ABSTRACT
To leverage modern hardware platforms to their fullest, more and more database systems embrace compilation of query plans to native code. In the research community, there is an ongoing debate about the best way to architect such query compilers. This is perceived to be a difficult task, requiring techniques fundamentally different from traditional interpreted query execution.

We aim to contribute to this discussion by drawing attention to an old but underappreciated idea known as Futamura projections, which fundamentally link interpreters and compilers. Guided by this idea, we demonstrate that efficient query compilation can actually be very simple, using techniques that are no more difficult than writing a query interpreter in a high-level language. Moreover, we demonstrate how intricate compilation patterns that were previously used to justify multiple compiler passes can be realized in one single, straightforward, generation pass. Key examples are injection of specialized index structures, data representation changes such as string dictionaries, and various kinds of code motion to reduce the amount of work on the critical path.

We present LB2: a high-level query compiler developed in this style that performs on par with, and sometimes beats, the best compiled query engines on the standard TPC-H benchmark.

CCS CONCEPTS
• Information systems → Database management system engines;
• Software and its engineering → Domain specific languages;

KEYWORDS
Query Compilation; Futamura Projections

1 INTRODUCTION
A typical database management system (DBMS) processes incoming queries in multiple stages (Figure 1.1): In the front-end, a parser translates a given SQL query into a logical plan. The query optimizer rewrites this plan into a more efficient form based on a cost model, and emits a physical plan ready for evaluation. The back-end is usually an interpreter that executes the optimized plan over the stored data, operator by operator, producing record after record of the computed query result.

Taking a slightly broader view, a database system fits – almost literally – the textbook description of a compiler: parser, front-end, optimizer, and back-end. The one crucial difference is that the last step, generation of machine code, is typically missing in a traditional DBMS.

For the longest time, this was a sensible choice: disk I/O was the main performance bottleneck. Interpretation of query plans provided important portability benefits, and thus, engineering low-level code generation was not perceived to be worth the effort [7]. But the circumstances have changed in recent years: with large main memories and storage architectures like NVM on the one hand, and the demand for computationally more intensive analytics on the other hand, query processing is becoming increasingly CPU-bound. As a consequence, query engines need to embrace full compilation to remain competitive, and therefore, compiling query execution plans (QEPs) to native code, although in principle an old idea, is seeing a renaissance in commercial systems (e.g., Impala [27], Hekaton [14], Spark SQL [6], etc.), as well as in academic research [13, 26, 33]. Given this growing attention, the database community has engaged in an ongoing discourse about how to architect such query compilers [26, 33, 44]. This task is generally perceived as hard, and often thought to require fundamentally different techniques than traditional interpreted query execution. The most recent contribution to this discourse from SIGMOD’16 [44] states that “it is fair to say that creating a query compiler that produces highly optimized code is a formidable challenge,” and that the “state of the art in query compiler construction is lagging behind that of the compilers field.”

At least in part, the compilers community is to blame for this situation, as they do not explain their key ideas, principles, and results clearly enough. Compilers and interpreters are among the most powerful tools computer science has to offer, but even graduate compiler classes and textbooks dwell on minuscule differences between variants of parsing and register allocation algorithms instead of making the pragmatics of building effective compilers accessible.
to a wide audience, and without conveying that in essence, many effective compilers can be very simple.

First of all, it is clear that a query compiler should reuse parts of an existing compiler back-end. There is no point for database developers to implement low-level functionality like register allocation and instruction selection for a range of different architectures (say, x86-64, POWER, and SPARC). Thus, the most sensible choice is to generate either C source or use a framework like LLVM [29] that provides an even lower-level entry point into an existing compiler toolchain. But how to get from physical query plans to this level of executable code? This is in fact the key architectural question. HyPer [33] uses the programmatic LLVM API and achieves excellent performance, at the expense of a rather low-level implementation that permeates large parts of the engine code.1 LegoBase [26] and DBLAB [44] are engines implemented in a high-level language (Scala [34]) that generate efficient C code. The latter two systems add significant complexity in the form of multiple internal languages and explicit transformation passes, which the authors of DBLAB [44] claim are necessary for optimal performance.

In this paper, we demonstrate that neither low-level coding nor the added complexity of multiple compiler passes are necessary. We present a principled approach to derive query compilers from query interpreters, and show that these compilers can generate excellent code in a single pass. We present LB2, a new query engine developed in this style that is competitive with HyPer and DBLAB.

The paper is structured around our specific contributions:

- In the spirit of explaining key compilers results more clearly, we draw attention to an important but not widely appreciated concept known as Futamura projections, which fundamentally links interpreters and compilers through specialization. We propose to use the first Futamura projection as the guiding principle in the design of query compilers (Section 2).
- We show that viewing common query evaluator architectures (pull-based, aka Volcano; and push-based, aka data-centric) through the lens of the first Futamura projection provides key insights into whether an architecture will lead to an efficient compiler. We use these insights to propose a novel data-centric evaluator model based on callbacks, which serves as the basis for our LB2 engine (Section 3).
- We discuss the practical application of the Futamura projection idea, and show how to derive high-level and efficient query compilers from a query interpreter. We implement a range of optimizations in LB2. Among those are row-oriented vs. column-oriented processing, data structure specialization, code motion, parallelization, and generation of auxiliary index structures including string dictionaries. The set of optimizations in LB2 includes all those implemented in DBLAB [44], and some en plus (e.g. parallelization). But in contrast to DBLAB, which uses up to 5 intermediate languages and a multitude of intricate compiler passes, LB2 implements all optimizations in a single generation pass, using nothing but high-level programming (Section 4).

---

1Of course, the internals of HyPer have changed a lot since the 2011 paper, and as we have learned from private communication with the HyPer team, internal abstractions have been added that are steps in the direction we propose. However, these have not been documented in published work.
shown in Figure 2b: if we have a program specializer, or partial evaluator, which for historical reasons is often called mix, then we can specialize the (query) interpreter with respect to a given source program (query). The result is a single-argument program that computes the query result directly on the data, and runs much faster than running the query through the original interpreter. This is because the specialization process strips away all the "interpretive overhead", i.e., dispatch the interpreter performs on the structure of the query. In other words, through specialization of the interpreter, we are able to obtain a compiled version of the given program!

This key result—partially evaluating an interpreter with respect to a source program generates a compiled version of that program—is known as the first Futamura projection. Less relevant for us, the second and third Futamura projections explain how self-application of mix can, in theory, derive a compiler generator: a program that takes any interpreter and produces a compiler from it.

Codd’s idea has been wildly successful, spawning multi-billion-dollar industries, to a large extent thanks to the development of powerful automatic query optimization techniques, which work very well in practice due to the narrow semantic model of relational algebra. Futamura’s idea of deriving compilers from interpreters automatically via self-applicable partial evaluation received substantial attention from the research community in the 1980s and 1990s [21], but has not seen the same practical success. Despite partial successes in research, fully automatic partial evaluation has not captured any kind of program analysis or optimization. It is important to note that the first Futamura projection itself does not capture any kind of program analysis or optimization. This poses the question how to generate optimized code. A key observation is that the shape of specialized code follows the shape of the interpreter that is specialized. Hence, we can derive the following design principle: By engineering the source interpreter in a certain way, we can control the shape of the compiled code.

In the following, we review popular query evaluation models with an eye towards specialization, and present our improved data-centric execution model.

**3 STRUCTURING QUERY EVALUATORS**

It is important to note that the first Futamura projection itself does not capture any kind of program analysis or optimization. This poses the question how to generate optimized code. A key observation is that the shape of specialized code follows the shape of the interpreter that is specialized. Hence, we can derive the following design principle: By engineering the source interpreter in a certain way, we can control the shape of the compiled code.

In the following, we review popular query evaluation models with an eye towards specialization, and present our improved data-centric execution model.

The **Iterator (Volcano) Model** is based on a uniform open(), next() and close() interface for each operator. Figure 3a-b shows a QEP, and the operator interface in Volcano [18]. Evaluation starts when the root operator (e.g., hash join) invokes next() to probe offspring operators for the next tuple. Subsequent operators (e.g., select) repeatedly invoke next() until a scan (or materialized state) is reached. At this point, a tuple is pipelined back to its caller and the operator’s code is executed. Thus, the mode of operation can be understood as pull-based. Although the iterator model is intuitive and allows pipelining a stream of tuples between operators, it incurs significant overhead in function calls, which one might hope to eliminate using compilation.

We follow Futamura’s idea and specialize the Volcano Select operator in Figure 3d to a given query, as illustrated in Figure 4b. However, the specialized code is inefficient. Each operator checks that the pulled record is not a null value, even though this check is really necessary only inside the Scan operator. These isNull(rec) conditions cannot be specialized away since they impose a control-flow dependency on dynamic data.
The Data-Centric (Produce/Consume) Model as introduced in HyPer [33], leads to better compilation results, and is therefore used by most query compiler developments, including LegoBase [26], DBLAB [44], and Spark SQL [6]. In this model, the control flow is inverted. Data are pushed towards operators, improving data and code locality. Operators that materialize tuples (e.g., aggregates and hash joins) are marked as pipeline breakers. As shown in Figure 3e, the operator interface consists of methods produce and consume. The producer’s task is to obtain tuples, e.g., from a scan or other interfacing operator in the query pipeline. The consumer carries out the operation, e.g., evaluating the predicate in Select.

Viewing the data-centric model through the lens of Futamura projections delivers the key explanation why it leads to better compilation results than the Volcano model: the inter-operator control flow does not depend on dynamic data, and hence specialization can fully remove the dispatch on the operator structure, leading to the tight residual code shown in Figure 4c. In contrast to the Volcano model, there are no null records since each operator in the pipeline invokes consume only on valid data.

3.1 Data-Centric Evaluation with Callbacks

For developers, the data-centric evaluation model is somewhat unintuitive since it spreads out query evaluation across produce and consume methods. But given that we have identified the desired specialization result, we can think about whether we can achieve the same specialization from a different API.

Figure 5a walks through the hash join evaluation in the data-centric model. First, the executor invokes HashJoin.open to initialize the hash join branches (i.e., left and right operators) followed by HashJoin.produce. Second, the produce method invokes produce on each branch (i.e., left.produce and right.produce). Thus, control moves to left.produce and perhaps invokes produce on that branch a few times until a scan or a pipeline breaker is reached. At this point, consume performs its actions and invokes parent.consume to dispatch a record to its consumer. When the control eventually returns back to HashJoin.consume, the record is added to the hash table. In the following step, right.produce is invoked to process the records from the right branch. As the example illustrates, it is not always clear how the produce and consume methods are behaving: consume will be called by actions triggered by produce. This becomes even more complex when the operator possesses multiple children. In this case consume behaves differently depending on which intermediate operator is pushing the data.

As our first contribution, we show that we can achieve the same functionality and specialization behavior by refactoring the produce and consume interface into a single method exec that takes a callback. This new model is directly extracted from the desired specialization shown in Figure 4c. Figure 5b shows how the hash join operator can be implemented using this new interface. The exec method does not require additional state. Each join branch is invoked using a different callback function; first the left, and then the right. This avoids the key difficulty in the produce/consume model, namely the conflation of phases in consume, and also the need to maintain parent in addition to child links. A variant of this model was presented as part of a functional pearl at ICFP ’15 [38]. Intuitively, the statement op.exec(cb) can be read as follows: operator op, generate your result and apply the function cb on each tuple.

```scala
// Schema: Dep(dname: String, rank: Int) // Specialized Volcano evaluation
var matchRec = 0 // scan state
val size = data.length
while (true) { // print loop
  val rec = {
    var recSel = null
    do { // select loop
      recSel = if (matchRec < size) data.nextRec else null
    } while (!isLeft(recSel) || !isRight(recSel))

    if (recSel == null) break
    println(rec.dname + "," + rec.rank)
  }
}
```

Figure 4: Specializing select query (a) in Volcano (b) and Data-centric (c)

```scala
class HashJoin(left: Op, right: Op) { // operator template
  (lkey: KeyFun) -> (rkey: KeyFun) extends Op {
    val hm = new HashMap()
    var isLeft = true
    var parent = null
    def open() = { // Step 1
      left.parent = this; right.parent = this
      left.open; right.open;
    }
    def produce() = {
      lLst = true; left.produce() // Step 2
      rLst = false; right.produce() // Step 4
    }
    def consume(rec: Record) = {
      if (isLeft) // Step 3
        hm = (lkey(rec), rec)
      else // Step 5
        for (lr <- hm(rkey(rec)))
          parent.consume(wrangle(lr, rec))
    }
  }
}
```

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

4 BUILDING OPTIMIZING QUERY COMPILERS

Having identified the general approach of deriving a query compiler from an interpreter (the first Futamura projection) using programmatic specialization as described in Section 2, and having identified the desired structure of our query interpreter in Section 3, we are now faced with the task of actually making it happen. Recall, obtaining a compiled query target from an interpreted query engine requires identifying three components: the staged interpreter, the static input and the dynamic input. Figure 6 shows LB2’s query evaluator, essentially an interpreter, that implements the data-centric evaluation with callbacks. We can now start specializing this query engine to emit source code in the same way we specialized the power function using the symbolic data type MyInt in Section 2. But where in a query engine should code generation be placed to minimize changes to the operator code?

Pure Template Expansion. As a first idea we could perform coarse-grained code generation at the operator level. Each operator is specialized as a string with placeholders for parameters. We show the aggregate operator as example:

```scala
class AggOp { @1(op: String): AggFun } extends Op { // operator template
  def exec(t: String) = s"""" + op.exec(t)
}
```

At runtime, this query evaluator performs a direct mapping from an operator to its code, i.e., substitutes op.exec with the operational code of the child operator op. Template expansion is easy to implement and removes some of the interpreter overhead. Still, the
approach is criticized as inflexible [44]. First, a query is generated exactly as written, with generic and inefficient data structure implementations. Second, cross-operator optimizations, data layout changes, etc., are all off-limits. Third, string templates are inherently brittle, and rewriting the core engine code as templates may introduce variable name clashes, type mismatches, etc., that may cause both subtle errors and hard crashes in generated code.

**Programmatic Specialization.** In order to avoid the problems of coarse templates, we can push specialization further down into the structures that make up the query engine in Figure 6, in particular the `Record` and various `HashMap` classes. The key benefit of this approach is that the main engine code in Figure 6 can remain unchanged.!

The generated code is the same as for operator templates, but all the code generation logic is now confined to `Record` and `HashMap`, with a much smaller surface exposure to bugs related to string manipulation. The implementation is as follows:

```scala
class MyInt {
  val name: String
  println(s"val $name = new MyInt($name)")
}
```

Still, the generated code uses unsophisticated `Record` and `HashMap` implementations, which will not exhibit optimal performance. When targeting C code, we will likely use an equivalent library such as GLib, which has high performance overhead.

**Optimized Programmatic Specialization.** For optimum performance, we want to implement specialized internal data structures instead of relying on generic libraries. To achieve this, we take the specialization idea one step further and push code generation even further down to the level of primitive types and operations. This means that not even our `Record` and `HashMap` implementations need to mention code generation directly, and as we will show in Sections 4.1-4.2, this will enable a range of important optimizations while retaining a high-level programming style throughout. This leads us to use exactly the `MyInt` class shown in Section 2, and, while it would be possible to build an entire query engine in this way, it pays off to use an existing code generation framework that already implements this low-level plumbing.

**Lightweight Modular Staging (LMS).** LB2 internally uses a library-based generative programming and compiler framework LMS [41] to encapsulate the code generation logic. LMS maintains a graph-like intermediate representation (IR) to encode high-level constructs and operations. Moreover, LMS provides a high-level interface to manipulate the IR graph. LMS distinguishes two types of expressions: present-stage expressions that are executed normally and future-stage expressions that are compiled into code. LMS defines a special type constructor `Rep[T]` to denote future-stage expressions, e.g., our `MyInt` corresponds to `Rep[Int]` in LMS, and given two `Rep[Int]` values `a` and `b`, evaluating the expression `a + b` will generate code to perform the addition. LMS provides implementations for all primitive `Rep[T]` types, i.e., strings, arrays, etc.

In addition, LMS also provides overloaded control-flow primitives, e.g., if (`c`) a else b where c is a `Rep[Boolean].`

LMS is a full compiler framework, which supports a variety of intermediate layers, but we only use it as a code generation substrate for our purposes. To draw a comparison with HyPer, LB2 is implemented in Scala instead of C++ and uses LMS instead of LLVM. While LLVM [29] operates on a lower level than LMS, the difference is not fundamental, and it is important to note that abstractions similar to those we propose can also be built on top of LLVM in C++, using standard operator overloading and lambda expressions, which have been available since C++11.

**Preliminary Example.** In order to better understand how a query is compiled using the optimized programmatic specialization, consider the following aggregate query and the query execution plan (QEP) (refer to Figure 6 for operators implementation).

In LB2, like in any other database, the query is represented by a tree of operators. Evaluation starts with the root operator in the QEP (i.e., `Print`). Calling `Print.exec` with an empty callback will call its child operator’s `Agg.exec` method with a callback that encodes evaluation actions to be carried out on the records that `Agg` produces: in this case, just printing out the result. After that, `Agg.exec` calls `Scan.exec` with a callback tailored to the aggregate operation and the callback it received from `Print`. Based on Futamura projections discussed in Section 2, the result of executing a staged query interpreter is a residual program that implements the query evaluation on record fields where all abstractions are optimized away and indirect control flow removed. Appendix B.2 gives detailed code generation steps along with the generated C code in Figure 14.

For the remainder of this section, we discuss how interesting performance optimizations for data structures, storage layout, memory management, etc., are implemented in LB2 while retaining the single code generation pass architecture.

### 4.1 Row or Column Layout

Typically, query engines either support row-oriented or column-oriented storage. Each data layout works better in one situation than the other, so it is attractive to be able to support both within the same query engine.

In LB2, the `Record` class is the entry point to query engine specialization. Record is an abstract class that contains a schema (a sequence of named `Field` attributes) and supports lookup of `Values` by name. It is important to note that the schema is entirely static, i.e., only exists at code generation time, while subclasses of `Value` carry actual `Rep[T]` values, i.e., dynamic data that exists at query evaluation time. Subclasses of `Record` can support either flat, row-oriented, storage through a base pointer (class `NativeRec`), or abstract over the storage model entirely by referencing individual values directly from a record in a *scalared* form, mapped to local variables in generated code (class `ColumnRec`).

```scala
abstract class Field {
  val name: String
  // models an attribute's name and type
  case class IntField(name: String) extends Field // Int type attribute
  case class StringField(name: String) extends Field // String type attribute
}

abstract class Rep[T] {
  // denotes a future-stage expression
  case class MyInt extends Rep[Int]
  case class Rep[T](v: Rep[T]) extends Rep[T]
  case class Rep[T](v: Rep[T]) extends Rep[T]
}
```

Abstract class `Value` models an attribute’s value:

```scala
abstract class Value {
  // models an attribute's value
  case class IntValue(value: Rep[Int]) extends Value (..., operations) // operations elided
  case class StringValue(value: Rep[String], length: Rep[Int]) extends Value (..., operations)
}
```
The internal implementations are unsurprising, and exactly what
abstract class HashJoin(left: Op, right: Op)(lkey: KeyFun)
case class HashJoin(left: Op, right: Op)(lkey: KeyFun)
    extends Op {
    def exec(ctx: Record => Unit) = {
        val res = new HashMultiMap()
        for (tuple <- hm)
            hm.update(tuple, res) = {
                val key = apply(tuple)
                res.add(key, tuple)
            }
    }
    def filter(tup: Tuple): Unit = {
        if (tup matches lkey)
            res.add(tup, key)
    }
    def foreach(f: Record => Unit): Unit = {
        for (tuple <- hm)
            f.apply(tuple)
    }
    def foreach(f: Record => Unit): Unit = {
        for (tuple <- hm)
            f.apply(tuple)
    }
    def foreach(f: Record => Unit): Unit = {
        for (tuple <- hm)
            f.apply(tuple)
    }
}

class HashJoin(left: Op, right: Op)(lkey: KeyFun)
    extends Op {
    def exec(ctx: Record => Unit) = {
        val res = new HashMultiMap()
        for (tuple <- hm)
            res.add(tuple, key)
    }
    def filter(tup: Tuple): Unit = {
        if (tup matches lkey)
            res.add(tuple, key)
    }
    def foreach(f: Record => Unit): Unit = {
        for (tuple <- hm)
            f.apply(tuple)
    }
    def foreach(f: Record => Unit): Unit = {
        for (tuple <- hm)
            f.apply(tuple)
    }
}

Likewise, LB2 abstracts over storage and computation through an abstract
class Buffer, with implementation classes for row-oriented (FlatBuffer) and
column-oriented storage (ColumnarBuffer):
abstract class Buffer(s: Seq[Field], size: Long) extends Record {
    val size = defaultSize
    val keys = new ColumnarBuffer(s, size)
    val used = new Array[Int](size)
    var next = 0
    def update(k: Record, init: Record)(up: Record => Record): Unit = {
        keys(k) = used(k) = next
        next = up(used)(next) + 1
    }
    def foreach(f: Record => Unit): Unit = {
        for (idx <- 0 until next) {
            f.apply(keys(idx))
        }
    }
}

The internal implementations are unsurprising, and exactly what
one might write in a high-level query interpreter without much
concern for low-level performance. But it is notable to
remark that there is never any code like new Record(...)
for lambda expressions. The hash map is fully specialized for key and values.
The hash join operator uses a HashMultiMap, which differs from
this code in its interface, and also in its internal implementation (we
use linked buckets instead of open addressing). We elide the code
for space reasons—it follows the same high-level of abstraction,
and if we wish, we can pull out some aspects of the two hash map
classes into a common base class, again without any runtime cost.

4.3 Data Partitioning and Indexing

Query engines use statistics and metadata to determine when
an index can be used to speedup query execution. We assume
that these decisions are made during the query planning and
optimization phase. To add index capabilities to LB2, we provide
a corresponding set of indexed query operators in the same style
as those defined in Figure 6. The code below shows an index join.

The operator interface is extended with a get method, which
enables IndexJoin.exec to find tuples that match the join key:

IndexJoin.exec (op: Op)(k: KeyFun)(sk: Sequence[Field]) extends Op {
    def exec(ctx: Record => Unit) = {
        val res = new HashMultiMap()
        for (tuple <- hm)
            res.add(tuple, key)
    }
    def foreach(f: Record => Unit): Unit = {
        for (idx <- 0 until next) {
            f.apply(keys(idx))
        }
    }
    def foreach(f: Record => Unit): Unit = {
        for (idx <- 0 until next) {
            f.apply(keys(idx))
        }
    }
}

LB2 realizes passive and dense index data structures for primary and
foreign keys on top of the abstractions presented in Sections 4.1
and 4.2, behind a uniform Index interface. The IndexEntryView class
enables iterating over index lookups via foreach. Method exists is
used by IndexSemiJoin and IndexAntiJoin operators.
In addition to query evaluation, LB2 generates data loading code for different storage modes, which we extend to create index structures. These can serve as additional access paths on top of underlying abstract class DicField(name: String)

```scala
def get(idx: DicValue): StringValue = convert(string: StringValue)
```

`class DicField(name: String)) extends Field {
  def get(idx: DicValue): StringValue = convert(string: StringValue)
```

\} 
```scala
// ... print records ...
```

\}) } 
``` scala
def get(string: StringValue): DicValue = StringValueComparer.convertAs(dicValue: DicValue) 
```

Consider the simple case of compressing a single string column.

At loading time, the StringDict is loaded from memory. When the loader reads a string, it creates a StringValue and uses the StringDict to convert it into its DicValue compressed form: the index where the StringValue is stored inside the StringDict.

While string dictionaries may speed up most string operations, their use requires some care. The comparison of two strings in `DicValue` index where the `StringValue` is to extract the string representation. Moreover, some operations generate strings at runtime (e.g., substring).

In the event of a comparison between string belonging to two different dictionaries, the fallback is to extract the StringValue and perform the operation on the string representation. Moreover, some operations generate strings at runtime (e.g., substring). In that case, uncompressed strings need to be used as well. Conceptually, LB2 can support both implementations. The `DicField` class keeps track of the dictionary associated with a given attribute, which allows LB2 to generate compressed string operations where possible and uncompressed operations otherwise. Finally, string dictionaries do not add new query operators, i.e., they operate transparently as part of the data representation layer.

\} 
``` scala
// ... print records ...
```

\}) } 
``` scala
def get(string: StringValue): DicValue = StringValueComparer.convertAs(dicValue: DicValue) 
```

Consider the simple case of compressing a single string column.

At loading time, the StringDict is loaded from memory. When the loader reads a string, it creates a StringValue and uses the StringDict to convert it into its DicValue compressed form: the index where the StringValue is stored inside the StringDict.

While string dictionaries may speed up most string operations, their use requires some care. The comparison of two strings in `DicValue` index where the `StringValue` is to extract the string representation. Moreover, some operations generate strings at runtime (e.g., substring).

In the event of a comparison between string belonging to two different dictionaries, the fallback is to extract the StringValue and perform the operation on the string representation. Moreover, some operations generate strings at runtime (e.g., substring). In that case, uncompressed strings need to be used as well. Conceptually, LB2 can support both implementations. The `DicField` class keeps track of the dictionary associated with a given attribute, which allows LB2 to generate compressed string operations where possible and uncompressed operations otherwise. Finally, string dictionaries do not add new query operators, i.e., they operate transparently as part of the data representation layer.

\} 
``` scala
// ... print records ...
```

\}) } 
``` scala
def get(string: StringValue): DicValue = StringValueComparer.convertAs(dicValue: DicValue) 
```
The communication between operators is illustrated in the drawing above. The downstream client of parSelect initiates the process by calling exec, which parSelect forwards upstream to ParScan. ParScan starts a number of threads, and on each thread, calls the exec callback with the thread id `tid` and another callback `dataloop`. This will allow the downstream operator to initialize the appropriate thread-local data structures. Then the downstream operator triggers the flow of data by invoking `dataloop`, and passing another callback upstream, on which the ParScan will send each tuple for the data partition corresponding to the active thread.

While the `parallelPipeline` transformation covers the simpler state-less operators, some extra work is required for pipeline breakers. The callback interface makes sophisticated threading schemes possible, in an elegant manner. For operators such as `Agg`, LB2’s parallel implementations split their work internally across multiple threads, accumulating final results, etc. By using callbacks in a clever way, we can delegate some of the synchronization effort to specialized parallel data structures. In the case of `Agg`, LB2 uses a `ParHashMap` abstraction, which internally has to enforce thread safety. Multiple implementations are possible, using synchronization primitives, partitioning, or an internal lock-free design.

```
abstract class ParHashMap[T: Long, kSche: Seq[Field], vSche: Seq[Field]] {
  def apply(tId: Rep[Int]): DataLoop
  def merge(init: Record, agg: AggFun): Unit
  def partition(tid: Rep[Int]): DataLoop
}
```

This design is powerful enough to encapsulate all subtle issues related to multi-threading. Similar to the code motion idea, we use the callbacks to re-organize the operator code into different parts:

```
class Agg(grp: GrpFun)(init: Record)(agg: AggFun) extends ParOp {
  def exec = {
    val opExec = op.exec
    val hm = new ParHashMap(tid, grp.schema, agg.schema)
    (tId: ThreadCallback) => {
      opExec ( (tId: Rep[Int]) => (dataLoop: DataLoop) => // parallel section starts
        hm.update(k, init) { c => agg(c, tuple) }
      ) // parallel section ends
    }
    hm.merge(init, agg) // merge the results across threads
  }
}
```

The distinct parts are the global initialization of the data-structure, the local initialization for each thread, the computation, and finally the merging between threads, followed by the beginning of a new parallel pipeline.

### 4.6 Comparison with a Multi-Pass Compiler

We contrast LB2’s implementation with a recent multi-pass query compiler, DBLAB [44]. Both systems aim to implement a query interpreter in a high-level language once and control query compilation by modifying parts of the query engine. In LB2, we modified the core abstractions underneath the main engine code to add code generation. In DBLAB, the query engine code itself is transformed to lower-level code, through multiple intermediate stages. For each optimization, an analysis pass identifies pieces of code to be rewritten followed by one or more re-writing passes. DBLAB offers a flag per optimization that can be configured for each query. We compare how key optimizations are implemented in both systems.

**Data Layout.** The default data layout in DBLAB is row-oriented. DBLAB supports column-oriented layout on a best effort basis using a compiler pass that converts arrays of records to a record of arrays where possible. In LB2, operators decide which storage layout to use by instantiating one of several implementation classes, e.g., `FlatBuffer` or `ColumnarBuffer` as discussed in Section 4.1. These decisions can be made based on input from the query optimizer.

**Data Structure Specialization.** DBLAB introduces a number of intermediate abstraction levels (referred to as stack-of-DSLs) and defines optimizations that can be performed at each level. Specializing high level data structure, e.g., hash maps, into native arrays takes one analysis pass and up to three re-writing passes. In LB2, the same transformation is achieved by implementing hash maps as generation-time abstractions, which ensures that only native array operations are generated. Adding a new hash map variant requires a high-level implementation in LB2, using normal object-oriented techniques (see Section 4.2). For example, LB2 uses open addressing for aggregates and linked hash buckets for joins. In DBLAB, all analysis and transformation passes are specific to a linked-bucket hash table implementation. Adding a variant based on open addressing would require an entire new set of analysis and transformation passes.

**Index Structures.** DBLAB’s makes indexing decisions not based on query plans, but on a lowered version of the query engine code after inlining the operators. A first analysis extracts hash join patterns and evaluates whether an index can be instantiated. In some sense, this appears to be putting the cart before the horse, because high-level query plan structure needs to be reverse-engineered from comparatively low-level code. A second analysis rule determines the type of index, and a third rule determines whether the hash map is collision-free on the hash function and hence can use one dimensional arrays. Finally, a rewrite rule updates the generated code to use the index, and inserts index creation code into the loading phase. This automatic index inference in DBLAB is a global approach that always creates indexes without reasoning about the index cost or whether a non-index plan could be more efficient. LB2 does not attempt to infer indexes automatically and instead delegates such decisions to the query optimizer.

For string dictionaries, DBLAB performs a compiler pass to create string dictionaries on all string attributes and hoists the allocation statements to loading time. The rewrite rule identifies the type of string operation and updates the code accordingly. Again, LB2 does not attempt to infer string dictionary usage from low-level code, but assumes that this information is already available. Instead of transforming code, LB2 uses corresponding implementation classes as discussed in Section 4.3.

**Code motion.** DBLAB performs detailed analysis to collect data structures allocation statements along with dependent statement. After that, a rewrite rule moves allocation to loading time. Hence, hoisting is sensitive to order. As illustrated in Section 4.4 LB2’s
In this section, we evaluate the performance of LB2 on the standard TPC-H benchmark with scale factor SF10. We compare LB2 with Postgres [2] and two recent state-of-the-art compiled query engines: Hyper [33], and DBLAB [44]. Hyper implements compilation using LLVM and DBLAB is a multi-pass query compiler that generates C.

**Configurations.** We present three sets of experiments. The first set evaluates the performance of LB2 with only those optimizations that are compliant with the official TPC-H rules. The second set focuses on comparing optimizations that replicate data and create auxiliary indexes (Section 4) with their counterparts in DBLAB. Finally, the last set evaluates parallelism in LB2 and Hyper when scaling up the number of cores (DBLAB only runs on a single core). We run each query five times and record the median reading. For DBLAB and LB2, we use numactl to bind the execution to one CPU. Hyper provides a flag for the same purpose PARALLEL-off.

Query plans in LB2 and DBLAB are supplied explicitly while Hyper and Postgres implement a cost-based query optimizer. Since it is difficult to unify query plans across all systems, we report two sets of results for LB2. The line LB2 (DBLAB plan) uses DBLAB’s plans and LB2 (Hyper plan) uses Hyper’s plans to the extent possible but at least with the same join ordering. We choose not to turn indexing off in Hyper to allow the query optimizer to pick the best plan. Also, DBLAB replaces the outer join in Q13 with a hard-coded imperative array computation using side effects that is neither expressible in SQL nor in their internal query plan language.

DBLAB offers close to 30 configuration flags to enable/disable optimizations. In the first experiment, we use the -compliant option (a subset of compliant configurations per query). In the second experiment, we use the compliant configuration with the -m-part flag, DBLAB/LB 4 and DBLAB/LB 5 configurations respectively to enable indexing, date indexing with partitioning and string dictionaries as described in [42, 44]. We compare each of these configurations to the corresponding one in LB2.

**Performance Evaluation Parameters.** The first experiment (i.e., TPC-H compliant runtime) and the third experiment (i.e., parallelism) use the absolute runtime as key evaluation parameter. The second experiment evaluates the impact of individual optimizations that are pre-computations. In this case, we also report the overhead introduced by all these pre-computations relative to the loading time of LB2 without any index generation (fastest loading).

**Experimental Setup.** All experiments are conducted on a single machine with 4 Xeon E7-8890v4 CPUs, 18 cores and 256GB RAM per socket (1 TB total). The operating system is Ubuntu 14.04.1 LTS. We use Scala 2.11, Postgres 9.4, Hyper v0.5-222-g04766a1, GCC 4.8 with optimization flag -O3 and Glib library 2.0. We tried different versions of GCC and Clang with very similar results. We use the version of DBLAB released by the authors as part of the SIGMOD ‘16 artifact evaluation process [42].

### 5.1 TPC-H Compliant Runtime

In the first experiment, we compare LB2 with Postgres, DBLAB and Hyper on a single core under the TPC-H compliant settings. Figure 8 reports the absolute runtime for all TPC-H queries. We follow [44] in evaluating DBLAB without any indexing and use the same configuration for LB2. We did not disable primary key indexing in Hyper, as doing so appears to lead to very suboptimal query plans. In the plans reported here, Hyper employed one or more index joins in Q2, Q8-Q10, Q12, and Q21. Postgres is a Volcano-style interpreted query engine that is representative of wide-spread traditional systems.

At first glance, LB2 outperforms Postgres and DBLAB in all queries where query plans are matched. Furthermore, LB2 and Hyper’s performance is comparable. On a query by query analysis, LB2 outperforms DBLAB in aggregate queries Q1 and Q6 by 70% and 4% respectively. On join queries Q3, Q5, Q10, etc. LB2 is 3×-13× faster than DBLAB. Similarly, LB2 is 5×-13× faster in semi

---

2At the time of writing, Hyper binaries are no longer publicly available. We use a version obtained in late 2015.

3The generated C files for TPC-H queries submitted in [42] do not include an equivalent configuration for (compliant+index) and also use nonstandard string constants, e.g., in Q3, Q7, etc. For a uniform comparison, we re-generated the C files using [43].

---

![Figure 8: The absolute runtime in milliseconds (ms) for DBLAB, LB2 (with DBLAB’s plans), HyPer, LB2 (with HyPer’s join ordering plans) in TPC-H SF10. Only TPC-H compliant optimizations are used.](image-url)
Index Optimizations

The second experiment focuses on evaluating three advanced optimizations that were used by DBLAB to justify a multi-pass compiler pipeline [44]: primary and foreign key indexes, date indexes, and string dictionaries. In their full generality, these optimizations are not compliant with the TPC-H rules [11] since they incur pre-computation and potentially a duplication of data. While a subset of these optimizations that indexes data uniformly across all queries would be allowed by the TPC-H spec, we do not evaluate this setting for consistency with previously published DBLAB configurations [42, 44]. The results of this experiment are shown in Figures 9 and 10. The first table and graph give the absolute runtime of the TPC-H queries with different levels of indexing enabled: primary/foreign key, date columns, and string dictionaries. The second graph shows the overhead on loading time associated with creating these indexes.

Primary and Foreign Key Indexes. The results for this configuration are shown in line DBLAB/LB2-idx. Recall that DBLAB analyzes intermediate code to decide on which indexes it will create. LB2 makes those decisions based on the query plan. For the purpose of this experiment, we have tuned LB2’s decision rules to lead to the same decisions as DBLAB. We observe that DBLAB’s index cost is greater than LB2’s in all queries with Q5, Q12 and Q3 as the top three. In these queries, a hash map index is created on a sparse dictionary. The second graph shows the overhead on loading time associated with creating these indexes.

5.2 Index Optimizations

The second experiment focuses on evaluating three advanced optimizations that were used by DBLAB to justify a multi-pass compiler pipeline [44]: primary and foreign key indexes, date indexes, and string dictionaries. In their full generality, these optimizations are not compliant with the TPC-H rules [11] since they incur pre-computation and potentially a duplication of data. While a subset of these optimizations that indexes data uniformly across all queries would be allowed by the TPC-H spec, we do not evaluate this setting for consistency with previously published DBLAB configurations [42, 44]. The results of this experiment are shown in Figures 9 and 10. The first table and graph give the absolute runtime of the TPC-H queries with different levels of indexing enabled: primary/foreign key, date columns, and string dictionaries. The second graph shows the overhead on loading time associated with creating these indexes.

Primary and Foreign Key Indexes. The results for this configuration are shown in line DBLAB/LB2-idx. Recall that DBLAB analyzes intermediate code to decide on which indexes it will create. LB2 makes those decisions based on the query plan. For the purpose of this experiment, we have tuned LB2’s decision rules to lead to the same decisions as DBLAB. We observe that DBLAB’s index cost is greater than LB2’s in all queries with Q5, Q12 and Q3 as the top three. In these queries, a hash map index is created on a sparse dictionary. The second graph shows the overhead on loading time associated with creating these indexes.

5.2 Index Optimizations

The second experiment focuses on evaluating three advanced optimizations that were used by DBLAB to justify a multi-pass compiler pipeline [44]: primary and foreign key indexes, date indexes, and string dictionaries. In their full generality, these optimizations are not compliant with the TPC-H rules [11] since they incur pre-computation and potentially a duplication of data. While a subset of these optimizations that indexes data uniformly across all queries would be allowed by the TPC-H spec, we do not evaluate this setting for consistency with previously published DBLAB configurations [42, 44]. The results of this experiment are shown in Figures 9 and 10. The first table and graph give the absolute runtime of the TPC-H queries with different levels of indexing enabled: primary/foreign key, date columns, and string dictionaries. The second graph shows the overhead on loading time associated with creating these indexes.

Primary and Foreign Key Indexes. The results for this configuration are shown in line DBLAB/LB2-idx. Recall that DBLAB analyzes intermediate code to decide on which indexes it will create. LB2 makes those decisions based on the query plan. For the purpose of this experiment, we have tuned LB2’s decision rules to lead to the same decisions as DBLAB. We observe that DBLAB’s index cost is greater than LB2’s in all queries with Q5, Q12 and Q3 as the top three. In these queries, a hash map index is created on a sparse dictionary. The second graph shows the overhead on loading time associated with creating these indexes.

Primary and Foreign Key Indexes. The results for this configuration are shown in line DBLAB/LB2-idx. Recall that DBLAB analyzes intermediate code to decide on which indexes it will create. LB2 makes those decisions based on the query plan. For the purpose of this experiment, we have tuned LB2’s decision rules to lead to the same decisions as DBLAB. We observe that DBLAB’s index cost is greater than LB2’s in all queries with Q5, Q12 and Q3 as the top three. In these queries, a hash map index is created on a sparse dictionary. The second graph shows the overhead on loading time associated with creating these indexes.

Figure 9: The absolute runtime in milliseconds (ms) after enabling non-TPC-H-compliant indexing, date indexing and string dictionary in SF10 using DBLAB plans
When generating the C code for Q13 in this experiment, with 2 to 8 cores. In outer join Q13, LB2 is 10%-20% faster than DBLAB, whereas difference in implementation and parallel data structures can result in faster code. In terms of implementation, LB2 generates C code and realizes parallelism in a high-level style using OpenMP. HyPer generates LLVM and uses Pthreads for fine-grained (Morsel-driven [30]) parallelism.

Conclusion The results of our experiments show that LB2 can compete against the state-of-the-art query compilers. However LB2’s design is simpler than both DBLAB and HyPer; it is derived from a straightforward query interpreter design and does not require multiple compiler passes or additional intermediate languages.

6 RELATED WORK

The Origin of Compiling Query Engines dates back to IBM’s System R [7]; the initial design realized a form of template-based code generation. However, compilation was abandoned for interpretation, mainly for improved portability and maintenance. More than two decades later, AT&T developed Daytona [19] that translates its high-level query language Cymbal into C. IBM compiled queries to Java bytecode [37] by removing virtual functions from iterator evaluation. The performance gain was limited since the iterator model does not lend itself to efficient low-level optimizations.

Compiled Query Engines. The recent changes in architecture and the push towards main memory databases have revived interest in developing efficient query compilation methods. The timeline in Figure 12 shows a selection of query engines that employ a form of query compilation since System R. The illustration broadly classifies the compilation method used. An arrow from A to B denotes that system B is built on or extends system A. HIQUE [28] performs operator re-ordering, data staging, and generates query code through template expansion. HyPer [33], RAW [25], H2O [5], Radish [32], ViDa [24], Voodoo [36] and others [10, 23, 35] follow a low-level approach to query compilation where LLVM assembly is intermixed with pre-compiled C++. The pre-compiled code implements complex data structures and the LLVM assembly manages the tuple level work. DBToaster [4] uses compilation and recursive delta queries to realize high-order incremental view maintenance (IVM). Inside Postgres [23] compiles query subexpressions into machine code and [45] uses program specialization inspired by Futamura projections and LLVM to generate query code.
7 DISCUSSION

In this section, we draw some general insights and aim to clarify some specific points of discussion, notably regarding single-pass vs. multi-pass compilers.

Templates Expansion vs. Many Passes: A False Dichotomy. The authors of DBLAB [44] motivate their multi-pass architecture by claiming that “all existing query compilers are template expanders at heart” and that “template expanders make cross-operator code optimization impossible”. But this argument confuses first-order, context-free, template expansion with the richer class of potentially more sophisticated single-pass compilers, which can very well perform context-dependent optimizations. Especially in the context of Futamura projections, we have shown that the structure of the query interpreter that is specialized into a query compiler has a large effect on the style of generated code, and can achieve all the optimizations implemented in DBLAB.

Are Many Passes Necessary? Staying with the specialization idea of Futamura projections, we can specialize away further abstraction levels without auxiliary transformation passes. Hence, we demonstrate that for standard relational workloads, no additional intermediate languages besides the query plan language and the generated C level are necessary.

Do Many Passes Help or Hurt? Multiple passes help if one step of expansion/specialization can expose information to an analysis that is not present in the source language. For database queries, the source language of query plans is already designed to contain all relevant information. If some information is missing, it can likely be added to the execution plan language. Hence, we find that additional abstraction levels in between relational operators and C code do not provide tangible benefits. To the contrary, some information may get lost and has to be arduously recovered. Multiple transformation passes depend on analysis of imperative code that manipulates low-level data structures, which is notoriously difficult.

In contrast, the database community has already solved the query optimization problem for interpreted engines, and cost-based optimizers that produce good plans are available. In particular, deciding when an index can be used to speed up a join query is readily solved by looking at the query plan. Trying to make such a decision by analyzing low-level code generated from a physical plan (as DBLAB does) seems overall backwards, and unlikely to scale to realistic use cases.

When to use Multiple Intermediate Languages? We have argued that we do not need stacks of multiple intermediate languages for compiling relational query plans. So when do we need them? A key use case is in combining multiple front-end query languages (DSLs), e.g., SQL and a DSL for machine learning or linear algebra (as in Delite’s OptiQL and OptiML [47]). First, domain-specific optimizations need to be applied on each DSL independently e.g., arithmetic simplification, join ordering, etc. After that, a series of compiler transformations may be needed to translate both DSLs into a common intermediate core, where loop fusion and other compiler level optimizations can be performed before code generation.

In a relational system, the query plan DSL is essential. We have to perform cost-based optimization of join ordering before we can think about generating code. The situation is similar in a linear algebra DSL, where we have to perform arithmetic optimizations before switching representations from matrices and vectors to loops and arrays.

8 CONCLUSIONS

In this paper, we advocate that query compilation need not be hard and that a low-level coding style on the one hand and the added complexity of multiple compiler passes and intermediate languages on the other hand are unnecessary. Drawing on the old but underappreciated idea of Futamura projections, we have presented LB2: a fully compiled query engine, and we have shown that LB2 performs on par with, and sometimes beats, the best compiled query engines on the standard TPC-H benchmark. Specifically, LB2 is the first query engine built in a high-level language that is competitive with HyPer [33], both in sequential and parallel execution. LB2 is also the first single-pass query engine that is competitive with DBLAB [44] using the full set of non-TPC-H-compliant optimizations. In conclusion, we demonstrate that highly efficient query compilation can be simple and elegant, with not significantly more implementation effort than an efficient query interpreter.

ACKNOWLEDGMENTS

We thank Kunle Olukotun for providing access to computing resources at Stanford and Viktor Leis for insightful discussions. Parts of this research were supported by NSF awards 1553471 and 1564207, and DOE award DE-SC0018050.
As demonstrated in Sections 2-4, LB2 compiles queries to native code by executing a staged query interpreter that emits code as a side effect. In this section, we walk through two concrete examples that illustrate key steps in this process, the `power` function from Section 2 and the aggregate query from Section 4. We introduce a double bracket notation `[]` to show intermediate states of the execution; we can think about these brackets as visualizing the execution stack as program expressions that still need to be executed. It is important to note that the intermediate steps are observable only indirectly—LB2 generates the full code in a single pass, but achieves similar transformations as multi-pass compilers (e.g., DLAB) without representing intermediate steps explicitly in an intermediate language.

B.1 Example: Power

In this section, we visualize the intermediate states of generating code for the `power` function discussed in Section 2. The following code shows the `power` function and `MyInt` class implementation:

```java
// symbolic integer type
class MyInt(ref: String) {
  def *(y: MyInt) = {
    if (1 == 0) 1
    else new MyInt(y.toString)
  }
  def + (y: MyInt) = {
    if (2 == 0) 1
    else new MyInt(s"1 + \$y")
  }
  def - (y: MyInt) = {
    if (3 == 0) 1
    else new MyInt(s"1 - \$y")
  }
  def / (y: MyInt) = {
    if (4 == 0) 1
    else new MyInt(s"1 / \$y")
  }
  constInt(x: Int) =
  new MyInt(x.toString)
}
```

The `power` function is recursive, with the bottom case reached when `n` becomes zero. Following the standard call-by-value evaluation rules, steps 1-8 show how `power` is expanded for `n = 4, 3, . . . , 1`.

1. `power(new MyInt("1"), 4)`
2. `[power(new MyInt("1"), 3) + power(new MyInt("1"), 3))]
3. `new MyInt("1") + power(new MyInt("1"), 3)]]
4. `new MyInt("4")`
5. `new MyInt("4") + power(new MyInt("1"), 2)]]
6. `new MyInt("4") + new MyInt("1")]
7. `new MyInt("4") + new MyInt("1") + power(new MyInt("1"), 1)]
8. `new MyInt("4") + new MyInt("1") + new MyInt("1") + new MyInt("1") + new MyInt("1")]

When `power` is invoked with `n = 0`, we reach the bottom of the recursion, i.e., no further function calls will be pushed on the call stack. It is the time to go up and perform the remaining evaluation actions, which will generate code.

#### Table 1: Lines of code needed to add each optimization

<table>
<thead>
<tr>
<th></th>
<th>LB2-comp</th>
<th>LB2-opt</th>
<th>DLAB-comp</th>
<th>DLAB-opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q1</td>
<td>243</td>
<td>178</td>
<td>265</td>
<td>324</td>
</tr>
<tr>
<td>Q2</td>
<td>560</td>
<td>488</td>
<td>568</td>
<td>699</td>
</tr>
<tr>
<td>Q3</td>
<td>381</td>
<td>363</td>
<td>363</td>
<td>363</td>
</tr>
<tr>
<td>Q4</td>
<td>224</td>
<td>224</td>
<td>224</td>
<td>224</td>
</tr>
<tr>
<td>Q5</td>
<td>444</td>
<td>458</td>
<td>458</td>
<td>458</td>
</tr>
<tr>
<td>Q6</td>
<td>477</td>
<td>477</td>
<td>477</td>
<td>477</td>
</tr>
<tr>
<td>Q7</td>
<td>427</td>
<td>427</td>
<td>427</td>
<td>427</td>
</tr>
<tr>
<td>Q8</td>
<td>302</td>
<td>292</td>
<td>292</td>
<td>292</td>
</tr>
<tr>
<td>Q9</td>
<td>356</td>
<td>347</td>
<td>347</td>
<td>347</td>
</tr>
<tr>
<td>Q10</td>
<td>197</td>
<td>187</td>
<td>187</td>
<td>187</td>
</tr>
<tr>
<td>Q11</td>
<td>147</td>
<td>139</td>
<td>139</td>
<td>139</td>
</tr>
<tr>
<td>Q12</td>
<td>322</td>
<td>311</td>
<td>311</td>
<td>311</td>
</tr>
<tr>
<td>Q13</td>
<td>286</td>
<td>295</td>
<td>295</td>
<td>295</td>
</tr>
<tr>
<td>Q14</td>
<td>171</td>
<td>164</td>
<td>164</td>
<td>164</td>
</tr>
<tr>
<td>Q15</td>
<td>366</td>
<td>358</td>
<td>358</td>
<td>358</td>
</tr>
<tr>
<td>Q16</td>
<td>228</td>
<td>254</td>
<td>254</td>
<td>254</td>
</tr>
<tr>
<td>Q17</td>
<td>387</td>
<td>379</td>
<td>379</td>
<td>379</td>
</tr>
<tr>
<td>Q18</td>
<td>398</td>
<td>385</td>
<td>385</td>
<td>385</td>
</tr>
<tr>
<td>Q19</td>
<td>211</td>
<td>201</td>
<td>201</td>
<td>201</td>
</tr>
<tr>
<td>Q20</td>
<td>316</td>
<td>302</td>
<td>302</td>
<td>302</td>
</tr>
<tr>
<td>Q21</td>
<td>231</td>
<td>214</td>
<td>214</td>
<td>214</td>
</tr>
<tr>
<td>Q22</td>
<td>316</td>
<td>292</td>
<td>292</td>
<td>292</td>
</tr>
</tbody>
</table>

#### Figure 13: Code generation and compilation for LB2 and DLAB
The base case of `power` returns the value 1. Next, the multiplication operator in `MyInt("x1") * 1` emits the first line `int x0 = in * 1`, as side effect, and returns `MyInt("x0")`.

Similarly, the remaining multiplication expressions in steps 13-15 generate the rest of the code.

Once the full code is generated, the result is stored in the variable `x3`, and the final return value will be `MyInt("x3")`.

### B.2 Example: Aggregate Query

Following the programmatic specialization ideas discussed in Section 4, we expand the preliminary example presented there and show how query code is generated for the following aggregate query end to end:

```scala
// Schema edname: String, eid: Int
Emp = Buffer() // preloaded

// generating a hash map data structure
hashmap = HashMap(Seq(StringField("edname")), Seq(IntField("count")))

// expanding the aggregate operator with Scan as a child and printRec as a callback
Print(Agg(Scan("Emp"))(x => x("edname")))(0)((agg,x) => agg + 1)).exec(t => ())
```

We observe that the generated Scala code is not yet fully optimized. The hash map and the update operation are generated in a high-level form. As a consequence, generating the equivalent C code would rely on a library such as GLib to provide the underlying data structure code. In the following optimized programmatic specialization approach, we walk through specializing the `Emp` table, `Record` and `HashMap` classes to generate optimized C code.

### Optimized Programmatic Specialization (Native C)

We use the same example query but now we use the specialized `Record` and `HashMap` implementations to generate optimized code in C, as in Sections 4.1 and 4.2. The starting point is the same.

```c
// Schema edname: String, eid: Int
0 [ val Emp = Buffer() // preloaded
Print(Agg(Scan("Emp"))(x => x("edname")))(0)((agg,x) => agg + 1)).exec(t => ()) ]
```

The first expansion step generates code for the `Buffer` that contains the table `Emp`. Given a column-oriented storage, each attribute (edname and eid) is generated as one-dimensional array.

```c
1 [ // generating Emp table as two flat arrays: ednames and eids
Emp = malloc(sizeof(Emp))
int table_length = ... // preloaded
cfor (t <- table)
cfor (t <- table) cfor (t <- table) cfor (t <- table)
cfor (t <- table)
cfor (t <- table)
Emp = malloc(size* sizeof(char))
cfor (t <- table)
cfor (t <- table)
cfor (t <- table)
cfor (t <- table)
]
```

In the second step, the `Print` operator evaluates the `exec` method of the `Agg` operator with print as a callback.

```c
2 [ // generating the hash map data structure
hashmap = HashMap(Seq(StringField("edname")), Seq(IntField("count")))
]
```

Performing `Agg.exec` generates code to create a hash map and evaluates the `Scan.exec` method with a callback that performs the aggregate operation i.e., extracting the grouping key, updating the `HashMap`, accumulating the aggregate computation, etc.

```c
3 [ val hm = new HashMap(Seq(StringField("edname")), Seq(IntField("count")))
Scann("Emp"),exec(t) =>
val key = t("edname")
hm.update(key, 0) { c += c + 1 }
}
for (t <- hm) println(t) ]
```

This leaves execution of `Scan.exec`, its callback, and the embedded call to `hm.update`. Next, the `Scan`’s for loop is generated, the tuple is instantiated as a specialized `Record` and the `hm.update` with its aggregate function is expanded.
struct Anon2377293 {
    struct Anon2377293 {
        // struct definitions
        int length;
    }
}

// table and arrays i.e., the Record

volatile long x118 = DEFAULT_AGG_HASH_SIZE;
x118 = 0L << 6;
long x158 = 0L;
struct Anon2377293 x179 = x178;
struct Anon2377293 x178 = x176;
bool x159 = TRUE_AGG_HASH_SIZE;
bool x158 = 0L;
long x157 = x156 + x150;
}

// table

// Scan loop, update the hash map with aggregate computation
for (int i = 0; i < table_length; i++) {
    int index = stringHash(used[i]) % size;
    if (key_edsnames[index] == 0) {
        // disregard possibility of hash collisions
        used[index] = i;
        used[index] = 1L;
        key_edsnames[index] = used[index];
        agg_count[index] = 0 + 1;
    } else
        agg_count[index] += agg_count[index] + 1;
}

for (tuple in table)
    println("Hash check: ", tuple(0), ", tuple(1) = ");

In Summary, slightly changing the implementation of Record and HashMap results in generating optimized code.

Final Generated Code from LB2. Figure 14 shows the final C code LB2 generates for the same query, without any simplifications for readability. In contrast to the code shown inline above, the code in Figure 14 implements a proper hash lookup loop to support collisions using open addressing. The first iteration of this loop is peeled to implement a fast path for the common case where the entry is found without collision. In addition, we found that LB2 often achieves better performance when using structs for aggregate entries, as shown in Figure 14, instead of a fully columnarized hash table implementation. These differences are all realized using high-level programming choices, not by low-level compiler transformations. Using proper abstractions, the change is often just a single line. This highlights again the utility of generative programming in rapidly exploring a multitude of design choices, and in building systems that can rapidly adapt to changing requirements.