Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra

Timo Kersten · Viktor Leis · Thomas Neumann

Received: date / Accepted: date

Abstract Although compiling queries to efficient machine code has become a common approach for query execution, a number of newly-created database system projects still refrain from using compilation. It is sometimes claimed that the intricacies of code generation make compilation-based engines too complex. Also, a major barrier for adoption, especially for interactive ad-hoc queries, is long compilation time.

In this paper, we examine all stages of compiling query execution engines and show how to reduce compilation overhead. We incorporate the lessons learned from a decade of generating code in HyPer into a design that manages complexity and yields high speed. First, we introduce a code generation framework that establishes abstractions to manage complexity, yet generates code in a single fast pass. Second, we present a program representation whose data structures are tuned to support fast code generation and compilation. Third, we introduce a new compiler backend that is optimized for minimal compile time, and simultaneously, yields superior execution performance to competing approaches, e.g., Volcano-style or bytecode interpretation.

We implemented these optimizations in our database system Umbra to show that it is possible to unite fast compilation and fast execution. Indeed, Umbra achieves unprecedentedly low query latencies. On small data sets, it is even faster than interpreter engines like DuckDB and PostgreSQL. At the same time, on large data sets, its throughput is on par with the state-of-the-art compiling system HyPer.

Keywords Relational Query Execution · Code Generation · Low Latency

1 Introduction

Query compilation is a widely adopted approach for relational database systems [1, 7, 10, 34, 46]. Creating machine code for every query removes interpretation overhead and allows the database system to extract the highest performance from the underlying hardware. So far, high processing performance was most relevant in the field of in-memory databases [5, 6, 13, 15, 17, 20, 23, 25, 30, 31, 36]. Yet, the growing bandwidth capabilities of solid state drives and non-volatile memory (also with large bandwidth) make query compilation attractive for a growing field of hardware configurations [11, 27, 40].

Compilation works well for large analytical workloads. However, for some use-cases the extra time spent on compilation—the latency overhead of compilation—can be a problem. For example, interactive data exploration tools send many queries to the underlying database system; often even multiple queries for a single user interaction. Any overhead from compilation delays the query response and, especially with a large number of queries per interaction, becomes noticeable to the user and causes them to idly wait. Vogelsgesang et al. reported that for the interactive data exploration tool Tableau some queries, even after careful tuning, still take multiple seconds just in compilation step of the
underlying database system Hyper [45]. The Northstar project also encountered the issue. They observed that compilation "has an up-front cost, which can quickly add up" [19] and thus severely deteriorates the interactive user experience.

This paper presents multiple components for compiling query engines to achieve low query latency: that is, to minimize the total time spent for query compilation and execution. Compile time must be addressed in the whole compilation pipeline, thus we address every component (c.f., Figure 1). We introduce ① Tidy Tuples, a fast code generation framework, ② Umbra IR, an efficient program representation, and ③ Flying Start, a compiler to quickly generate machine code. All components are integrated into the database system Umbra [27] and our experiments show that together, they effectively reduce compilation time and maintain high query execution speed.

The first step toward low query latency is a fast code generator. We present ① the Tidy Tuples relational code generator framework. It lowers algebraic operators to Umbra IR in a single pass for low compilation time and in certain cases utilizes pre-compiled code to avoid compilation time altogether. Tidy Tuples is a latency-streamlined design that achieves code generation up to three orders of magnitude faster than competitors (e.g., 1000× faster than LB2 [44]) while still providing a clean, type-safe, and easy to understand interface.

The question of how to build a code generator is not yet settled [42–44], as the generator must handle the complexity of relational operators, many SQL types, NULL values, and much more. To handle complexity, a code generator should adhere to the principles of good software engineering. Tahboub et al. found an elegant way to achieve this. With the LB2 system [44] they built a well-architected query interpreter in Scala. The interpreter is based on the data-centric model, but uses callback functions to structure communication between operators. Employing callback functions is a structural advancement that provides the data-centric model with the clear structure of Volcano-style interpreters. Through extensions in the Scala compiler they are able to transform this interpreter into a code generator so that they get a system with a type safe, easy to read, well-architected code generator.

Unfortunately, the LB2 approach requires very long code generation times, which add to query latency. It fundamentally limits query execution speed to three queries per second. The authors report 299 ms for code generation geometric mean over all TPC-H queries, and that is even before the compiler started generating machine code, so the approach is not viable for low query latencies. We transfer the essence of the LB2 code generator architecture to the systems language C++ and into our Tidy Tuples design. This way, Tidy Tuples obtains a clear structure, yet achieves code generation more than 1000× faster. Additionally, we contribute abstractions on top of the code generator that decompose all issues of code generation into a layered structure.

The next component for low query latency is ② Umbra IR, a custom intermediate program representation. It is modelled after LLVM’s intermediate representation, but its data structures are optimized for writing and reading speed. Tidy Tuples uses Umbra IR as target for the code generator and source for all compilation backends. This reduces the time to generate programs and to transform them to executables. An alternative, the commonly used intermediate representation from the LLVM compiler framework, is expressive and agile. In the compiler framework, it is used as the common format which all optimization passes edit during compilation. However, we found that its flexibility is counter-productive for query latency. Therefore, with Umbra IR, we trade off the ability to arbitrarily transform programs for optimal writing and reading speed.

Lastly, we introduce ③ the novel Flying Start compilation backend which transforms Umbra IR directly into machine-code. Flying Start reduces query latency in two ways: It minimizes time spent for compilation as it generates machine-code very quickly. Further, it reduces the time spent for execution as the speed of the created machine-code is close to that of thoroughly optimized code. The Flying Start backend is integrated into Umbra through the adaptive execution technique [18]. This allows Umbra to switch dynamically between low-latency compilation with Flying Start and highest-speed query execution.
Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra

1

Umbra and HyPer use a single thread for compilation and multiple threads for query execution.

Adaptive execution was introduced first to the HyPer query engine. For query execution it has a choice between using intensively optimized code for high-speed execution and two low-latency compilation backends. For low latency, HyPer can either use a bytecode interpreter or the optimizing compiler LLVM with most optimizations turned off (turning optimizations on takes too much compilation time for short-running queries). In the example of TPC-H query 2, HyPer’s low-latency choices are the top two in Figure 2. It can either prioritize fast execution, but spend more time in compilation with the LLVM backend, or use the bytecode interpreter for fast compilation at the cost of slower execution. Unfortunately, in cases like this, both options have significant shortcomings: Compilation time with LLVM is not amortized and the bytecode interpretation is so slow that it diminishes the gains from its fast compilation. Ultimately, the query engine is stuck in a performance gap between interpretation and compilation, with no great choice for low query latency.

With the Flying Start backend we show a solution for the low-latency spectrum, i.e., short-running queries. It generates code even faster than HyPer’s bytecode interpreter and the resulting execution speed is on par with HyPer’s LLVM-generated code. The Flying Start compilation backend, thus, is able to capture the best of both worlds: It combines great compilation speed with great execution speed. Effectively, it closes the performance gap between the two execution options and therefore offers much lower query latencies than previous approaches.

Tidy Tuples, Umbra IR, and the Flying Start backend represent the foundation of our new database system Umbra. Together, these three components achieve query latencies for short-running queries that previously were only possible using interpretation. Overall, experimental results show that the triad is so effective at reducing latency that Umbra reaches the latency realms of interpretation-based engines like DuckDB and PostgreSQL, all while keeping the execution speed of state-of-the-art compiling systems like HyPer for long-running queries.

The paper is organized as follows: Section 2 explains the code generator and the Tidy Tuples design. Section 3 details how our custom intermediate representation aids fast code generation. Section 4 outlines the Flying Start compiler and how it achieves low compilation time. The impact of these latency optimizations on the performance of Umbra is evaluated experimentally in Section 5. Section 6 discusses related work, and Section 7 summarizes the main results of this work.

2 Tidy Tuples: A Low-Latency Code Generation Framework

The initial component important for low latency is the code generator—the component that lowers relational plans to imperative programs. For maximum speed, we propose to create programs in a single pass over the input. Unfortunately, performance optimizations are often at odds with principles of good software engineering, e.g., separation of concerns, readability, extensibility, and accessibility to newcomers. This also applies to building a SQL database system. Such a system must be able to handle arbitrarily complex SQL queries, handle many SQL types, and cope with the intricacies of NULL values. These requirements are already complex, but paired with the need to optimize for speed one can quickly clash with software engineering principles.

In this section we present our Tidy Tuples design for a relational code generator. It caters to the need for speed, but also provides structure to adhere to principles of good software engineering. Tidy Tuples is a toolbox of complementary components that are organized into layers. It is a solid base to implement relational operators that are easy to read and achieve fast execution.

To introduce the architecture, we first take a short look at the life of a query within a compiling database system in the following Section 2.1. Section 2.2 starts with an overview of the layers in the toolbox and their contents. In Section 2.3, we demonstrate the layers using a short example—peeling off abstractions step by step to provide insight into how the layers fit together. This section shows most clearly how the query plan
is conceptually lowered in multiple steps. Finally, we discuss some important details, including the SQLValue abstraction in Section 2.4 and the low-level code generator interface in Sections 2.5, 2.6, and 2.7.

2.1 Background: Compilation Pipeline

Let us first give an overview of the life of a SQL query inside a compiling DBMS, using the system Umbra as an example. A query is parsed to an abstract syntax tree, which is then semantically analyzed and translated to relational algebra. The query optimizer takes the relational algebra tree and creates an optimized physical plan. The plan describes how to process data to obtain the result. All steps described up to here are commonly found in any relational database system. Only the following steps are specific to compiling query execution engines. From the optimized physical plan, the code generator must create a program so that the execution of the program produces the query result. To see an illustration of the process, find the query plan in the top left corner of Figure 4.

Tidy Tuples translates the physical plan operator by operator. It instantiates an operator translator for each algebraic operator which is responsible for generating code that will execute its algebra operator. Conceptually, operator translators get tuples from their child operators and pass control to each other following the produce/consume interface [25]. During this translation, every translator appends instructions to a program. Ultimately, all operator translators together create a program that will produce the query result. Umbra represents these programs in a custom intermediate representation called Umbra IR (see Section 3).

There are two options for converting a program from Umbra IR into an executable: The low-latency Flying Start backend (see Section 4) or the LLVM-based optimizing backend [21]. Both produce machine code which computes the query result when executed. Of all the steps involved in this process, the Tidy Tuples framework focuses on translating algebraic execution plans into IR in a clear, modular, and maintainable fashion. The remainder of this section explains in detail how Tidy Tuples structures the code generator.

For the description of the Tidy Tuples framework, it is important to differentiate between the two compilation phases “system compile time” and “query compile time” (c.f., Figure 3). At system compile time the source code of the DBMS is compiled into an executable (DBMS) binary. A user can start that binary to run the system. During runtime the system accepts SQL queries, compiles a binary for each query, and runs the query-specific binary to obtain the query result. The distinction between system compile time and query compile time is important to differentiate between the two compilation phases “system compile time” and “query compile time” (c.f., Figure 3). At system compile time the source code of the DBMS is compiled into an executable (DBMS) binary. A user can start that binary to run the system. During runtime the system accepts SQL queries, compiles a binary for each query, and runs the query-specific binary to obtain the query result. The distinction between system compile time and query compile time is relevant for the description of the Tidy Tuples compilation framework. For example, Tidy Tuples relies on the C++ type system to ensure correctness of code generated at query compile time. Naturally, correctness checks within the C++ type system already happen at system compile time (thus produce no overhead at query compile time).

2.2 Layer Overview

The components of Tidy Tuples are arranged into the five logical layers shown in Figure 4. Each layer acts as a level of abstraction and can use the tools of lower layers to implement its functionality so that conceptually a query plan is lowered through the layers.

- **Operator Translators**: The top-most layer contains algebra operator translators which coordinate in produce-consume style [25].
- **Data Structures**: To handle algorithmic challenges, operator translators use components from the data structure layer, e.g., hash tables (i.e., components that generate code to act on hash tables).
- **Tuples**: The tuples layer provides operations that work on multiple SQL values, e.g., packing tuples into a memory efficient format and hashing.
- **SQL Values**: Operators use the SQL value layer to implement SQL data-type specific parts in which operations on SQL types are performed. These operations include addition, substring search, equality comparison, and many more. Furthermore, the SQL value layer offers tools to operate on SQL values with standard-conform NULL-semantics.
- **Codegen API**: All these layers directly or indirectly use the Codegen layer to append instructions to the output program. Codegen offers operations on low-level types which are close to the hardware, e.g., Int8, UInt64, Double, Ptr<Int8>, and also seamlessly integrates C++ types and functions. This is exposed
The explanation in the previous section uses the Codegen interface only for a simple store instruction (and some arithmetic). This is one of the simplest operations inside a SQL database system, but clearly, a DBMS needs to support more complex functionality than that. Strings, dates, intervals, JSON, and fixed-point numerics offer many (sometimes) complex functions that need to be integrated into generated code. One option would be to implement this functionality in the Codegen layer and provide layers above that with the complex SQL types they need to work with. However, our design aims to reduce complexity from top to bottom layers and to keep each layer simple. To keep the Codegen layer simple, it only offers *primitive types* plus the means to operate on C++ types. Therefore, we implement the rich semantics of SQL types above the Codegen layer in the SQL Value layer.

2.4 SQL Values

The explanation in the previous section uses the Codegen interface only for a simple store instruction (and some arithmetic). This is one of the simplest operations inside a SQL database system, but clearly, a DBMS needs to support more complex functionality than that. Strings, dates, intervals, JSON, and fixed-point numerics offer many (sometimes) complex functions that need to be integrated into generated code. One option would be to implement this functionality in the Codegen layer and provide layers above that with the complex SQL types they need to work with. However, our design aims to reduce complexity from top to bottom layers and to keep each layer simple. To keep the Codegen layer simple, it only offers *primitive types* plus the means to operate on C++ types. Therefore, we implement the rich semantics of SQL types above the Codegen layer in the SQL Value layer.

---

2 Information Unit (IU) is effectively a reference to a column [24].
2.5 Primitive Types for Code Generation

The SQL Values described in the previous section map SQL types to primitive types and construct operations on SQL types from operations on primitive types. Codegen offers a statically-typed interface to work with primitive types and other means to create programs which we show in the following sections.

Most importantly, Codegen offers classes to generate code for primitive types and uses C++ operator overloading to make it convenient to use. The types are modeled after data types that modern CPUs provide and the basic types that are available in C++. Table 1 lists those types and gives an overview of the main methods they provide to generate code.

Any of the operations on the primitive types have a statically-typed interface. For example, the result of a comparison of Doubles is a Bool and the Ptr<Int64>::load() returns a Int64. This greatly helps to reduce bugs in code generation and reduces the complexity burden on the programmer as they do not have to keep track of types while implementing algorithms. To see this static type system in action, let us have a look at the implementation of our hash function. It generates code to compute a hash of all given SQLValues. As it operates on multiple SQLValues, it belongs to the Tuples layer.
Table 1 Codegen primitive types – Type wrappers and operations on them available in code generation API

| Type       | Available Operations                                                                 |
|------------|---------------------------------------------------------------------------------------|
| (U)Int(8-64) | + - * / % - & | - << >> ashr rotateL rotateR bswap crc32 == ...                                        |
| Bool       | lnot && || select == ...                                                            |
| Double     | + - * / % pow == ...                                                              |
| Data128    | build extract                                                                     |
| Ptr<T>     | load store atomicLoad atomicStore atomicXchg atomicCmpXchg refMember                |

Observe how the primitive types from the Codegen interface are used as regular variables (e.g., Line 9 and 12), and the implementation reads as if the hash function directly acted on the values to hash them. This makes the implementation accessible to readers, yet, when executed on, e.g., two 32-bit integers, the following IR code is generated:

```
1 %1 = zext i64 %int1;  // Zero extend to 64 bit
2 %2 = zext i64 %int2;
3 %3 = rotl i64 %2, 32;  // Rotate right
4 %y = or i64 %1, %3;    // Combine int1 and int2
5 %5 = crc32 164 6753793487589347598, %y;  // First crc32
6 %6 = crc32 164 4593845798347983834, %y;  // Scnd. crc32
7 %7 = rotl i64 %6, 32;  // Shift second part
8 %8 = xor i64 %5, %7;  // Combine hash parts
9 %hash = mul i64 %8, 11400714819323198485;  // Mix parts
```

What also becomes apparent in this example is that even though the implementation of our hash function takes SQLValues as input the generated code is without any remainders of these abstractions. It merely consists of the necessary instructions to perform the task, which constitutes very compact code that can be translated and executed efficiently.

2.6 Host Language Integration

Previous sections explained how to conceptually lower high-level constructs such as relational algebra operators, data structures, and SQL types to programs in Umbra intermediate representation. This code generation process is already fast, as only a single pass over the query plan is required. It is even faster, though, not to generate code at all. Instead, in some situations, it is possible to call functions implemented in the host language, previously compiled at system compile time, without any runtime performance penalty.

To enable seamless integration between generated and precompiled code, Codegen provides a system of proxies that lets us generate operations on any C++ class. We can access data members and call member functions from generated code. Thus, for every feature to implement, the proxy system offers a choice of whether to write code that generates code or to implement the functionality in C++ and call it from generated code. The advantage of the latter option is reduced code generation time.

The use of this technique is shown, e.g., in Figure 5 Line 25. Instead of generating code to create an entry in a hash-table, manage memory allocation, etc., we call a precompiled C++ function. This reduces code generation time and removes complexity from the code generator.

The proxy system is statically typed like the rest of Codegen and therefore offers a fully typed view of C++ classes. It does not need to be created or maintained manually. We generate proxies completely automatically during C++ compile time for a predefined list of classes and functions.

The proxy system has the valuable property that it reduces query compile time by incorporating precompiled snippets, yet does not sacrifice peak execution performance. A function call from generated code into C++ is already quite cheap, as no marshaling is required (as, e.g., would be necessary when using the JVM). It does, however, come at a slight cost at runtime because, e.g., register values must be saved, arguments transferred, and the call stack managed. To avoid this call overhead, the proxy system allows that a programmer can mark functions to be inlined. The Flying Start backend will ignore this inlining marker and only profit from lower compilation time. Our optimizing backend, which aims for peak performance, will react to the marker and inline the function at all call sites, thus removing any calling overhead. This mechanism provides an elegant way to implement functionality in C++, use it in generated code, reduce code generation and compilation time, but without any runtime overhead.

2.7 Control Flow

In previous sections, we showed how to lower operations from complex to primitive types in an architecture that creates clean code and a Codegen that enables fast code generation. This section shows the last missing piece: How to generate control flow in a type-safe interface directly into static single assignment (SSA) form. This form is the preferred program representation for many compilers, especially for our fast compiler Flying Start, which requires it to calculate value life spans (c.f., Section 4.3). Generating SSA directly is important for compilation speed, as it removes the necessity to run an extra compiler pass.
The Codegen provides classes for three control-flow constructs: if, Loop, and Function. They need to handle two aspects: Basic blocks and PHI nodes. Umbra IR organizes instructions in basic blocks (see Section 3). During code generation, there is one current block to which all operations append.

The first aspect is that control-flow constructs need to set the current basic block, so that the following instructions are written to the right location. For example, the if first chooses the then block and when the else block is requested sets it accordingly. When the if goes out of scope, it wires all basic blocks together to produce the desired control flow.

Second, Codegen needs to produce static single assignment form. This means that there are no variables, only names for instruction results. As a substitute for multiple variable assignment, PHI nodes are used. A PHI node is an instruction at the beginning of a basic block and has multiple arguments. Depending on which basic block was executed before the PHI node’s basic block, it chooses one of its arguments as its value. This is used, for example, to choose values in the presence of control-flow without using multiple variable assignments. So to produce static single assignment form, the control-flow constructs offer facilities to construct PHI nodes when needed.

The Codegen provides classes for three control-flow constructs: if, Loop, and Function. They need to handle two aspects: Basic blocks and PHI nodes. Umbra IR organizes instructions in basic blocks (see Section 3). During code generation, there is one current block to which all operations append.

The first aspect is that control-flow constructs need to set the current basic block, so that the following instructions are written to the right location. For example, the if first chooses the then block and when the else block is requested sets it accordingly. When the if goes out of scope, it wires all basic blocks together to produce the desired control flow.

Second, Codegen needs to produce static single assignment form. This means that there are no variables, only names for instruction results. As a substitute for multiple variable assignment, PHI nodes are used. A PHI node is an instruction at the beginning of a basic block and has multiple arguments. Depending on which basic block was executed before the PHI node’s basic block, it chooses one of its arguments as its value. This is used, for example, to choose values in the presence of control-flow without using multiple variable assignments. So to produce static single assignment form, the control-flow constructs offer facilities to construct PHI nodes when needed.

The code above demonstrates how both aspects are handled during a lookup in a chaining hash table. Traversing the chain of hash-table entries is implemented with the Loop construct. Within a nested block we instantiate an object of class Loop, named loop. In its constructor, Loop creates a new basic block for the loop body and sets the new block as the current block. The constructor arguments are a name for the loop for debugging purposes, an entry criterion, and a list of variables that can be “updated” in the loop.

Loops often need to update values in every iteration, for example, they iteratively follow a pointer or increment a loop counter. In single static assignment form, however, no values can be updated. Instead, the Loop class internally uses PHI nodes to pass values to subsequent loop iterations and uses these to present a concept of loop variables to the user (of the Loop class).

In the example above, the constructor argument entry is the initial value for the first loop variable. Inside the loop we access the first loop variable with getLoopVar. Behind the scenes, this constructs a PHI node which also manages updated values in later iterations. The value for the subsequent iteration is then set in loop.done.

Besides PHI nodes the Loop class creates the loop control flow. The constructor generates the loop entry along with the entry criterion, in the example, the criterion was that the entry is not null. The done function connects the last block to the loop head to form a loop under the condition provided in loop.done. On destruction loop create a new basic block for after the loop and thus finalizes the control flow.

We observe that with the help of these control-flow constructs, the code that generates code becomes easier to write and read. Additionally, they allow to directly create static single assignment form in a type-safe manner.

3 Umbra Program Representation

A second element important for query latency in the compilation pipeline are the programs the code generator creates. Programs are the main artifact of the compilation pipeline, thus it is important that the code generator is able to quickly write programs and the backends can quickly read them.

To support this, we designed an intermediate (program) representation that we call Umbra IR. It serves as intermediary between code generator and compilation backends. We took special care that creating programs with Umbra IR is fast. Its data structures are carefully tuned for low memory allocation cost and compactness of representation to efficiently utilize processor caches. The reading speed of Umbra IR is optimized with a low-overhead internal reference format and database-specific instructions. Overall, we chose trade-offs towards low compile-time, yet still perform some optimizations on-the-fly when a program is created.

In the following, we present Umbra IR’s internal data layout, the optimizations performed on the IR, and database specific instructions.
3.1 Umbra IR Structure

Before going into detail of how Umbra IR contributes to low compilation times this section gives an overview of the logical structure of IR programs. A program in Umbra IR consists of functions, basic blocks, and instructions. Functions contain basic blocks of which one is the entry point of the function—i.e., function execution starts there. The example in Figure 6 defines the function `foo` with the basic blocks `start:`, `yes:`, and `no:`.

Basic blocks contain sequences of instructions to be executed in the given order. Each basic block must be terminated by a control flow instruction, for example a conditional branch as shown at the end of the `start:` block in Figure 6. The targets of branches are again basic blocks, so the control flow during execution of a program is determined by control flow instructions and the basic blocks they point to.

Umbra IR offers instructions for arithmetic, loading and storing values, comparisons, casts, atomic memory operations, function calls, returns from functions, branches, conditional branches, and switch, similar to optimizing compilers, e.g., LLVM. Putting this all together in the example in Figure 6, execution would begin with the first block, compare the function argument `%x` to 5 and then either branch to block `yes:` or `no:`. From either one of these blocks, execution returns from the function.

3.2 Physical Program Layout

To make the creation of and analyses on Umbra IR programs fast we utilize three properties of the code generation pipeline:

- Code generation mostly appends instructions at the end of basic blocks. We do not move instructions.
- Code generation has high locality. We generally first complete one basic block/function before moving to the next.
- All instructions have the same lifetime as the program.

With the help of these properties we seek to store the program as compactly as possible to make use of caches, but still allow for quick navigation through the program. We also want to minimize the number of memory allocations. A careless implementation can cause thousands of memory allocations during program generation. Naturally, a fast implementation avoids this as allocations require time.

The first ingredient to Umbra IR’s compact program representation is a variable length instruction format. All 104 instructions begin with an opcode which identifies the instruction—and determines its length—followed by a type identifier that specifies the result type. Each instruction then continues with its specific arguments. The example in Figure 6 shows the program’s instructions on the right side. They begin with an opcode and return type followed by a variable number of arguments.

To achieve data locality while reading and writing instructions Umbra IR stores all instructions of a program in a dynamic array (as illustrated by the box around the instructions in Figure 6). This keeps instructions grouped together in memory and appending instructions does not require allocations (most of the time). It also enables us to reference instructions with a 4-Byte offset into the array. That is particularly helpful as it saves space when instructions reference each other, but still allows to follow references with low overhead.

The basic blocks of a program are similarly stored consecutively in memory. A basic block contains a dynamic array of instruction offsets which point into the instruction array and determine which instructions are in the basic block and in which order. This is depicted in Figure 6 by the dotted arrows. Storage for functions is similar. Each function, however, only contains the offset of the first basic block. From there, all other basic blocks are discoverable through the branches at the end of each block.

The shown representation is less flexible than intermediate representations used in optimizing compilers, e.g., LLVM. However, we find that it yields good cache efficiency and accelerates the generation of programs and executables from it.
3.3 Constants and Dead-Code Removal

The layout of Umbra IR is optimized for fast program generation and is therefore not well suited for complex restructuring passes. However, there are two important optimizations.

First, the Umbra IR builder applies constant folding to instructions at the moment they are appended to the program and deduplicates constants. This potentially decreases the program's size and reduces the workload of later stages in the compilation pipeline.

Second, a dead code elimination pass removes all instructions whose results are not used by any other instruction and any unreachable blocks. Employing an explicit dead code elimination pass gives an advantage in all layers above the Codegen layer. It removes complexity at places in the code generation where we are not completely certain that there will be a user for the value currently produced. With dead code elimination the code generator does not have to carefully determine all users beforehand which makes the generator simpler. As an example for these complexities consider how Tidy Tuples generates code for this if-then-else construct and how constant folding can help to eliminate the else branch:

```plaintext
1
2
3
4
5
6
7
```

```
Bool cond = Int32(4) * Int32(5) > Int32(15);
If test(cond); // Condition is constant
Int64 a = ...;
test.elseBlock(); // Else branch is dead
Int64 b = ...;
test.done();
Int64 result = test.phi(a, b);
```

At the time of code generation we know that the else branch will never be taken. Thus, we could try to not even generate a block for it. However, this would mean that all the instructions that would usually belong into that block, in this case the instruction that generates b, could not be placed into the program. All later sections of the code would then have to handle that any of the values may not exist. This approach would introduce additional corner-cases, be prone for errors, and code which generates code in this manner would be hard to understand. We find that a later dead code elimination pass circumvents those problems.

Because many arithmetic operations in SQL require overflow checking, Umbra IR offers checked arithmetic. Check arithmetic branches when an overflow occurs or continues otherwise. For example, the following performs a 32-bit integer addition of %a and %b that branches to the basic block %overflow on overflow:

```
%c = checkedsadd i32 %a, %b %continue %overflow
```

Such specific instructions remove the need for an extra overflow check and lets backends use the expressed intent to create efficient code.

Umbra IR also combines some instructions in the fashion of inlining to obtain a more compact representation. The `getelementptr` instruction calculates addresses within arrays or structures. Load and store instructions are often combined with address calculation, therefore loads and stores can inline address calculation. Other instructions can also benefit from this technique. We introduced an instruction `isnull` which checks if a value is NULL. It does not require a second argument and thus also no constant for NULL. For the same reason we introduced instructions for CRC checksums, bit rotation, and the 128-bit data type introduced in Section 2.5.

Overall, the benefits of adding database-specific instructions to Umbra IR is (1) that it is easier to generate efficient code in the backends and (2) it yields a more compact program representation.

3.4 DBMS-Specific Instructions

A benefit of using a custom IR is that we can co-design the instruction set with the database system. Most importantly, instructions can express the intent of operations so execution backends can create efficient code for them. Also, instructions that occur frequently can be represented by compact, specific instructions.

Because many arithmetic operations in SQL require overflow checking, Umbra IR offers checked arithmetic. Check arithmetic branches when an overflow occurs or continues otherwise. For example, the following performs a 32-bit integer addition of %a and %b that branches to the basic block %overflow on overflow:

```
%c = checkedsadd i32 %a, %b %continue %overflow
```

Such specific instructions remove the need for an extra overflow check and lets backends use the expressed intent to create efficient code.

Umbra IR also combines some instructions in the fashion of inlining to obtain a more compact representation. The `getelementptr` instruction calculates addresses within arrays or structures. Load and store instructions are often combined with address calculation, therefore loads and stores can inline address calculation. Other instructions can also benefit from this technique. We introduced an instruction `isnull` which checks if a value is NULL. It does not require a second argument and thus also no constant for NULL. For the same reason we introduced instructions for CRC checksums, bit rotation, and the 128-bit data type introduced in Section 2.5.

Overall, the benefits of adding database-specific instructions to Umbra IR is (1) that it is easier to generate efficient code in the backends and (2) it yields a more compact program representation.

3.5 Comparison to LLVM IR

Compared to HyPer, which uses LLVM’s intermediate representation [21], using a custom IR is a different approach. It allows to specifically tune the data layout for low-latency execution and add instructions that more closely express the intent. In terms of semantics, Umbra IR is closely related to LLVM IR. However, LLVM IR is designed to be more generic to support a wide variety of optimization passes. Therefore, LLVM IR has an emphasis on instruction reordering, replacement, and deletion. We observed that this generality entails a performance penalty which we circumvent with Umbra IR.

4 Flying Start Backend

The goal of the Flying Start compilation backend is to reduce query latency, that is, the sum of compile time and query runtime. Flying Start is the default backend for adaptive execution, therefore compile time should be as low as possible. At the same time, as a secondary

3 Umbra’s compilation backends all use the same Umbra IR program. Thus adding a new backend does not add complexity to layers above.
goal, it should create fast code, so it achieves the best combination of compile time and runtime.

The best way to make code run fast and remove any interpretation overhead is to directly generate machine code (as opposed to bytecode for an interpreter). However, generating optimal machine code can be very time consuming. Our approach is to start out from the most basic machine code generator possible. It maps each Umbra IR instruction to exactly one sequence of x86-instructions. There are no choices or optimizations involved, so this is the fastest way to generate machine code from Umbra IR.

Obviously, the resulting code is completely unoptimized and that impedes the secondary goal, fast query execution. To investigate cheap optimization opportunities, we propose the optimizations in Figure 7, which are applied on-the-fly while generating machine code (denoted with “+”). These optimizations explore the design space in the vicinity of the fastest compile time and create different compile-time vs. run-time trade-offs.

The next section gives a short introduction of the adaptive execution technique and details how Flying Start fits into the compilation pipeline. Subsequent sections show the basic translator design and introduce the proposed optimizations step by step.

4.1 Background: Adaptive Execution

There are multiple ways to execute an intermediate representation. All have different trade-offs in code generation time and execution time. Generally, interpreters need little preparation time but execute slower, while optimizing compilers produce fast code, but are slow to generate the code.

Algorithm 1 Basic translation of an add instruction

```python
function compile(Program p)
    for Function f ∈ p; Block b ∈ f; Instruction i ∈ b do
        translate(i)
    end for
end function

function translate(AddInstruction i)
    scratch ← allocScratchRegister()
    firstArgSlot ← i.firstArg()
    secondArgSlot ← i.secondArg()
    result ← allocStackSlotFor(i)
    emit "copy firstArgSlot into scratch register";
    emit "add secondArgSlot onto scratch register";
    emit "copy scratch register value to result";
    free(scratch)
end function
```

Kohn et al. created the adaptive execution method which incorporates multiple execution backends into the HyPer database system [18]. Adaptive execution switches dynamically between execution backends at runtime—even half-way through a query—in order to profit from fast compilation for short-running queries and from fast execution for long-running queries. Figure 2 exemplarily shows two execution backends of HyPer with the trade-offs intrinsic to each backend. HyPer’s bytecode interpreter has slow execution, but compilation does not take long. The LLVM backend (with most optimizations turned off) needs some time for code generation, but execution is faster. Additionally, HyPer can employ LLVM with enabled optimizations to generate even faster code (c.f. Figure 14).

Umbra also applies the adaptive execution approach. It has an execution backend that uses the LLVM optimizing compiler to produce fast executables. Additionally, we introduce the Flying Start backend for fast compilation.

4.2 Minimal Compile-Time Design

The most basic variant of Flying Start uses a single pass over all instructions to generate machine code. Algorithm 1 shows how a program can be compiled with this approach. Each instruction is translated by calling the translator function for its type. Algorithm 1 also shows exemplarily how to translate an Umbra IR add instruction (for an introduction to Umbra IR see Section 3). The translator for add emits a sequence of instructions that load the inputs from stack, perform the addition, and store the outputs.

To show what this means concretely, let us consider the example Umbra IR snippet of Figure 8. The compile function emits code for instruction after instruction.
Eventually, it calls the translate function for the \texttt{add} instruction in Line 4:
\begin{verbatim}
%3 = add i32 %1, i32 %2
\end{verbatim}

Obviously, the translate function in Algorithm 1 is only a sketch. To emit concrete machine code for the Umbra IR \texttt{add} instruction we must choose actual machine instructions. The x86 machine-instruction we want to use for the addition operation is the \texttt{add a, b} instruction. It computes the sum of \texttt{a} and \texttt{b} and stores the result in \texttt{a}. This means the instruction overrides the first input operand.

The translate function must take this peculiarity into account. To keep the first operand value available after the addition instruction, it must first copy the value to a scratch register and use the copy as first operand. So, to prepare the translation, it first reserves a scratch register and also collects bookkeeping information about where the input data resides on the stack and where the result must be stored. Second, it emits instructions to copy the first operand from the stack into the scratch register. Then, to perform the addition, and to copy the result onto the stack. This emits the following machine code to perform addition:
\begin{verbatim}
mov eax, [rsp+firstArgSlot]; Copy arg. into scratch
add eax, [rsp+secondArgSlot]; Add arg. onto scratch
mov [rsp+result], eax; Copy result on stack
\end{verbatim}

Implementing such translators for all Umbra IR instructions yields a program compiler that only requires a single pass over the IR to lower it to machine instructions. The approach has the lowest compile-time because it performs the least work possible to translate each instruction. However, it has some drawbacks: All values are stored on the stack, which causes extra memory traffic, it introduces many superfluous copies, and uses more space on the stack than necessary.

In the following, we devise optimizations to address these issues. They allow us to quantify the trade-offs towards better code quality at the expense of longer compile time.

4.3 Stack Space Reuse

The first optimization that improves the created machine code is using stack space more efficiently. Flying Start can reuse a stack slot once it knows that the value occupying the slot is never used again. To obtain this information, we arrange the program’s basic blocks in reverse post-order and calculate the live spans of all values with the linear time algorithm described by Kohn et al. [18,26]. The live span of a value is the interval from first to last point in the program where the value is live (similar to live intervals of Poletto and Sarkar [38]). Compared to detailed liveness information, e.g., computed by data-flow analysis, live spans are only an approximation. However, live spans can be calculated in linear time and require little memory space per value, which makes them ideal for fast compilation.

Flying Start uses live-span information during compilation to reuse stack slots whose values are not used again. For example in Figure 8, the annotated value live spans on the right show that after the addition in Line 4 the stack slot of argument \texttt{\%1} will not be used again. The computation result \texttt{\%3} can reuse that slot. In the next line, \texttt{\%5} can again reuse the slot and so on.

Generally, reusing stack slots can be cheaply implemented by checking after every translation of an instruction whether the arguments have reached the end of their live span. If they have, we return their stack slots to the stack space allocator\footnote{To be exact: In some cases we need to keep the value in the stack slot until the end of the loop in which the value is defined.}

Stack space allocation introduces additional translation cost as it requires a pass over the program to determine value live-spans. On the plus side, however, it decreases the memory footprint of the resulting code and thus increases the cache friendliness. More importantly, the analysis enables the next (very profitable) optimization.

4.4 Machine Register Allocation

A major issue still is that the compiler generates many superfluous mov instructions to retrieve values from stack and put back results. This behavior is especially unnecessary for values that are passed between instructions within one block, e.g., for the two consecutive additions of Figure 8. For these, the approach of keeping all values on stack generates six instructions of which the majority is unnecessary data movement:
\begin{verbatim}
mov rax, [rsp+slot1]; Data movement
add rax, [rsp+slot2]; Actual operation
\end{verbatim}

\begin{verbatim}
block1:
...%3 = add i32 %1, i32 %2
block2:
%3 = add i32 %1, i32 %2
%5 = add i32 %3, i32 %4
%6 = cmpult i32 %5, i32 64
condbr %6 %block1 %block3
block3:
...%3 %4
%5 %6
\end{verbatim}

\textbf{Fig. 8 Umbra IR snippet} — as running example for Flying Start translation.

Eventually, it calls the translate function for the \texttt{add} instruction in Line 4:
\begin{verbatim}
%3 = add i32 %1, i32 %2
\end{verbatim}

\begin{verbatim}
mov eax, [rsp+firstArgSlot]; Copy arg. into scratch
add eax, [rsp+secondArgSlot]; Add arg. onto scratch
mov [rsp+result], eax; Copy result on stack
\end{verbatim}

\begin{verbatim}
mov rax, [rsp+slot1]; Data movement
add rax, [rsp+slot2]; Actual operation
\end{verbatim}

\textbf{Fig. 8 Umbra IR snippet} — as running example for Flying Start translation.

\begin{verbatim}
block1:
...%3 = add i32 %1, i32 %2
block2:
%3 = add i32 %1, i32 %2
%5 = add i32 %3, i32 %4
%6 = cmpult i32 %5, i32 64
condbr %6 %block1 %block3
block3:
...%3 %4
%5 %6
\end{verbatim}

\textbf{Fig. 8 Umbra IR snippet} — as running example for Flying Start translation.

\begin{verbatim}
mov eax, [rsp+firstArgSlot]; Copy arg. into scratch
add eax, [rsp+secondArgSlot]; Add arg. onto scratch
mov [rsp+result], eax; Copy result on stack
\end{verbatim}

\begin{verbatim}
mov rax, [rsp+slot1]; Data movement
add rax, [rsp+slot2]; Actual operation
\end{verbatim}

\textbf{Fig. 8 Umbra IR snippet} — as running example for Flying Start translation.

\begin{verbatim}
block1:
...%3 = add i32 %1, i32 %2
block2:
%3 = add i32 %1, i32 %2
%5 = add i32 %3, i32 %4
%6 = cmpult i32 %5, i32 64
condbr %6 %block1 %block3
block3:
...%3 %4
%5 %6
\end{verbatim}

\textbf{Fig. 8 Umbra IR snippet} — as running example for Flying Start translation.
A way to eliminate data movement to and from stack is to keep values in machine registers beyond the boundaries of the translation of a single instruction. Therefore, in addition to assigning each value a slot on the stack, we also try to assign a machine register. This reduces the need for data movement instructions as values reside in registers. The two additions, for example, can then be implemented with only two instructions:

```asm
add r9, r10;  // Add second arg. onto first
add r9, r11;  // Add third arg. onto prev. result
```

Of course, this example shows the ideal case. In reality, there are much fewer registers available on x86 machines than there are usually values in our Umbra IR programs. In order to make best use of the available registers we adopt a best effort approach. Out of the available 16 registers on the target machine, four registers are scratch registers, one register contains the stack pointer, and the remaining 11 registers store values beyond the translation of a single IR instruction.

One could try to assign these 11 machine registers to values on a first-come first-served basis. Unfortunately, this strategy leads to a shortage of available machine registers for short-lived values and especially inside nested loops. We found it to be more beneficial to assign machine registers to values that either only live within the block they were created in or were created in the most deeply nested loop. This heuristic is cheap to compute from the data already at hand and effectively shifts the register usage to the passing of intermediate data and into loops. Consequently, it reduces the number of generated instructions and memory accesses.

### 4.5 Lazy Address Calculation

Besides register allocation, there are two additional minor optimizations. The first concerns the address calculation instruction `getelementptr`. Generated code frequently accesses different elements of one tuple or data structure. For data access Umbra IR programs use pointer arithmetic with the `getelementptr` instruction to compute data locations. This often leads to a chain of multiple address calculation instructions. To extend the running example of Figure 8, `block1` obtains the input data for the example with these address calculations and load instructions:

```asm
mov [rsp+slot1], rax;   // Data movement
mov rax, [rsp+slot1];   // Actual operation
add rax, [rsp+slot3];   // Data movement
mov [rsp+slot1], rax;   // Data movement
```

The program first computes a pointer to a tuple, then computes the pointer to the first and second element with separate `getelementptr` instructions. If the compiler would emit machine instructions for each address calculation instruction separately, it would produce an extra `add` instruction and use an extra register. However, the x86 instruction set offers an alternative as it allows to integrate address calculation into instruction operands. To implement the load in Line 5 the compiler can use the `mov` instruction with one register and one memory operand:

```asm
mov r9, [rdx + offset]
```

This form of integrated addressing can be achieved by delaying address calculation. When translating pointer arithmetic instructions the compiler does not fully resolve them to yield a single pointer value. Instead it keeps the form `[base + offset]` (where `base` is a register and `offset` a constant). This enables the translator to use the composite form in instruction operands.

### 4.6 Fuse Comparison and Branch

The second minor optimization concerns comparisons and branches. Comparisons in Umbra IR result in a Boolean value which conditional branches take as input. This is an elegant construct, but unfortunately, it does not map directly to any machine instructions. In x86, a comparison sets a special flags register and branches take the flags as input. Translating comparison and branch instruction separately would produce extra machine instructions. The compiler would have to retrieve the comparison result from the flags register, only to move it right back into the flags register when translating the next instruction:

```asm
cmp r9, 64;               compare
setlt r13b;              retrieve cmp. result from flags register
cmp r13b, 1;             put decision into flags register
jnz .block1;             branch, depending on flags value
```

To avoid this situation, we must achieve during translation that the comparison and the conditional branch are translated adjacently. Also, during the translation of the comparison the compiler must decide whether to leave the result only in the flags register.

---

5. Similar to address inlining in Umbra IR (Section 3.4). Unfortunately, we can not rely on that, because addresses with multiple users can not be inlined.

6. The lifetime of the base register must be extended to cover later uses.
Fig. 9 Foundation of the Flying Start Backend — This is the core of translation from Umbra IR to x86. The classes Reg and ScratchReg perform book-keeping of values and free registers. They determine where inputs are located, where results should be placed, and which temporary registers to use.

An extra reordering and analysis pass over the input program could enforce adjacency, but the extra pass would come at the expense of compilation time. Instead, within a basic block we defer deference of translation, all but load, store and control-flow instructions, e.g., we defer comparisons. Instructions are translated at the latest possible time, that is when their results are required by other instructions.

For the above example, the lazy approach first skips the translation of the comparison instruction. On translation of the branch the compiler notices that the input is not yet computed. At this point, it starts translating the input and also passes along the request to put the result into the flags register. Then, in the translation of the comparison it sees the request, checks if there is only one consumer, and puts the result into the flags register. The branch instruction can then directly use the flags register:

\[
\text{cmp r9, 64;} \quad \text{compare}
\]
\[
\text{jl .block1;} \quad \text{branch, depending on flags value}
\]

On-demand instruction translation can also pass requests from value users to producers and in this case also guarantees that there is no user of the flags register in between comparison and branch.

4.7 Implementation of Flying Start

So far, Algorithm 1 presented the code emitter in a fairly abstract fashion. Our actual implementation in C++, is very similar. Figure 9 shows an implementation of the translation of the Umbra IR addition instruction.

We use the classes Reg and ScratchReg to keep track of input and result data (Lines 4-6), and to allocate scratch registers (Line 8). To emit machine instructions our implementation uses the asmJIT library [16]. It provides the ability to directly assemble x86 instructions. E.g., in Line 12 the translator emits the add instruction from the running example into a buffer. Appending multiple instructions to this buffer forms the translated program.

All the described optimizations fit very well into this code structure. For example, the register allocation heuristic is hidden in the bookkeeping class Reg. Lazy address calculation and fusing comparisons and branches require only small additions.

Figure 10 shows an implementation of the bookkeeping functions for instruction translation. Observe how the function resultReg decides right at the moment of instruction translation where to place computation results. Either the allocation heuristic decides to put the value into a machine register or the result is placed on the stack. Similarly, fusion of comparisons and branches is handled behind the scenes. The function argumentReg also handles deferred translation of instructions. When the result of an instruction %b is required, e.g., during instruction translation of %a = add(%b, %c). function argumentReg checks if %b is already computed. If not, the
Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra

5 Evaluation

This section evaluates the quantifiable properties of Tidy Tuples and Flying Start, confirming these performance hypotheses:

– The design achieves very low overall query latency over all database sizes and across multiple machine configurations (Section 5.2).
– Umbra IR speeds up code generation (Section 5.3).
– The Flying Start backend dominates multiple state-of-the-art alternatives (Section 5.4).
– The optimizations in the Flying Start backend all provide performance benefits (Section 5.5).

5.1 Experimental Setup

All experiments were run on a machine with a 10-core Intel Skylake X i9-7900X clocked at 3.4 GHz and a turbo boost of 4.5 GHz. The processor provides 20 hyperthreads, an L1-cache of 32 kB for every core and a last-level cache of 14 MB. The machine has 128 GB of DRAM with an aggregate bandwidth of 56 GB/s and uses Ubuntu 19.04 with kernel 5.0.0 as operating system. The TPC-H benchmark serves as workload with scale factors from 0.001 with 10 thousand tuples to scale factor 30 with about 260 million tuples. PostgreSQL was installed with version 11.7 and configured to use up to 20 workers per query. Further, index and bitmap scans are disabled to obtain query plans comparable to Umbra. DuckDB was compiled from commit aec86f6; MonetDB was installed in version 11.33.11.

5.2 Query Latency: Compile Time + Runtime

The goal of Tidy Tuples and Flying Start is to minimize the query latency of compiling query engines. That is, minimizing compilation overhead while at the same time processing queries as fast as possible. This section we evaluates to what extent that goal is achieved.

Compilation time can be traded for execution time, to a certain degree. Tidy Tuples and Flying Start constitute a specific design point in that trade-off. Whether a chosen trade-off is beneficial depends on the ratio of compilation time and execution time within a query. For a given system, compilation time is directly determined by the query. Execution time depends on the data set size and the amount of resources/threads used for processing. To evaluate the trade-off in a variety of scenarios we use the TPC-H benchmark.8 It provides representative OLAP queries that cover a range of compilation

Fig. 11 Flying Start achieves low query latency over a wide range from tiny to large datasets. – Over geometric mean of queries per second over all 22 TPC-H queries Flying Start out-performs even DuckDB, MonetDB, and PostgreSQL, which do not spend any time on code generation and compilation. Threads7 = 20

7 Currently, DuckDB can only use one thread for execution. Nevertheless, it provides an interesting comparison on small datasets.

8 At the time of writing, Umbra does not have a high-performance transaction processing implementation. Thus, we can not yet compare on OLTP benchmarks. Umbra’s relational operator implementations, however, are prepared to integrate well with transaction processing—similar to HyPer’s operators. For example, Umbra does not use precomputed values or dictionary encoding for query processing.
Fig. 12 Umbra with Flying Start achieves low query latency across machine configurations. – In geometric mean (Geo.M.) Umbra answers queries faster than the fastest competitor Monet DB. This is already the case when only one thread is available for query processing and holds true with additional threads. Query latency is low over the full range of long running (Q1, Q9) and short running (Q6, Q2) queries as well as queries with short (Q1, Q6) and long (Q9, Q2) compile time. SF=0.1

The quality and speed of generated code are relevant here. As shown in the bottom half of Figure 11, Flying Start produces sufficient code quality to out-perform other approaches on data sets with up to hundreds of millions of tuples. Altogether, Umbra processes queries with high speed over all data set sizes, which means the trade-off is beneficial for a large range of scenarios.

The other important factor in the trade-off is the number of threads used for execution. Figure 12 shows Umbra’s execution time depending on the number of threads. As a reference point it also shows the fastest competitor, Monet DB. Note, Umbra’s execution phase can make use of multiple threads and operators use morsel-driven parallelisation [22]. The compilation phase, with code generation and compilation, uses only a single thread.

In a broad view over all TPC-H queries (Geo.M.), Umbra is able to respond to queries faster than the other systems when using a single thread for execution up to using all available threads. Figure 12 also shows detailed performance results for queries which are chosen to cover the full range of runtime/compile time characteristics. There are long running (Q1, Q9) and short running (Q6, Q2) queries to examine the interaction of overall query runtime and number of threads. Both groups have a query with low (Q1, Q6) and high (Q9, Q2) compilation time to additionally vary the compile time/runtime ratio within each group and thus cover the whole spectrum. Notably, in call cases code generation and compilation provides faster overall query response time than the fastest interpreter-based approach.

Overall, we observe that Umbra’s latency optimizations work very well. They allow Umbra to reach far into the low latency realms of query engines that do not compile at all. Furthermore, note that the latency optimizations do not interfere with query execution speed.

9 We manually decorrelated queries for PostgreSQL for a fair comparison.
Table 2 Tidy Tuples, Umbra IR, and Flying Start speed up Umbra's preparation phase. Umbra is twice as fast as HyPer, thus preparation time is as low as interpreter-based systems. – The table lists detailed timing in milliseconds for TPC-H queries 1-22 and geometrical mean (\(G\)). Planning time ("plan") includes query parsing, semantic analysis and algebraic optimization. Umbra and HyPer also list compile time, split into generation of IR ("cdg.") and generation of machine code ("x86"/"bc."). For Umbra and HyPer LLVM compilation is excluded, as its compile times are too long for a millisecond for TPC-H queries 1-22 and geometric mean (\(\Sigma\)). Table 2 Tidy Tuples, Umbra IR, and Flying Start speed up Umbra's preparation phase. Umbra is twice as fast as HyPer, thus preparation time is as low as interpreter-based systems. – The table lists detailed timing in milliseconds for TPC-H queries 1-22 and geometrical mean (\(G\)). Planning time ("plan") includes query parsing, semantic analysis and algebraic optimization. Umbra and HyPer also list compile time, split into generation of IR ("cdg.") and generation of machine code ("x86"/"bc."). For Umbra and HyPer LLVM compilation is excluded, as its compile times are too long for a millisecond for TPC-H queries 1-22 and geometric mean (\(G\)).

The combination of Flying Start and the optimizing compiler backend outperform the competitors in all cases. From this, we conclude that our latency optimizations are effective.

5.3 Compilation Time

Now that we have seen that the overall design achieves good query latency let us focus on how time is spent in the query compilation phase, i.e., before query execution. Table 2 shows a breakdown of query processing time for Umbra with Flying Start and its competitors on the little TPC-H data set at scale factor 0.01. Umbra timing is split into the planning phase ("plan"), the code generation phase ("cdg."), machine code generation ("x86"), and query execution ("exec."). The planning phase includes query parsing, semantic analysis, and algebraic optimization. Creation of Umbra IR happens in the code generation phase and the machine code generation phase produces x86 instructions. Similarly for the competitor HyPer, yet instead of generating machine code, it produces bytecode ("bc."), for its interpreter. For the interpreting engines DuckDB, MonetDB, and PostgreSQL the table only distinguishes between plan and execution. Finally, it also lists the time for the sum of all components ("\(\Sigma\)").

The combination of Flying Start and the optimizing compiler backend outperform the competitors in all cases. From this, we conclude that our latency optimizations are effective.

For Umbra, we observe that once the query plan is prepared, execution ("exec.") on the little dataset does not take long—it is even shorter than query preparation. All competitors spend more time during execution. The time Umbra spends before execution, to prepare the executable, though, is slightly more than the competitors. On average Umbra takes 0.66 ms to prepare, whereas DuckDB and MonetDB only need 0.47 ms and 0.53 ms respectively. This puts Umbra with Tidy Tuples and Flying Start well within the same order of magnitude of query preparation time as interpreter engines, even though Umbra additionally performs all the steps required for machine code generation.

Compared to HyPer, the most similar system as it also spends time on code generation, we observe that Umbra starts faster. HyPer needs 1.33 ms on average to prepare for a query and Umbra only 0.66 ms. The main difference here is that HyPer generates LLVM’s intermediate representation and Umbra uses its Umbra IR representation. The effect clearly shows in the differences of code generation time ("cdg."), where Umbra is more than 2× faster than HyPer. A similar, but smaller, effect is visible during generation of the executable ("x86" and “bc.”). Flying Start is faster at x86 generation than HyPer at bytecode generation. We conclude that Umbra IR speeds up code generation and thus serves its purpose well as it effectively reduces Umbra’s query latency.
5.4 Runtime Performance Robustness

The previous section established that the compilation times of Flying Start are competitive with interpreter engines. Let us now explore the compile time versus execution speed that it offers.

Recall from Section 4.1 that Umbra and HyPer both use adaptive execution to run the generated code and to balance compilation time and runtime. The systems use multiple compilation backends that offer different compilation and execution speeds. HyPer switches between the three backends bytecode Interpreter, LLVM unoptimized, and LLVM optimized. Umbra only uses the Flying Start backend and LLVM for thoroughly optimizing machine code.

How all these runtime backends perform is depicted in Figure 14 for the example of TPC-H query 3 at scale factor 1. Among HyPer’s execution backends, bytecode interpretation provides the lowest compilation time, albeit with a noticeable execution time penalty. HyPer’s next best option is to use the LLVM compiler with almost all optimizations turned off. This yields good execution performance, but comes with a higher compile time. Note, that it is already apparent, that Umbra’s Flying Start backend offers a better choice. It is on par with the bytecode interpreter’s compilation time and the runtime performance of LLVM (unoptimized) machine code. Hence, Flying Start combines the advantages of HyPer’s two low-latency backend options into only one.

When expanding the view from this one example query to all the TPC-H queries, we see a similar picture in the trade-offs in Table 3. In comparison to fully optimizing the machine code with LLVM, on geometric mean over all queries the Flying Start backend offers 108× faster compilation at the low cost of only 1.2× slower execution. This all happens in a single compilation backend. For Umbra’s competitor HyPer, this option is split...
in two. The Hyper interpreter backend provides 91× faster compilation at the cost of 4.1× slower execution. The alternative cheap compilation backend with LLVM offers 6× faster compilation producing code that executes 1.3× slower.

To summarize, HyPer must juggle three execution backends. As shown in Figure 14 each backend provides a different trade-off between compilation time and runtime. The results can be observed in Figure 11, where every backend yields the fastest overall execution speed over a limited range of scenarios. Thus, the system must carefully choose the correct one of three backends, as a wrong choice can gravely impede execution performance. Umbra, on the other hand, only has to choose from two backends. Flying Start combines the best of the byte-code interpreter and the unoptimized LLVM backend. It is as fast in generating code as the interpreter and as fast in execution as the unoptimized LLVM backend. Consequently, it is safe to always begin execution with Flying Start and, if necessary, shift into high gear by using the optimizing compiler. As the difference in execution speed between the backends is only 1.2×, a wrong choice only has a small impact on execution time and the performance cliff in a sense becomes a small performance step.

5.5 Flying Start Optimizations

We described in Section 4 that the Flying Start backend uses four optimizations to improve the speed of the generated code. We measured the effect of each optimization on compilation and execution time for all TPC-H queries.

Figure 15 shows the results for execution time on some exemplary and interesting queries and also of the geometric mean over all 22 queries. Observe that the biggest effect is achieved by register allocation. On average it provides a 32% reduction of execution time. Interestingly, in the Umbra LLVM backend, register allocation also provides the largest performance benefit among the applied optimizations (c.f., Figure 17). Switching from fast instruction selection to the default instruction selection enables machine specific optimizations, such as register allocation and instruction scheduling. Further optimizations only have a small effect on the runtime. In other words, the largest optimization potential is covered by Flying Start’s register allocation.

Given that register allocation has such a large impact, an interesting idea to improve Flying Start would be to use a better register allocator than the already applied heuristic. An allocation scheme often used in fast compilers is Linear Scan [38]. In a single pass over all lifetime intervals it decides which values live in registers. To compare with our allocation heuristic, we added
Linear Scan to the Flying Start backend. Linear Scan produces good allocations; the machine code produced with linear scan leads to 1% faster query execution on TPC-H (c.f. Figure 16). However, allocation with Linear Scan takes 14% more compilation time. This presents an interesting trade-off, yet in the interest of low compile time for now we chose not to add Linear Scan to the Flying Start default optimizations.

The experiment shows that some queries profit more from optimizations than others. Query 1 shows the largest gains, as most of its work is in expression evaluation. Thus, keeping intermediate values in registers increases the CPU’s instruction throughput. Third, address calculation and comparison-branch fusion provide only a moderate effect. The benefit is most pronounced on query 18. Overall, we observe that every one of the optimizations increases execution speed.

The quality of the machine code generated by Flying Start is good in comparison to the fully optimized code from LLVM. As Figure 18 shows, performance metrics of Flying Start code for TPC-H queries are well within the same order of magnitude as the corresponding LLVM-generated machine code. Previous experiments already showed that the execution speed of Flying Start code is close to the speed of highly optimized code. This also shows in Figure 18, in which the amount of cycles to execute queries with Flying Start is on median about 1.6× higher than with highly optimized code. Notably though, the number of instructions executed is about 2.3× higher, which means that Flying Start produces some amount of extra instructions. Fortunately, also the number of instructions executed per cycle (IPC) is 1.4× higher. The processor is able to execute more instructions in parallel within each cycle which reduces the negative effect of extra instructions. Branch-miss-predictions and last level cache (LLC) misses are about the same for both compilers. The size of the generated code from Flying Start is about 2.4× larger than optimized code. Overall, Flying Start generates some superfluous instructions, yet the hardware is able to partly compensate that. More importantly, Flying Start code triggers the same amount of hardware hazards, i.e., branch-misses and cache-misses, as optimized code, but triggers no additional hardware hazards.

Another optimization that Umbra performs is to eliminate dead (unused) code. Technically, it is an optimization applied during the code generation process, not by Flying Start, yet it effects compile- and runtime performance. Figure 19 shows TPC-H compile- and runtime with and without dead code elimination (DCE). As explained in Section 3.3, Tidy Tuples uses dead code elimination to simplify structure of the code generation layer. These experiments show that dead code elimination is a rather quick pass compared to the remaining compilation time. Also, as DCE removes about 4% of instructions, it reduces the following compile- and runtime, thus recaptures some of the time spent on the optimization pass.

For all optimizations there is a compilation time price to pay, as shown in the bottom of Figure 15. We note that the only optimization that comes at a measurable cost is the value-lifetime computation which is first used for the stack reuse optimization. On average, it adds 45% to compilation time. Interestingly, all further optimizations more than offset their cost. Any one of these optimizations helps reduce the number of emitted instructions and consequently reduces the time necessary to write machine code.

Given that lifetime computation adds about 45% to compilation time one may also choose to skip it and therefore not employ any of the four optimizations. In Umbra, however, we use it, because it makes the query engine more robust. It prevents that queries with many intermediate values have an unnecessarily large memory footprint. Additionally, it reduces the performance cliff
towards the optimizing compiler. To summarize, the optimization in Flying Start increase the execution speed and robustness of the query engine.

### 5.6 Implementation Effort

Building a compiling SQL query engine from scratch is a large undertaking and Tidy Tuples is meant to structure such an effort and serve as a guideline. To give an idea of the size of the code generator in Umbra, Table 4 lists the lines of C++ code required to implement components of Tidy Tuples and Flying Start. Lines with comments and documentation do not count towards the lines of code. We also exclude lines which only contain opening or closing curly braces to account for a peculiarity of the used code style.

The shown components follow the structure of Tidy Tuples as presented in Section 2.2. Lines of code are separately counted for C++ header files, C++ implementation files, and unit tests that directly test the functionality of the component (integration tests for the entire system are not listed). Operator translators include table scan, nested-loop join, hash join, multi-way join, group-join, group by, sort, map, select, set operations, expressions, recursive views, and many more. Each of the operators in turn may need to handle multiple variants of the operator. For example, the hash join translator can produce inner, outer, semi, mark [29], and single [29] joins. Thereof all, but the inner join have different right join and left join implementations. Overall, there are many concepts in relational queries and their efficient implementation often requires attention to detail. In our experience with the implementation of Umbra, that detail and the inherent complexity is structured well by the Tidy Tuples design.

### 6 Related Work

There are two state-of-the-art query processing paradigms: vectorization and compilation. Vectorization reduces the overhead of Volcano-style interpreters by performing an operation on many tuples at the same time. It was pioneered in MonetDB [3] and improved upon by MonetDB/X100 [2]. As vectorized engines are interpreters, they can use Volcano-style interpretation and generally have a reputation of being easier to build. Furthermore, because they do not generate machine code, they can potentially have lower query latency—while being efficient for analytical workloads [14]. However, there are drawbacks with complicated expressions and especially when only few tuples are in a query, as is commonly the case in transaction processing.

Compilation-based engines eliminate interpretation overhead by generating query-specific machine code. An architecture for generating machine code was shown with the HyPer system [25, 28]. This approach was criticized as too low-level [15] and, in the context of LegoBase, an alternative approach was proposed. Instead of generating code from the query plan in one single step, LegoBase gradually lowers it through a cascade of intermediate representations to the effect that each lowering by itself is less complex [43]. Using multiple representations was then criticized as adding unnecessary complexity [44]. A solution was presented by using the idea of the Futamura projection to specialize an interpreter to obtain a code generator. The LB2 system uses Scala language features and compiler extensions to implement this idea and create an interpreter engine as well as a code generator, derived from the same code base. Further research on the structure of relational code generators has shown that, besides HyPer’s produce-consume model, Volcano-style communication between operators can also be used for code generators. However, extensive compiler optimizations are required to obtain efficient code from code generators with Volcano-style iterators [41]. An alternative to distinguishing between interpreters and code generators is to use micro-specialization on an interpreter system [47, 48]. Kohn et al. presented the adaptive execution approach for HyPer, which combines an interpreter and a code generator to achieve low latency for cheap queries and fast execution speeds for expensive queries [18].

The work presented in this paper builds on all of these contributions. Tidy Tuples features a layered architecture of abstractions that conceptually incorporates the gradual lowering of LegoBase, but still achieves code
We show that this approach enables low query latencies as in the sea-of-nodes representation, the arguments of one instruction reach into the realm of interpreted and vectorized code from code generators with Volcano-style iteration. The Flying Start backend is tailored to these qualities and makes use of them to leverage this information to manage with only a single pass. Liftoff thus depends on the compiler that generates WebAssembly to encode lifetimes. The Flying Start backend cannot do this, as its compiler is executed right ahead of it in the same compilation pipeline. Similarly, Flounder IR is a program representation that relies on the code generator to encode value lifetimes [9]. The proposed design for Flounder IR is to estimate values lifetimes with relational operator lifetimes. For Umbra IR and Flying Start, we observed for TPC-H queries that operator lifetimes overestimate the lifetimes and lead to a shortage of available registers.

LuaJIT is a fast just-in-time compiler for the dynamically typed language Lua. Execution starts with interpreting Lua bytecode [32]. A tracer then finds code sections worth compiling and creates a statically typed IR [33]. This IR, much like the Umbra IR, contains features and instructions that are very specific to Lua. A backend with multiple compiler passes can lower the IR to machine code.

Destination driven code generation, Liftoff, and LuaJIT rely on certain properties of their input programs and so does the Flying Start backend. It profits from the fact that the produce/consume interface generates efficient and short code. Values are typically loaded from memory only once and are then used in multiple places by reference to only a single Umbra IR handle. In addition, constant folding is performed on-the-fly during program generation. The Flying Start backend is tailored to these qualities and makes use of them to save compilation time.

7 Summary

This paper presented the Tidy Tuples architecture, the Umbra IR program representation, and the Flying Start compiler backend to minimize query latency in compiling relational database systems. They optimize the whole execution pipeline from the arrival of a query plan to when the result is ready.
The Flying Start compilation backend showed that very fast machine code generation is possible and the generated code executes queries only slightly slower than highly optimized code. Furthermore, Umbra IR, a customized intermediate representation with optimized data structures helps reduce the time spent for generating code and transferring code into machine instructions. Lastly, Tidy Tuples structure code generators so that complexity is well managed, yet code generation is very fast and thus contributes to lower query latency.

We implemented the proposed optimizations in the database system Umbra. An evaluation found that the optimizations are effective at lowering query latency. The experiments showed that Umbra’s compilation latency becomes competitive with systems that do not compile at all, e.g., DuckDB and MonetDB. At the same time, the execution speed of Umbra is on par with state-of-the-art query engines.

To conclude, we advocate the use of a fast compiler that directly generates machine code and in some cases, falls back to an optimizing compiler. This approach reaches the low-latency realms of interpreter engines and at the same time keeps a high execution speed in larger datasets. Such a query engine can compile very quickly and produces machine code that makes efficient use of the processors. It is thus well equipped to optimally use the large bandwidth provided by main memory and new storage hardware, e.g., SSDs and Persistent Memory. Its low query response time makes it predestined for a burst of many small queries intermixed with large queries—as regularly happens during interactive database use.

References

1. Agarwal, S., Liu, D., Xin, R.: 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)
2. Boncz, P., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-pipelining query execution. In: CIDR (2005)
3. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in MonetDB. Commun. ACM 51(12), 77–85 (2008)
4. Click, C.: Global code motion / global value numbering. In: SIGPLAN, pp. 246–257 (1995)
5. Crotty, A., Galakatos, A., Dursun, K., Kraska, T., Binnig, C., Çetintemel, U., Zdonik, S.: An architecture for compiling UDF-centric workflows. PVLDB 8(12), 1466–1477 (2015)
6. Crotty, A., Galakatos, A., Dursun, K., Kraska, T., Çetintemel, U., Zdonik, S.B.: Tupleware: “big” data, big analytics, small clusters. In: CIDR (2015)
7. Diacou, C., Freedman, C., Ismert, E., Larson, P., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL server’s memory-optimized OLTP engine. In: SIGMOD, pp. 1243–1254 (2013)
8. Dybvig, R.K., Hieb, R., Butler, T.: Destination-driven code generation. Tech. rep., Indiana University Computer Science Department (1990)
9. Funke, H., Mühlig, J., Teubner, J.: Efficient generation of machine code for query compilers. In: DaMoN, pp. 6:1–6:7 (2020)
10. Gupta, A., Agarwal, D., Tan, D., Kulesza, J., Pathak, R., Stefani, S., Srivivasan, V.: Amazon Redshift and the case for simpler data warehouses. In: SIGMOD, pp. 1917–1923 (2015)
11. Haas, G., Haubenschild, M., Leis, V.: Exploiting directly-attached nvme arrays in DBMS. In: CIDR (2020)
12. Hammacher, C.: https://v8.dev/blog/liftoff (2018). URL https://v8.dev/blog/liftoff
13. Karpathiotakis, M., Alagaginis, I., Heintz, T., Branco, M., Alamaki, A.: Just-in-time data virtualization: Lightweight data management with ViDa. In: CIDR (2015)
14. Kersten, T., Leis, V., Kemper, A., Neumann, T., Pavlo, A., Boncz, P.A.: Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. PVLDB 11(13), 2209–2222 (2018)
15. Klonatos, Y., Koch, C., Rompf, T., Chafi, H.: Building efficient query engines in a high-level language. PVLDB 7(10), 853–864 (2014)
16. Kohn, A., Leis, V., Neumann, T.: Adaptive execution of compiled queries. In: ICDE (2018)
17. Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., Shaikhha, A.: DBToaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J. 23(2), 253–278 (2014)
18. Koh, A., Leis, V., Neumann, T.: Just-in-time data virtualization: Lightweight data management with ViDa. Commun. ACM 51(12), 2150–2164 (2018)
19. Krikellas, K., Viglas, S., Cintra, M.: Generating code for holistic query evaluation. In: ICDE, pp. 613–624 (2010)
20. Kroll, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: CGO, pp. 75–88 (2004)
21. Leis, V., Boncz, P., Kemper, A., Neumann, T.: Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In: SIGMOD, pp. 743–754 (2014)
22. Leis, V., Boncz, P., Kemper, A., Neumann, T.: Adaptive execution of compiled queries. In: ICDE (2018)
23. Menon, P., Mowry, T.C., Pavlo, A.: Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. PVLDB 11(1) (2017)
24. Moerkotte, G.: Building query compilers. http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf
25. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB 4(9) (2011)
26. Neumann, T.: Linear time liveness analysis (2020). URL http://pvlldb.fusionforinmemorydatabases/makecompilationvec-
27. Neumann, T.: Making compilation, vectorization, and prefetching work together at last. PVLDB 11(1) (2017)
28. Neumann, T.: Just-in-time data virtualization: Lightweight data management with ViDa. Commun. ACM 51(12), 77–85 (2008)
29. Neumann, T.: Adaptive execution of compiled queries. In: ICDE (2018)
30. Neumann, T.: Just-in-time data virtualization: Lightweight data management with ViDa. Commun. ACM 51(12), 1466–1477 (2015)
31. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB 4(9) (2011)
32. Neumann, T.: Linear time liveness analysis (2020). URL http://pvlldb.fusionforinmemorydatabases/makecompilationvec-
33. Neumann, T., Freitag, M.J.: Efficiently compiling efficient query plans for modern hardware. PVLDB 4(9) (2011)
34. Neumann, T., Freitag, M.J.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
35. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB 4(9) (2011)
36. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB 4(9) (2011)
37. Neumann, T., Freitag, M.J.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
38. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
39. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
40. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
41. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
42. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
43. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
44. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
45. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
46. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
47. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
48. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
49. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
50. Neumann, T.: Efficient compilation of query plans for modern hardware. PVLDB 4(9) (2011)
31. Palkar, S., Thomas, J.J., Shanbhag, A., Schwarzkopt, M., Amarasinghe, S.P., Zaharia, M.: A common runtime for high performance data analysis. In: CIDR (2017)
32. Pall, M.: http://wiki.luajit.org/Optimizations (2012). URL http://wiki.luajit.org/Optimizations
33. Pall, M.: http://wiki.luajit.org/SSA-IR-2.0 (2013). URL http://wiki.luajit.org/SSA-IR-2.0
34. Paroski, D.: Code generation: The inner sanctum of database performance. "http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html" (2016)
35. Pirk, H., Giceva, J., Pietzuch, P.R.: Thriving in the no man’s land between compilers and databases. In: CIDR (2019)
36. Pirk, H., Moll, O., Zaharia, M., Madden, S.: Voodoo - a vector algebra for portable database performance on modern hardware. PVLDB 9(14), 1707–1718 (2016)
37. Poletto, M., Engler, D.R., Kaashoek, M.F.: tcc: A system for fast, flexible, and high-level dynamic code generation. In: SIGPLAN, pp. 109–121 (1997)
38. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Trans. Program. Lang. Syst. pp. 895–913 (1999)
39. Raasveldt, M., Mühleisen, H.: Duckdb: an embeddable analytical database. In: SIGMOD, pp. 1981–1984 (2019)
40. van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L., Sato, M.: Managing non-volatile memory in database systems. In: SIGMOD, pp. 1541–1555 (2018)
41. Shaikhha, A., Dashti, M., Koch, C.: Push versus pull-based loop fusion in query engines. J. Funct. Program. 28, e10 (2018)
42. Shaikhha, A., Klonatos, Y., Koch, C.: Building efficient query engines in a high-level language. ACM Trans. Database Syst. 43(1) (2018)
43. Shaikhha, A., Klonatos, Y., Parreaux, L., Brown, L., Dashti, M., Koch, C.: How to architect a query compiler. In: SIGMOD, pp. 1907–1922 (2016)
44. Tahboub, R.Y., Essertel, G.M., Rompf, T.: How to architect a query compiler, revisited. In: SIGMOD, pp. 307–322 (2018)
45. Vogelsgesang, A., Haubenschild, M., Finis, J., Kemper, A., Leis, V., Mühlbauer, T., Neumann, T., Then, M.: Get real: How benchmarks fail to represent the real world. In: DBTest (2018)
46. Wanderman-Milne, S., Li, N.: Runtime code generation in Cloudera Impala. IEEE Data Eng. Bull. 37(1), 31–37 (2014)
47. Zhang, R., Debray, S., Snodgrass, R.T.: Microspecialization: dynamic code specialization of database management systems. In: CGO, pp. 63–73 (2012)
48. Zhang, R., Snodgrass, R.T., Debray, S.: Microspecialization in DBMSes. In: ICDE, pp. 690–701 (2012)