and, so far, always compiled it to machine code. By default, *optimized* compilation was used, which enables all machine-specific (backend) optimizations after executing a number of hand-picked LLVM IR optimization passes (peephole optimizations, reassociate expressions, common subexpression elimination, control flow graph simplification, aggressive dead code elimination). We also implemented an *unoptimized* compilation mode, which also generates machine code but disables most compiler optimizations to improve compile times. Specifically, this mode enables fast instruction selection, does not execute any LLVM IR optimization passes, and uses a low backend optimization level. Our *interpreter* translates the LLVM IR directly into the bytecode discussed in Section IV. Finally, the *adaptive* execution mode interleaves machine code generation and execution as described in Section III.

The experiments have been performed on a desktop system with an 8 core AMD Ryzen 7 1700X CPU, 32 GB of RAM, LLVM 3.8 and Linux 4.11. We repeated all experiments on an Intel CPU and observed similar results.

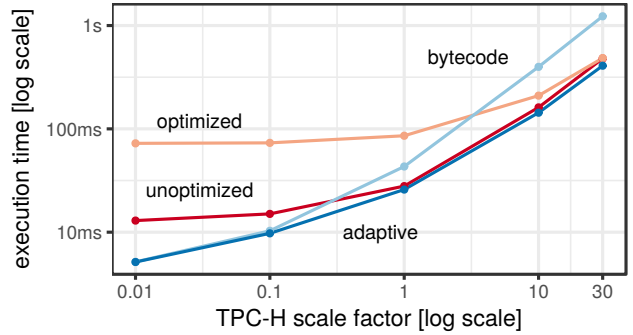


Fig. 13. Geometric mean of all TPC-H queries including planning, compilation, and execution using 8 threads for different scale factors and execution modes.

#### A. Static vs. Adaptive Mode Selection

Let us first investigate, whether an adaptive switching of the execution mode can compete with a static up-front decision. In this experiment, we run all 22 TPC-H queries on scale factors ranging from 0.01 (around 10 MB) to 30 (around 30 GB).

Fig. 13 presents the geometric means over all queries and for all execution modes. Without having prior knowledge about the exact data size, adaptive execution is able to always compete with the best statically chosen execution mode. For the scale factors 0.01 and 0.1, the superior strategy is determined solely by the query latency which clearly favors interpretation over compilation. At these data sizes, adaptive execution never chooses to compile and performs just as well as pure bytecode interpretation. Starting from scale factor 1, it becomes viable to compile many of the pipelines, making unoptimized compilation competitive. However, adaptive execution is still able to outperform unoptimized compilation as fast pipelines can still be processed as bytecode. Finally, at scale factor 30 the queries run long enough to justify the optimized compilation. Adaptive execution now picks the best out of three execution modes per pipeline and outperforms both compilation modes noticeably. At even larger scale factors, we expect this trend to continue, with optimized compilation becoming the main competitor for adaptive execution. However, we also expect that adaptive execution will continue to have the overall lowest processing time as there will still be cheap pipelines in the query plans, that can be executed immediately.

#### B. Adaptive Execution in Action

In a next step, we investigate the adaptive behavior of the framework based on TPC-H query 11 on scale factor 1 using 4 threads<sup>3</sup>. We compare adaptive execution with its competitors using the dynamic execution trace shown in Fig. 14 which shows precise timing information about the morsels being processed. Starting with the bytecode interpreter, the figure shows that the database quickly uses all 4 worker threads to process the pipeline morsels in parallel. It also reveals that the amount of work is distributed very unequally among the 7 pipelines and that most of the time is spent

<sup>3</sup>Query 11 has been chosen such that the individual morsels are graphically distinguishable.

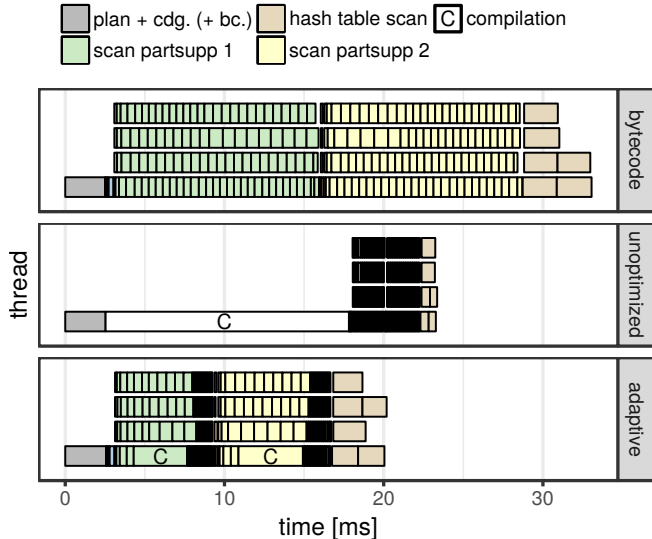


Fig. 14. Execution trace of TPC-H query 11 on scale factor 1 using 4 threads. The optimized mode is not shown, as its compilation takes very long (103ms).

on the processing of the pipelines “scan partsupp 1” and “scan partsupp 2”. Unoptimized compilation, in contrast, uses a significant proportion of the time for the initial single-threaded compilation of the query plan. Afterwards, the morsels can hardly be distinguished from each other as the processing with compiled pipelines is much faster. The execution trace of optimized compilation looks very similar to the one from unoptimized compilation but is not shown for graphical reasons, as the additional compiler optimizations lengthen the compilation time to 103 milliseconds. In summary, unoptimized compilation dominates the other statically chosen execution modes for this query (at scale factor 1) due to being a good tradeoff between an increased efficiency and a fast query preparation. These observations already indicate, that the quality of a static up-front decision highly depends on the complexity of individual pipelines and the data that is being processed.

Fig. 14 also shows the execution trace of our adaptive execution mode which is able to outperform all of its competitors. Very similar to the pure bytecode interpreter, adaptive execution can immediately start to process the pipeline morsels on all 4 worker threads. After 1 millisecond, it determines for the two largest pipelines that switching the execution mode is worthwhile and therefore dedicates a worker thread to compile them. As the compilation is restricted to a single function, it only takes a fraction of the time we observed when transforming the whole query plan. Once compiled, all worker threads automatically shift gear to the newly-created machine code and process the remaining morsels very efficiently. However, the unequal complexity distribution favors the compilation of only 2 out of 7 pipelines. Thus our framework processes the remaining pipelines using the bytecode interpreter and finishes the query 10%, 40% and 80% faster than the competitors (i.e., unoptimized compilation, bytecode interpreter and optimized compilation).

TABLE I  
PLANNING AND COMPILATION TIMES IN MS FOR TPC-H QUERIES ON POSTGRESQL (“PG”), MONETDB (“MONET”), AND HYPER.

TPC-H #	plan		HyPer				
	PG	Monet	plan	cdg.	bc.	unopt.	opt.
1	0.1	0.8	0.2	0.7	0.4	6	42
2	1.0	0.7	0.7	1.5	1.2	23	149
3	0.3	0.5	0.4	0.9	0.7	10	69
4	0.2	0.4	0.2	0.7	0.4	7	47
5	1.2	0.8	0.7	1.2	0.9	15	104
max	1.9	1.0	0.8	1.5	1.2	23	149

TABLE II  
EXECUTION TIMES OF TPC-H QUERIES ON SCALE FACTOR 1 ON POSTGRESQL (“PG”), MONETDB (“MONET”) AND HYPER. THE GEOMETRIC MEANS (“GEO.M.”) ARE OVER ALL 22 QUERIES.

TPC-H #	1 thread					8 threads		
	PG	Monet	bc.	unopt.	opt.	bc.	unopt.	opt.
1	4908	484	858	161	77	170	34	16
2	254	5	94	13	8	25	5	3
3	1258	64	323	104	80	54	21	17
4	193	56	352	67	45	57	16	12
5	516	51	362	60	37	67	14	10
geo.m.	497	57	232	60	46	45	15	12

### C. Planning and Compilation Time

Bytecode interpretation is a viable approach to provide a low-latency execution mode in compilation-based databases. In order to provide evidence for this statement, we evaluate the planning and compilation times of HyPer and compare them with PostgreSQL 9.6, which uses Volcano-style interpretation, and MonetDB 1.7, which uses column-at-a-time processing. Table I shows the planning times for TPC-H queries 1 through 5 and the maximum over all 22 queries. For TPC-H, plan generation (labeled as “plan” in the table), which includes parsing, semantic analysis, and query optimization, is very fast in all systems. While MonetDB and PostgreSQL can directly execute this plan, HyPer generates LLVM IR code in the code generation phase (abbreviated as “cdg.” in the table). LLVM IR generation typically takes slightly longer than planning, but is still very fast (less than 2ms over all 22 TPC-H queries). The next phase in HyPer is either bytecode (“bc.”), unoptimized (“unopt.”), or optimized (“opt.”) machine code generation. The table shows, that even unoptimized machine code compilation is generally around 10x slower than planning and code generation. Optimized compilation is even slower and takes up to 150 ms for TPC-H. Bytecode generation, on the other hand, is very fast and is always finished in less than 2 ms.

### D. Performance of Interpreted and Compiled Code

Let us next compare the execution times of the bytecode interpreter and the compiled machine code. Table II shows the TPC-H performance on scale factor 1 for the different execution modes and compares them with MonetDB as well as PostgreSQL. Considering the geometric mean across all

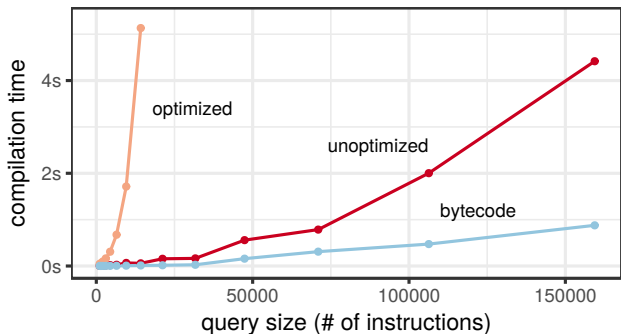


Fig. 15. Compilation times of queries with a large number of instruction using optimized compilation, unoptimized compilation and interpretation.

22 queries, the bytecode interpreter is 3.6 times slower than unoptimized machine code and 5.0 times slower than optimized machine code. While interpreted code is slower than compiled code, it is still 2.1 times faster than PostgreSQL and scales just as well as compiled code when multiple cores are used.

### E. Compiling Very Large Queries

In Section III, we introduced a linear cost function that estimates the compilation time based on the number of instructions in the pipeline. However, we derived this function from the TPC-H and TPC-DS benchmarks which do not contain particularly complex queries. Machine-generated queries, on the other hand, can easily comprise multiple MB of SQL text with very unpleasant properties for the query compiler. In our last experiment, we therefore investigate the effects of very large queries on our three execution modes and show that fast translation into bytecode is indispensable for these workloads.

Our sample queries consist of a single table scan and an increasing number of aggregate expressions. By scaling this number from 10 to 1900, we receive query plans that contain between 1,000 and 160,000 LLVM instructions, most of which are in a single large function. Fig. 15 shows the compilation times of these queries with the different execution modes. Above all, the measurements show that optimized LLVM compilation is no longer a viable approach for larger query sizes. Its compilation times are characterized by an explosive growth and exceed the 4 seconds mark already for 10,000 LLVM instructions. Without optimization passes, the query compilation scales better but still requires 4.4 seconds for the largest of our queries. In comparison, the bytecode interpreter scales perfectly and is able to process this very large query in only 0.9 seconds. This workload stresses the importance of the fast bytecode translation that we introduced in Section IV. The translation allows us to execute queries of (almost) arbitrary size with the interpreter and adaptively compile parts of the query whenever efficiency is needed.

## VI. RELATED WORK

Many papers on compilation-based query processing only report execution times without stating the time it takes to generate the machine code itself. Those papers that do report them, show compilation times between 5ms and 37ms when LLVM IR is used as a target language [26] and closer to

1 second for compilation to C [11], [27]. As compilation is becoming widespread, we expect that compilation times will receive more attention as any industrial-strength system must deal with very large queries. Indeed, our personal experience has been that after transitioning from standard benchmark queries, which are usually well-designed and “sane”, to real-world customer queries, which are sometimes very “interesting”, query compilation latency becomes a major problem. We therefore believe that adaptive execution is a crucial component for making query compilation truly practical—in particular since traditional engines and modern columns stores (e.g., [28], [29], [30], [31]) do not have large compilation times. In the following, we describe how adaptive compilation can be integrated into other systems, and discuss other approaches for reducing compilation time.


Adaptive execution has been designed for systems that directly compile queries to LLVM IR, which includes HyPer [7] and Peloton [15]. MemSQL is another system that is based on compilation that would benefit from adaptive execution. It originally compiled queries to template-heavy C++, which resulted in very high compilation times. Likely for this reason, recent versions compile to a high-level imperative language called MemSQL Plan Language, which is then lowered down via a mid-level intermediate language called MemSQL Bit Code to LLVM IR [3]. Since the MemSQL Bit Code can be interpreted, switching between interpretation and execution could easily be implemented at that level. A similar approach like adaptive execution could also be applied to systems like LegoBase [11] and its successor system [27], both of which can either execute queries through the Java VM or by compiling to a low-level language like C. Adaptive execution might also be useful for traditional (e.g., Volcano-style) systems that use compilation to specialize the query engine code for a particular query [32], [33], [5]. Microsoft Hekaton, which is part of SQL Server, compiles stored procedures to C [2]. For this use case, compilation times are arguably less important than for ad hoc queries because stored procedures are generally defined infrequently, but executed often.

Automatic plan caching, i.e., reusing query plans between subsequent executions of the same (or a similar) query, is another, orthogonal approach for reducing compilation times. However, plan caching, like explicit prepared statements, cannot hide the compilation time of the first incoming query. For interactive applications this means that the initial user experience of compilation-based systems is far from ideal. Another disadvantage of plan caching is that recurring queries are often not exactly the same, but, e.g., differ by the selection constants. Our adaptive approach can re-optimize queries on every execution, which has the advantage that the specific query constants are visible to the query optimizer, potentially leading to better query plans. Nevertheless, it would also be possible to combine our approach with plan caching. Indeed, one could extend adaptive execution to incorporate multiple executions of the same query by keeping track of how often each pipeline is executed. In this design, eventually all pipelines of frequently-executed queries would be compiled with optimizations.

Adaptive execution bears similarities with the execution engines of modern managed languages like Java (HotSpot), C# (CLR), and JavaScript (V8, JägerMonkey). These systems initially execute code in an interpreter and then, for hot code, dynamically switch to compilation. Our adaptive execution framework can be considered a database-specific implementation of similar ideas. However, for maximum performance, database systems require precise control over memory management and are therefore generally written in (or generate) low-level languages. Therefore, databases cannot use automatic solutions at the language level, which, to the best of our knowledge, only exist for managed languages. On the other hand, in contrast to a general-purpose programming language, a database system knows much more about the code structure and the instructions generated. This simplifies the design and implementation of adaptive execution (e.g., we do not implement LLVM IR instructions that we do not generate) and allows database-specific optimizations (e.g., macro operations for common operations like overflow checking).

## VII. SUMMARY

We showed that interpretation and compilation are both important building blocks for achieving low query latency and high throughput. We also presented an adaptive execution framework that dynamically and automatically adjusts the execution mode of a query to minimize its overall execution time. In this approach, all decisions are made at a pipeline granularity and are based on runtime feedback instead of having to decide up-front. We further proposed a bytecode interpreter that features a linear-time translation of LLVM IR into efficient bytecode. Using this interpreter and the existing LLVM compiler with optional optimization passes, our system was able to dynamically adapt to data sizes ranging from 10MB to 30GB and outperform all statically chosen execution modes for queries in the TPC-H benchmark.

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286). 

## REFERENCES

- [1] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, “Hekaton: SQL server’s memory-optimized OLTP engine,” in *SIGMOD*, 2013, pp. 1243–1254.
- [2] C. Freedman, E. Ismert, and P. Larson, “Compilation in the Microsoft SQL Server Hekaton engine,” *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 22–30, 2014.
- [3] D. Paroski, “Code generation: The inner sanctum of database performance,” <http://highscalability.com/blog/2016/09/07/code-generation-the-inner-sanctum-of-database-performance.html>, 2016.
- [4] S. Agarwal, D. Liu, and R. Xin, “Apache Spark as a compiler: Joining a billion rows per second on a laptop,” <https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html>, 2016.
- [5] S. Wanderman-Milne and N. Li, “Runtime code generation in Cloudera Impala,” *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 31–37, 2014.
- [6] K. Krikellias, S. Viglas, and M. Cintra, “Generating code for holistic query evaluation,” in *ICDE*, 2010, pp. 613–624.
- [7] T. Neumann, “Efficiently compiling efficient query plans for modern hardware,” *PVLDB*, vol. 4, no. 9, 2011.
- [8] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha, “DBToaster: higher-order delta processing for dynamic, frequently fresh views,” *VLDB J.*, vol. 23, no. 2, pp. 253–278, 2014.
- [9] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik, “Tupeware: “big” data, big analytics, small clusters,” in *CIDR*, 2015.
- [10] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik, “An architecture for compiling UDF-centric workflows,” *PVLDB*, vol. 8, no. 12, pp. 1466–1477, 2015.
- [11] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi, “Building efficient query engines in a high-level language,” *PVLDB*, vol. 7, no. 10, pp. 853–864, 2014.
- [12] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki, “Just-in-time data virtualization: Lightweight data management with ViDa,” in *CIDR*, 2015.
- [13] H. Pirk, O. Moll, M. Zaharia, and S. Madden, “Voodoo - a vector algebra for portable database performance on modern hardware,” *PVLDB*, vol. 9, no. 14, pp. 1707–1718, 2016.
- [14] S. Palkar, J. J. Thomas, A. Shanbhag, M. Schwarzkoft, S. P. Amarasinghe, and M. Zaharia, “A common runtime for high performance data analysis,” in *CIDR*, 2017.
- [15] P. Menon, T. C. Mowry, and A. Pavlo, “Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last,” *PVLDB*, vol. 11, no. 1, 2017.
- [16] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang, “Self-driving database management systems,” in *CIDR*, 2017.
- [17] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, “How good are query optimizers, really?” *PVLDB*, vol. 9, no. 3, 2015.
- [18] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, “Query optimization through the looking glass, and what we found running the join order benchmark,” *VLDB J.*, 2018.
- [19] C. Chasseur and J. M. Patel, “Design and evaluation of storage organizations for read-optimized main memory databases,” *PVLDB*, vol. 6, no. 13, pp. 1474–1485, 2013.
- [20] V. Leis, P. Boncz, A. Kemper, and T. Neumann, “Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age,” in *SIGMOD*, 2014, pp. 743–754.
- [21] F. M. Q. Pereira, “The design and implementation of a SSA-based register allocator,” 2007.
- [22] M. Poletto and V. Sarkar, “Linear scan register allocation,” *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999.
- [23] L. Georgiadis, R. E. Tarjan, and R. F. F. Werneck, “Finding dominators in practice,” *J. Graph Algorithms Appl.*, vol. 10, no. 1, pp. 69–94, 2006.
- [24] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan, “Finding dominators via disjoint set union,” *J. Discrete Algorithms*, vol. 23, pp. 2–20, 2013.
- [25] T. Grust, “Accelerating XPath location steps,” in *SIGMOD*, 2002, pp. 109–120.
- [26] T. Neumann and V. Leis, “Compiling database queries into machine code,” *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 3–11, 2014.
- [27] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch, “How to architect a query compiler,” in *SIGMOD*, 2016, pp. 1907–1922.
- [28] P. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surma, and Q. Zhou, “SQL server column store indexes,” in *SIGMOD*, 2011, pp. 1177–1184.
- [29] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, “The design and implementation of modern column-oriented database systems,” *Foundations and Trends in Databases*, vol. 5, no. 3, pp. 197–280, 2013.
- [30] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang, “DB2 with BLU acceleration: So much more than just a column store,” *PVLDB*, vol. 6, no. 11, pp. 1080–1091, 2013.
- [31] J. Patel, H. Deshmukh, J. Zhu, H. Memisoglu, N. Potti, S. Saurabh, M. Spehlmann, and Z. Zhang, “Quickstep: A data platform based on the scaling-in approach,” University of Wisconsin - Madison, Tech. Rep., 2017.
- [32] R. Zhang, R. T. Snodgrass, and S. Debray, “Micro-specialization in DBMSes,” in *ICDE*, 2012, pp. 690–701.
- [33] R. Zhang, S. Debray, and R. T. Snodgrass, “Micro-specialization: dynamic code specialization of database management systems,” in *International Symposium on Code Generation and Optimization*, 2012, pp. 63–73.