How to Architect a Query Compiler, Revisited

Ruby Y. Tahboub, Grégory M. Essertel, Tiark Rompf
Purdue University, West Lafayette, Indiana
{rtahboub,gesserte,tiark}@purdue.edu

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

ACM Reference format:
Ruby Y. Tahboub, Grégory M. Essertel, Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of 2018 International Conference on Management of Data, Houston, TX, USA, June 10–15, 2018 (SIGMOD'18),* 16 pages.
https://doi.org/10.1145/3183713.3196893

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.

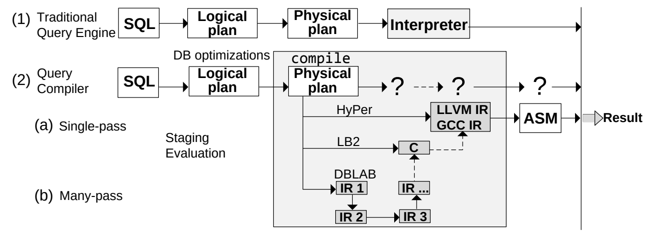


Figure 1: Illustration of (1) query interpreter (2) query compilers (a) single-pass compiler (b) many-pass compiler

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 in 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196893

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).

[1] Of 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.

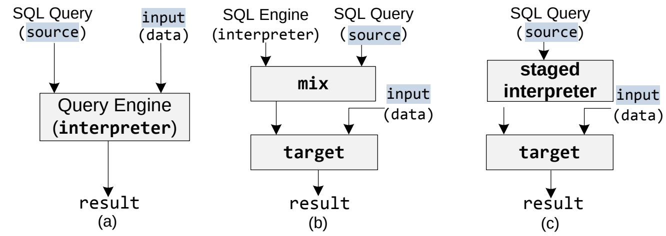


Figure 2: (a) Query interpreter (b) applying the first Futamura projection on a query interpreter (c) the LB2 realization of the first Futamura projection

- We compare the performance of LB2 with HyPer and DBLAB. We show that LB2 is the first system implemented in a high-level language that approaches HyPer performance in a fully TPC-H compliant setting, both sequential and on up to 16 cores. By contrast, LegoBase [26] and DBLAB [44] do not support parallelism and they need to resort to non-TPC-H-compliant indexing and precomputation to surpass HyPer on TPC-H queries. With all such optimizations turned on, LB2 is competitive with DBLAB as well (Section 5).

In summary, we show that query compilation can be simple and elegant, without significantly more implementation effort than an efficient query interpreter. We review related work in Section 6 and offer further discussion and concluding remarks in Sections 7 and 8.

2 FUTAMURA PROJECTIONS

In 1970, at a time when the hierarchical and network models of data [8] were *du jour*, Codd's seminal work on relational data processing appeared in CACM [12]. One year later, in 1971, Yoshihiko Futamura published his paper "Partial Evaluation of Computation Process–An approach to a Compiler-Compiler" in the Transactions of the Institute of Electronics and Communication Engineers of Japan [16]. The fundamental insight of Codd was that data could be profitably represented as high-level relations without explicit reference to a given storage model or traversal strategy. The fundamental insight of Futamura was that compilers are not fundamentally different from interpreters, and that compilation can be profitably understood as *specialization* of an interpreter, without explicit reference to a given hardware platform or code generation strategy.

To understand this idea, we first need to understand *specialization* of programs. In the most basic sense, this means to take a generic function, instantiate it with a given argument, and simplify. For example, consider the generic two-argument power function that computes x^n:

```
def power(x:Int, n:Int): Int =
  if (n == 0) 1 else x * power(x, n - 1)
```

If we know the exponent value, e.g., `n = 4`, we can derive a specialized, *residual*, power function:

```
def power4(x:Int): Int = x * x * x * x
```

This form of specialization is also known as *partial evaluation* [22].

Specializing Interpreters. The key idea of Futamura was to apply specialization to interpreters. Like `power` above, an interpreter is a two-argument function: its arguments are the code of the function to interpret, and the input data this function should be called with. Figure 2a illustrates the case of databases: The query engine evaluates a SQL query (static input) and data (dynamic input) to produce the result. The effect of specializing an interpreter is 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 produces 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 turned out to be largely intractable in practice due to the difficulty of binding-time separation [22]: deciding which expressions in a program to evaluate directly, at specialization time, and which ones to residualize into generated code.

Programmatic Specialization. Even though the magic `mix` component in Figure 2b has turned out to be elusive in practice, all is not lost. We just have to find another way to implement program specialization, perhaps with some help from the programmer. Going back to our example, we can implement a self-specializing power function like this:

```
def power(x:MyInt, n:Int): MyInt =
  if (n == 0) 1 else x * power(x, n - 1)
```

What is different? We changed the type of `x` from `Int` to `MyInt`, and assuming that we are working in a high-level language with operator overloading capabilities, we can implement `MyInt` as a symbolic data type like this:

```
// symbolic integer type
class MyInt(ref: String) {
  def *(y: MyInt) = {
    val id = freshName();
    println(s"int $id = $ref * ${y.ref};");
    new MyInt(id)
} }
```

```
// implicit conversion Int -> MyInt
implicit def constInt(x: Int) =
                    new MyInt(x.toString)
```

When we now call `power` with a symbolic input value:

```
power(new MyInt("in"),4)
```

it will emit the desired specialized computation as as a side effect:

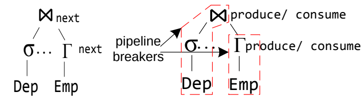
```
int x0 = in * 1;    int x1 = in * x0;
int x2 = in * x1;   int x3 = in * x2; // = in * in * in * in
```

Intermediate steps can be visualized in Appendix B.1. Binding each intermediate result to a fresh variable guarantees a proper sequencing of operations. Based on this core idea of introducing special data types for symbolic or *staged* computation, we suddenly have a handle on making the first Futamura projection immediately practical. As with `power` and `MyInt` above, we just need to implement a query interpreter in such a way that it evaluates a concrete query on symbolic input data. The result, illustrated in Figure 2c, is a

```
// Data schema: Dep(dname: String, rank: Int), Emp(eid: Int, edname: String)
select * from Dep, (select edname, count(*) from Emp group by edname) as T
where rank < 10 and dname = T.edname
```

```
Print(
  HashJoin(
    Select(Scan("Dep"))
        (x => x("rank") < 10),
    Agg(Scan("Emp"))
        (x => x("edname"))(0)
          ((agg,x) => agg + 1)))
```

(a) Query in SQL and query plan



(b) Volcano (c) Data-centric

```
abstract class Op {
  def open(): Unit
  def next(): Record
  def close(): Unit
}

class Select(op: Op)(pred: Record =>
      Boolean) extends Op {
  def open() = op.open
  def next(): Record = {
    var rec = null
    do {
      rec = op.next
    } while (!isNull(rec)
          && !pred(rec))
    rec
  }
  def close() = {...}
}
```

(d) Volcano interface and select

```
abstract class Op {
  def open(): Unit
  def produce(): Unit
  def consume(rec:Record): Unit
  def close(): Unit
}
class Select(op: Op)(pred: Record =>
      Boolean) extends Op {
  var parent = null
  def open() = {
    op.parent = this
    op.open
  }
  def produce(): Unit = op.produce
  def consume(rec: Record) = {
    if (pred(rec))
      parent.consume(rec)
  }
}
```

(e) Data-centric interface and select

Figure 3: Query Evaluation models

practical and directly implemented realization of the first Futamura projection without `mix`. The remainder of this paper will explain the details of this approach and present our query compiler LB2.

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.*

```
// Schema: Dep(dname: String, rank: Int)

// Query plan
Print(
  Select(
    Scan("Dep")))(x => x.rank < 10)
```

(a)

```
// Specialized Volvano evaluation
var nextRec = 0 // scan state
val size = data.length
while (true) {  // print loop
  val rec = {
    var recSel = null
    do {        // select loop
      recSel = if (nextRec < size)
        data(nextRec++) else null
    } while (!(selRec == null) &&
     !(recSel.rank < 10))
    recSel
  }
  if (rec == null) break
  println(rec.dname + "," + rec.rank)
}
```

(b)

```
// Specialized data-centric evaluation
for (rec <- data) {
  if (rec.rank < 10)
    println(rec.dname+","+rec.rank)
}
```

(c)

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

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = 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(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

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:

```
class Agg(op: Op)(grp: GrpFun)(init: String)(agg: AggFun) extends Op { // operator template
  def exec(cb: String => String) = s"""
    val hm = new HashMap()
    ${op.exec { tuple =>
      s"""val key = ${grp(tuple)}
      hm.update(key, $init) { curr => ${agg("curr", "tuple")} }
    }"""}
    for (tuple <- hm) { ${cb("tuple")} }"""
}
```

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:

```
class Record(fields: Seq[(String, String)]) {
  val name = freshName()
  println(s"""val $name = new Record(${fields map ... })""")
  def apply(field: String) = s"""$name.$field"""
  def update(field: String, v: String) = println(s"""$name.$field = $v;""")
}
class HashMap() {
  val name = freshName()
  println(s"""val $name = new HashMap()""")
  def apply(key: String) = s"""$name($key)"""
  def update(key: String, v: String)(up: String) =
    println(s"""$name($key) = $up($name.getOrElse($key,$v))""")
}
```

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).

```
select edname, count(*)
from Emp
group by edname
```

```
 Π
 |
 Γ
 |
Emp
```

```
Print(
  Agg(Scan("Emp"))
    (x => x("edname"))(0)
      ((agg,x) => agg + 1))
```

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 recived 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 *scalarized* form, mapped to local variables in generated code (class `ColumnRec`).

```
abstract class Field { def 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 Value                               // models an attribute's value
case class IntValue(value: Rep[Int]) extends Value { ... }  // operations elided
case class StringValue(value: Rep[String], length: Rep[Int]) extends Value { ... }
```

```
type Pred   = Record => Rep[Boolean]
type KeyFun = Record => Record
type OrdFun = (Record,Record) => Rep[Int]
type AggFun = (Record,Record) => Record
type GrpFun = Record => Record

class Scan(table: Buffer) extends Op {
  def exec(cb: Record => Unit) = {
    for (tuple <- table)
      cb(tuple)
} }
class HashJoin(left: Op, right: Op)(lkey: KeyFun)
(rkey: KeyFun) extends Op { /* code in Figure 5 */ }
```

```
class Select(op: Op)(pred: Pred) extends Op {
  def exec(cb: Record => Unit) = {
    op.exec { tuple =>
      if (pred(tuple)) cb(tuple)
} } }
class Sort(op: Op)(ordFun: OrdFun) extends Op {
  def exec(cb: Record => Unit) = {
    val res = new FlatBuffer()
    op.exec { tuple => res += tuple }
    res.sort(ordFun)
    for (tuple <- res)
      cb(tuple)
} }
```

```
class Agg(op: Op)(grp: GrpFun)
     (init: Record)(agg: AggFun) extends Op {
  def exec(cb: Record => Unit) = {
    val hm = new HashMap()
    op.exec { tuple =>
      val key = grp(tuple)
      hm.update(key, init) {
        curr => agg(curr, tuple)
    } }
    for (tuple <- hm)
      cb(tuple)
} }
```

Figure 6: LB2 Query Evaluator (data-centric with callbacks)

```
abstract class Record { def schema: Seq[Field]; def apply(name: String): Value }
case class NativeRec(pt: Rep[Pointer], schema: Seq[Field]) extends Record {
  def apply(name: String) = getField(schema,name).readValue(pt, getFieldOffset(schema,name))
}
case class ColumnRec(fields: Seq[Value], schema: Seq[Field]) extends Record {
  def apply(name: String) = fields(getFieldIndex(schema,name))
}
```

Likewise, LB2 abstracts over `Record` storage through an abstract class `Buffer`, with implementation classes for row-oriented (`FlatBuffer`) and column-oriented storage (`ColumnarBuffer`):

```
abstract class Buffer(schema: Seq[Field]) {
  def apply(x: Rep[Int]): Record
  def update(x: Rep[Int], rec: Record): Unit
}
case class FlatBuffer(schema: Seq[Field], size: Long) extends Buffer(schema) ...
case class ColumnarBuffer(schema: Seq[Field], size: Long) extends Buffer(schema) {
  val cols = schema map { fld => Column(fld,size) }
  def apply(x: Rep[Int]) = ColumnRec(cols map { c => c(x) }, schema)
  def update(x: Rep[Int], y: Record): Unit =
    cols foreach { c => c.update(x, y(c.field.name)) }
}
case class Column(field: Field, size: Long) {
  val buf: Rep[Pointer]  = malloc(size * field.size)
  def apply(x: Rep[Int]) = field.readValue(buf, x)
  def update(x: Rep[Int], y: Value): Unit = field.writeValue(buf, x, y)
}
```

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 important to note that there is never any code like `new Record(...)` or `new Value(...)` being generated. Hence, `Value`, `Record`, `Buffer`, etc., objects are generation-time-only abstractions that are completely dissolved as part of the symbolic evaluation. The generated code consists only of the operations on the `Rep[T]` values hidden inside the `Value` and `Buffer` subclasses, i.e., only raw `malloc`s and pointer reads/writes (refer to Appendix B.2 for details).

A pipeline of operators accesses records uniformly in either row or column format. A pipeline breaker materializes the intermediate `Records` inside a buffer or higher-level data structure, at which point a format conversion may occur.

4.2 Data Structure Abstractions

Query engines use map data structures to implement aggregate and join operations. For aggregations, a `HashMap` accumulates the aggregate result for a grouping key and for hash joins, a `HashMultiMap` collects all records for a given key. As discussed earlier, implementations from a generic library (e.g., `Glib`) tend to be inefficient due to expensive function calls, resizing, etc. Moreover, we want to make different low-level implementation choices for different parts of the engine (e.g., open addressing vs. linked hash buckets). Based on the `Buffer` abstractions (Section 4.1), we can build a considerable variety of fast hash table implementations with little effort.

The code example below illustrates the one we use for aggregates (see Figure 6). Class `HashMap` defines the interface, and subclass `LB2HashMap` provides an implementation based on open addressing on top of `ColumnarBuffer`. Method `update` locates the key and updates the aggregate value. Method `foreach`, which is invoked for Scala expressions of the form `for (rec <- hm)`, traverses all stored values. The hash map is fully specialized for key and value types.

```
abstract class HashMap(kSche: Seq[Field], vSche: Seq[Field]) {
  def update(key: Record, init: Record)(up: Record => Record): Unit
  def foreach(f: Record => Unit): Unit
}
class LB2HashMap(kSche: Seq[Field], vSche: Seq[Field]) extends HashMap {
 val size = defaultSize
 val agg = new ColumnarBuffer(vSche, size)
 val keys = new ColumnarBuffer(kSche, size)
 val used = new Array[Int](size)
 var next = 0
 def update(k: Record, init: Record)(up: Record => Record) = {
   val idx = defaultHash(k) % size
   if (isEmpty(keys(idx))) {
     used(next) = idx; next += 1
     keys(idx) = k; agg(idx) = up(init)
   } else agg(idx) = up(agg(idx))
 }
 def foreach(f: Record => Unit) = {
   for (idx <- 0 until next) {
     val j = used(idx)
     f(merge(keys(j), agg(j)))
} } }
```

It is important to note that this code is the same as one would write in a library implementation. However, as with `Buffers` and `Records`, the `HashMap` abstraction is completely dissolved at code generation time, leaving only low-level array and index manipulations.

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 speed up 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 `getIndex` method, which enables `IndexJoin.exec` to find tuples that match the join key:

```
// Index join operator that uses index created on the left table
class IndexJoin(left: Op, right: Op)(lkey: String)(rkey: KeyFun) extends Op {
  def exec(cb: Record => Unit) = {
    val index: Index = left.getIndex(lkey) // obtain index for left table
    right.exec { rTuple => // use index to find matching tuples
      for (lTuple <- index(rkey(rTuple))) cb(merge(lTuple, rTuple))
} } }
```

LB2 realizes sparse 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.

```
abstract class IndexEntryView {
  def foreach(f: Record => Unit): Unit
  def exists(f: Pred): Rep[Boolean]
}
abstract class Index(schema: Seq[Field]) {
  def apply(k: Record): IndexEntryView
}
```

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 data, or as primary partioned and/or replicated data format, e.g., when there are multiple foreign keys.

Date Indexes. LB2 represents dates as numeric values to speed up filter and range operations. If metadata about date ranges is available, it will enable further shortcuts. LB2 breaks down dates into year and month and uses existing abstractions to index dates based on year or month. Adding a date index is similar to creating an index on a primitive type. Hence, we elide further details.

String Dictionaries. Another form of indexing is compressed columns and dictionary encodings. LB2 implements string dictionaries to optimize string operations, e.g., `startsWith`. Individual columns can be marked as dictionary compressed in the database schema. Building on `StringValue` and `ColumnarBuffer` discussed earlier in Section 4.1, LB2 defines an alternative string representation class `DicValue` and a class `StringDict` that stores and provides access to compressed string values.

```
class DicField(name: String) extends Field {
  def dict: StringDictionary = ...
}
class DicValue(idx: Rep[Long]) extends Value {
  def startsWith(p: DicValue) = p.idx <= idx
  ...
}
class StringDict(attr: String, size: Long) {
  val store = Column(StringField(attr),size)
  def convert(string: StringValue): DicValue
  def get(idx: DicValue): StringValue
}
```

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 compressed form is only valid if they share the same dictionary. One solution is to use a single global dictionary for all string attributes, however it is not easy to maintain. Another solution is to group the columns that are the most likely going to be used together within one dictionary. 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.

4.4 Code Layout and Code Motion

Hoisting memory allocation and other expensive operations from frequently executed paths to less frequent paths can speed up evaluation dramatically, especially in hash join and aggregate queries where memory may be pre-allocated in advance.

```
def exec(cb: Record => Unit) = {
  val hm = new HashMap()

  op.exec { tuple =>
    hm.update(grp(tuple), ...)(...)
  }
  for (tuple <- hm) cb(tuple)
}
```

(a1) Aggregate skeleton

```
def exec = {
  val hm = new HashMap(); val dataLoop = op.exec
  (cb: Record => Unit) => {
    dataLoop { tuple =>
      hm.update(grp(tuple), ...)(...)
    }
    for (tuple <- hm) cb(tuple)
} }
```

(a2) Optimized aggregate skeleton

```
// execute aggregate

time { Print.exec(t => ()) }
```

(b1) Aggregate client program

```
// execute aggregate
val query = Print.exec
time { query(t => ()) }
```

(b2) Optimized aggregate client program

```
struct timeval start, end;
gettimeofday(&start, NULL);
// data struct allocation
int* agg = malloc(...); ...
// processing
for (int i = 0; i < N; i++) {
  // ... compute aggregate ...
}
for (int i = 0; i < next; i++) {
  // ... print records ...
}
gettimeofday(&end, NULL);
```

(c1) Generated code

```
// data struct allocation
int* agg = malloc(...); ...
struct timeval start, end;
gettimeofday(&start, NULL);
// processing
for (int i = 0; i < N; i++) {
  // ... compute aggregate ...
}
for (int i = 0; i < next; i++) {
  // ... print records ...
}
gettimeofday(&end, NULL);
```

(c2) Generated code

Figure 7: From a query plan to optimized C code including data structures specialization and code motion

Let us recall the design principle from Section 3, that the shape of the generated code follows how the interpreter is written. Based on this insight, we may reorder evaluation actions in a certain way that moves expensive operations off the hot path of query execution. In particular, we can extend LB2's callback interface to enable hoisting data structure allocation. In Figure 7, we demonstrate how a small change to the `exec` method signature enables data structures hoisting. Method `exec` in Figure 7-a1 takes a single argument, allocates data structures and invokes `parent.exec`. In Figure 7-a2, `exec` is re-implemented as zero-parameter method that separates the data structure allocation and query execution by performing the allocation first and only then *returning* a function that executes the operator's main data path. The rest of the example in Figure 7-b1 and b2 shows how client programs can inject code, e.g., timing, inbetween memory allocation and the main evaluation loop. Figure 7-c1 and c2 show the respective generated code.

4.5 Parallelism

Query engines realize parallelism either explicitly by implementing special *split* and *merge* operators [31], or internally by modifying the operator's internal logic to orchestrate parallel execution. LB2 uses the latter, and generates code for OpenMP [1].

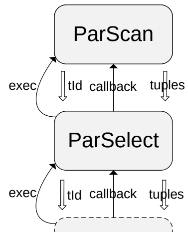
Interestingly, the same pattern we used in Section 4.4 to achieve code motion can be used to structure query engine code for parallel execution. The callback signature for `exec` we defined earlier works well in a single-threaded environment, but multi-threaded environments require synchronization and thread-local variables. Thus, LB2 defines a new class `ParOP` with a modified `exec` method that adds another callback level.

For state-less operators such as `Select`, the parallel implementation is very similar to the single threaded one; it is possible to use a wrapper to transform a single threaded pipeline into a parallel one. Assuming we have a parallel scan operator `ParScan`, we can perform a parallel selection like this:

```
val parSelect = parallelPipeline(op => Select(op)(t => t("rank") < 10))
parSelect(ParScan("Dep"))
```

The definition of `ParOp` and `parallelPipeline` is as follows:

```
class ParOp {
  type ValueCallback = Record => Unit
  type DataLoop = ValueCallback => Unit
  type ThreadCallback = Rep[Int] => DataLoop => Unit
  def exec: ThreadCallback => Unit
}
def parallelPipeline(seq: Op => Op) =
  (parent: ParOp) => new ParOp {
    def exec = {
      val opExec = parent.exec
      (tCb: ThreadCallback) => opExec { tId => dataloop =>
        tCb(tId)((cb: ValueCallback) =>
          seq(new Op { def exec = dataloop }).exec(cb) })
} } }
```



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(nT: Long, kSche: Seq[Field], vSche: Seq[Field]) {
  def apply(tId: Rep[Int]): HashMap
  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(nbThread, grp.schema, agg.schema)
    (tCb: ThreadCallback) => {
      opExec { (tId: Rep[Int]) => (dataLoop: DataLoop) => // parallel section starts
        val lHm = hm(tId): HashMap
        dataloop { tuple => // computes partial result for this thread
          val k = grp(tuple)
          lHm.update(k, init) { c => agg(c, tuple) }
      } }                                               // parallel section ends
      hm.merge(init, agg) // merge the results accross threads
      parallelRegion { tId => tCb(tId)(hm.partition(tId)) } // restart a pipeline in parallel
} } }
```

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 modifed 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 re-written 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 decision 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 hoist 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

SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Postgres	241581	6665	33835	7980	30158	23485	32577	29953	64429	33237	7113
DBLAB	809	219	6575	7977	3049	232	3598	5494	32313	4175	104
LB2 (DBLAB plan)	476	140	491	573	576	223	914	1941	4540	1230	52
HyPer	613	61	1193	872	991	233	982	637	2029	1021	138
LB2 (HyPer plan)	476	141	494	573	649	222	706	862	2806	1378	51

SF10	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Postgres	38040	31425	22154	23183	13450	155541	91241	29548	65681	299691	11766
DBLAB	3847	(1079)	526	933	2961	3795	4771	6353	763	13851	1328
LB2 (DBLAB plan)	691	4012	377	373	595	2001	1294	2623	569	1887	235
HyPer	504	3718	342	261	1274	580	3764	2003	439	1658	190
LB2 (HyPer plan)	691	4012	246	189	593	1994	1289	2965	833	1793	234

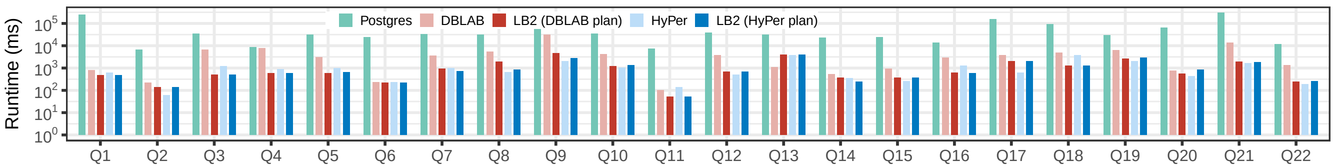


Figure 8: The absolute runtime in milliseconds (ms) for DBLAB, LB2 (with DBLAB's plans), HyPer, LB2 (with HyPer's join ordering plans) in TPC-H SF10. Only TPC-H compliant optimizations are used.

callback interface seamlessly hoist data structures allocation outside the operator code, with small changes to the operator interface.

Parallelism. LB2 implements parallelism as shown in Section 4.5. DBLAB does not implement parallelism, although the paper [44] discusses some ideas in this direction.

In summary, we believe that there are certain drawbacks to multi-pass query compilers. Most importantly, optimizations require a number of analysis and rewrite passes, i.e., database engineers have to become real compiler experts. The hard part about writing compilers is not to get a transformation working on a few examples, but to ensure correctness for *all* possible corner cases, and for interactions with other parts of the system, which may be changed independently. In comparison, LB2's single-pass approach facilitates query compilation using techniques that are no more difficult than writing a query interpreter in a high-level language.

5 EVALUATION

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 plan 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 `hm-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 configuration 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 E7-88904 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 [2], 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][3].

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

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

[3]The 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].

SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
DBLAB-idx	827	144	2341	631	380	236	123	332	900	765	100	1167	(1379)	272	919	4648	266	4680	247	312	1408	208
LB2-idx	521	87	685	316	207	223	127	81	1021	663	15	489	3895	272	440	1157	144	183	69	84	347	104
DBLAB-idx-date	928	101	422	217	207	117	78	292	846	703	85	1242	(1337)	289	40	5355	218	200	311	414	626	173
LB2-idx-date	485	85	152	162	212	80	108	81	1024	455	17	294	3895	26	54	1153	138	204	69	83	406	104
DBLAB-idx-date-str	497	97	138	222	225	82	79	23	510	603	130	362	(4507)	16	41	555	13	198	15	401	494	176
LB2-idx-date-str	487	74	166	185	170	79	85	35	641	408	16	184	3904	10	56	774	7	184	11	85	384	104

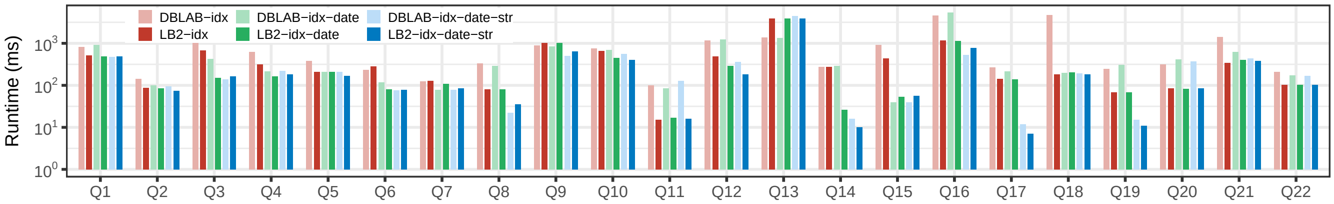


Figure 9: The absolute runtime in milliseconds (ms) after enabling non-TPC-H-compliant indexing, date indexing and string dictionary in SF10 using DBLAB plans

join and anti join queries Q4, Q16, Q21 and Q22. In Q13, DBLAB replaces the outer join operator with a hard-coded imperative array computation that has no counterpart in the query plan language. Hence, a direct comparison for this query is misleading, and we do not attempt to recreate an equivalent "plan" in LB2.

The performance gap between LB2 and DBLAB can be attributed, in part, to a number of implementation details. First, LB2 implements a generation-time string abstraction (explained in Section 4.2) that optimizes commonly used string operations while DBLAB primarily relies on C-strings and only optimizes one instance (the `startsWith` operation). Second, the systems make different decisions related to hash join implementations, e.g., the hash function and number of keys used while creating a hash table. LB2's hash table is always custom-generated while DBLAB sometimes relies on GLib C data structures and also uses function calls for some string operations as opposed to inlining and specialization in LB2. Furthermore, there are different trade-offs related to handling composite keys, i.e., either creating a very selective hash table that combines more than one key or using only a single key and relying on the join condition to filter unqualified tuples (essentially a short linear search). We observe that DBLAB opts for smaller hash tables at the expense of more extensive search, while LB2 chooses more precise hashing. Finally, there are differences in optimizing data layout, e.g., pertaining to row-oriented vs column-oriented layout for internal data structures (Section 4).

Comparing the performance of LB2 and HyPer, we observe that LB2 is faster by at least 2×-3× in Q3, Q11, Q16 and Q18. Also, LB2 is 25%-50% faster than HyPer in Q1, Q4, Q5, Q7, Q14 and Q15. On the other hand, HyPer is faster than LB2 by 2×-3× in Q2 and Q17. This performance gap is, in part, attributed to (1) HyPer's use of specialized operators like GroupJoin and (2) employing indexes on primary keys as seen in Q2, Q8-Q10, etc. Finally, while both LB2 and HyPer generates native code (LB2 generates C and HyPer generates LLVM) differences in implementation may result in faster generated code. One example is floating point numbers. HyPer uses proper decimal precision numbers, whereas LB2 and DBLAB use double precision floating point values.

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 key. While both systems follow a similar compromise in allocating larger space to optimize access time, LB2's column-oriented layout contributes to lowering index creation and access cost. This observation is the main reason of the better performance for LB2 over DBLAB. On a query by query analysis for the absolute runtime, LB2 outperforms DBLAB in join query Q3 by 3× and by 15%, 80% in Q10 and Q5 respectively. Similarly, LB2 is 2×-4× faster in semi join and anti join queries Q4, Q22, Q16, Q21. On the other hand, DBLAB is faster than LB2 in Q7 and Q9 by 3% and 13% respectively. As discussed in Section 5.1 the performance gap is attributed to a variety of implementation details that are different between the two systems.

Finally, we can notice that using index inference may result in access paths that are slower than hash joins, e.g., Q16. Indexes are built on leaf nodes whereas hash tables are built on interior nodes with smaller intermediate result. Therefore, a query optimizer with a cost model would avoid slower access paths and using an index everytime it is availble may not be the optimal solution for the query. For Q13 DBLAB is much faster than LB2 due to their use of imperative computation outside of query plans (as noted above). Curiously, enabling index optimizations causes a slowdown in DBLAB for Q13.

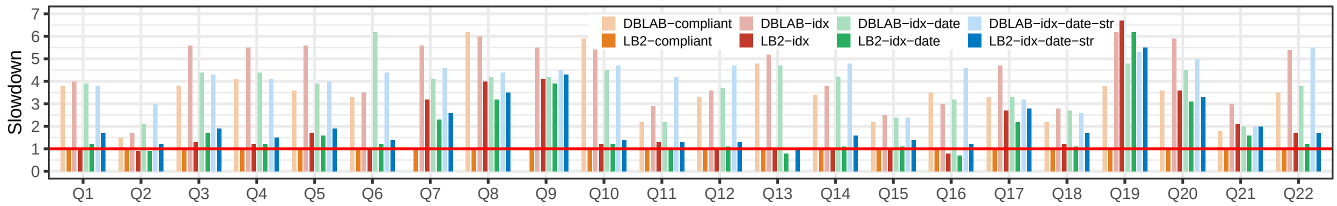


Figure 10: Overhead in loading time introduced by index creation, date indexing and string dictionary on DBLAB and LB2 in SF10 using DBLAB plans (slowdown relative to LB2-compliant)

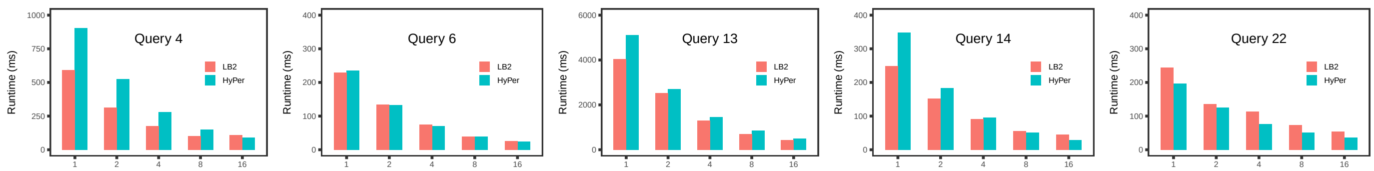


Figure 11: The absolute runtime in milliseconds (ms) for parallel scaling up LB2 and HyPer in SF10 on 2, 4, 8 and 16 cores

Date Indexes. The date indexing optimization is used when a table is filtered on a date attribute. The table is partitioned by year and month on the given attribute and the index is scanned only on the dates that satisfy the predicate. This optimization is always beneficial in both systems.

String Dictionaries. DBLAB creates string dictionaries to speed up commonly used string operations: `equality`, `startsWith`, `endsWith` and `like`. In LB2, a string dictionary is used only for the first three operations. We decided that tokenizing strings in order to optimize the `like` operation (as in DBLAB) is difficult in the general situation. A simple example is `like %green%`. This should match the word `greenway` but tokenizing over spaces, punctuation, etc. would not be correct. The line `DBLAB/LB2-idx-date-str` in the Figure 9 shows runtime when using the string dictionary optimization in DBLAB and LB2. Queries 3, 8, 12, 17, 19 are only using equality operations. LB2 is 35%-95% faster in Q12, Q17, Q19 whereas DBLAB is 20%, 50% faster in Q3 and Q8 respectively. Moreover, Q2 and Q14 uses `startsWith` and `endsWith`. LB2 is 30% and 60% faster in these queries. Finally, Q9, Q13 and Q16 use `like` that LB2 does not optimize. However, DBLAB does not optimize it for Q13 either. When generating the C code for Q13 in this experiment, the executable raised a segfault that we manually fixed.

5.3 Parallelism

In this experiment, we compare the scalability of LB2 with HyPer. DBLAB does not support parallelism. We pick five queries that represent aggregates and join variants. Figure 11 gives the absolute runtime for scaling up LB2 and HyPer for Q4, Q6, Q13, Q14 and Q22 in SF10. At first glance, the speedup of LB2 and HyPer increases with number of cores, by an average 4×-5× in Q22 and by 5×-11× in Q4, Q6, Q13 and Q14.

At a closer look, LB2 outperforms HyPer in semi join Q4 by 50% with 2 to 8 cores. In outer join Q13, LB2 is 10%-20% faster than HyPer up to 16 cores. On the other hand, the performance of LB2 and HyPer is comparable in aggregate query Q6. Finally, HyPer outperforms LB2 in anti join Q22 by 10%-50% with 2 to 16 cores. In summary, the parallel scaling in LB2 and HyPer lie within a close range. However, 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. The

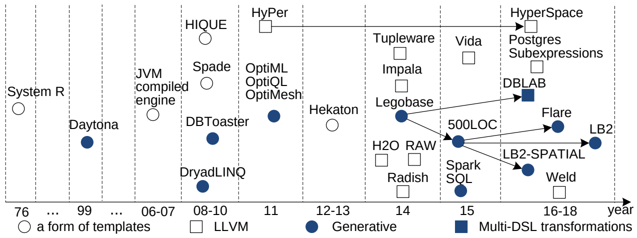


Figure 12: The evolution of query compilation

DBLAB system [44] compiles queries by applying multi-pass DSL transformations. Commercial examples of query compilation are Hekaton [14], DryadLINQ [20], Impala [27] and Spade[17]. Cluster computing examples are SparkSQL [6] and Tupleware [13]. Another line of works generate query code in high-level languages using multi-stage generative programming with LMS [40]. Closest to LB2 are Legobase [26], the "SQL to C in 500 lines" compiler [38], Flare [15], a native compiler back-end for SparkSQL, LB2-Spatial [48], Jet [3], OptiML [47] and other Delite's DSLs [9, 46] e.g., OptiQL and OptiMesh.

Code generation. *Domain-specific Languages* (DSLs) are high-level abstractions that allow expressing programs using domain terms, e.g., SQL. *Generative programming* [39] enables specialization as a programming pattern where developers implement a program *generator* and use high-level programming abstractions to achieve the desired forms of specialization. Lightweight Modular Staging (LMS) [40] is a library-based approach that implements generative programming abstractions using operator overloading and other features in regular general-purpose languages.

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 under-appreciated 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.

REFERENCES

[1] [n. d.]. OpenMP. http://openmp.org/. ([n. d.]). last accessed: 11-02-2017.

[2] [n. d.]. Postgresql. https://www.postgresql.org. ([n. d.]). last accessed: 11-02-2017.

[3] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. 2012. Jet: An Embedded DSL for High Performance Big Data Processing *(International Workshop on End-to-end Management of Big Data (BigData 2012))*.

[4] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *VLDB* 2, 2 (2009), 1566–1569.

[5] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *SIGMOD*. 1103–1114.

[6] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. ACM, 1383–1394.

[7] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: relational approach to database management. *TODS* 1, 2 (1976), 97–137.

[8] Charles W Bachman. 1973. The programmer as navigator. *Commun. ACM* 16, 11 (1973), 653–658.

[9] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. 2016. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)*. ACM, New York, NY, USA, 194–205.

[10] Dennis Butterstein and Torsten Grust. 2016. Precision Performance Surgery for PostgreSQL LLVM–based Expression Compilation, Just in Time. *PVLDB* (2016).

[11] Tatsuhiro Chiba and Tamiya Onodera. 2015. *Workload Characterization and Optimization of TPC-H Queries on Apache Spark*. Technical Report RT0968.

[12] Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

[13] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An architecture for compiling udf-centric workflows. *PVLDB* 8, 12 (2015), 1466–1477.

[14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*. ACM, 1243–1254.

[15] Grégory M Essertel, Ruby Y Tahboub, James M Decker, Kevin J Brown, Kunle Olukotun, and Tiark Rompf. 2017. Flare: Native compilation for heterogeneous workloads in Apache Spark. *arXiv preprint arXiv:1703.08219* (2017).

[16] Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process — An approach to a Compiler-Compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan* 54-C, 8 (1971), 721–728.

[17] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. 2008. SPADE: the system s declarative stream processing engine. In *SIGMOD*. ACM, 1123–1134.

[18] Goetz Graefe. 1994. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on* 6, 1 (1994), 120–135.

[19] Rick Greer. 1999. Daytona and the fourth-generation language Cymbal. In *ACM SIGMOD Record*, Vol. 28. ACM, 525–526.

[20] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *SIGOPS/EuroSys (EuroSys)*. 59–72.

[21] N.D. Jones, C.K. Gomard, and P. Sestoft. 1993. *Partial evaluation and automatic program generation.* Peter Sestoft.

[22] Neil D Jones. 1996. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)* 28, 3 (1996), 480–503.

[23] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB* 9, 12 (2016), 972–983.

[24] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. 2015. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*.

[25] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *PVLDB* 7, 12 (2014), 1119–1130.

[26] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment* 7, 10 (2014), 853–864.

[27] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael and Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*.

[28] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. IEEE, 613–624.

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. 75–88.

[30] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[31] Manish Mehta and David J DeWitt. 1995. Managing intra-operator parallelism in parallel database systems. In *SIGMOD Conference*. 743–754.

[32] Brandon Myers, Daniel Halperin, Jacob Nelson, Mark Oskin, Luis Ceze, and Bill Howe. 2014. *Radish: Compiling Efficient Query Plans for Distributed Shared Memory*. Technical Report.

[33] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[34] Martin Odersky and Tiark Rompf. 2014. Unifying functional and object-oriented programming with Scala. *Commun. ACM* 57, 4 (2014), 76–86.

[35] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. 2016. High-Performance Geospatial Analytics in HyPerSpace. In *SIGMOD*. 2145–2148.

[36] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a vector algebra for portable database performance on modern hardware. *VLDB* 9, 14 (2016), 1707–1718.

[37] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. 2006. Compiled query execution engine using JVM. In *ICDE*. IEEE, 23–23.

[38] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *ICFP*. ACM, 2–9.

[39] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *SNAPL (LIPIcs)*, Vol. 32. 238–261.

[40] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. 127–136.

[41] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (2012), 121–130.

[42] Amir Shaikhha, Ioannis Klonatos, Lionel Emile Vincent Parreaux, Lewis Brown, Mohammad Dashti Rahmat Abadi, and Christoph Koch. 2016. DBLAB SIGMOD 2016 reproducibility. https://dl.acm.org/citation.cfm?id=2915244. (2016).

[43] Amir Shaikhha, Ioannis Klonatos, Lionel Emile Vincent Parreaux, Lewis Brown, Mohammad Dashti Rahmat Abadi, and Christoph Koch. 2016. DBLAB SIGMOD 2016 reproducibility git repository. https://github.com/rtahboub/dblab/commit/e5d3fe2d5e8705fc827c40b07b752291f967a5a6. (2016).

[44] Amir Shaikhha, Ioannis Klonatos, Lionel Emile Vincent Parreaux, Lewis Brown, Mohammad Dashti Rahmat Abadi, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD*.

[45] Eugene Sharygin, Ruben Buchatskiy, Roman Zhuykov, and Arseny Sher. 2017. Runtime Specialization of PostgreSQL Query Executor. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 375–386.

[46] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM TECS* 13, 4s (2014), 134.

[47] Arvind K. Sujeeth, HyoukJoong. Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*.

[48] Ruby Y Tahboub and Tiark Rompf. 2016. On supporting compilation in spatial query engines:(Vision paper). In *ACM SIGSPATIAL*. ACM, 9.

A ADDITIONAL EXPERIMENTS

The experiments in Section 5 compared the performance of LB2 with HyPer and DBLAB using execution time as a metric. In this section, we evaluate the overhead of code generation and compilation times for LB2 and DBLAB. Furthermore, we conduct a productivity evaluation study that estimates the effort of developing each system.

A.1 Code generation and Compilation

In this experiment, we analyze the overhead of code generation and compilation using GCC in LB2 and DBLAB with the compliant and optimal configurations. The results are illustrated in Figure 13. The y-axis value shows both code generation and GCC compilation for each configuration.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
LB2-compliant	243	650	381	221	444	59	519	663	447	427	302	356	197	147	322	286	171	366	228	387	398	211
LB2-compliant GCC	175	558	421	336	664	180	436	507	498	439	320	283	203	195	228	292	214	368	349	367	449	229
LB2-opt	367	736	420	274	566	139	537	693	468	358	379	261	226	248	420	309	435	502	526	512	686	276
LB2-opt GCC	245	578	289	248	385	180	421	462	408	338	302	259	214	222	340	274	237	332	402	365	467	231
DBLAB-compliant	178	670	514	187	701	90	620	1199	745	458	266	258	164	154	124	485	182	251	198	467	410	271
DBLAB-compliant GCC	260	401	488	363	699	228	605	785	772	434	300	391	254	311	292	497	376	545	598	606	486	273
DBLAB-opt	605	2040	856	658	1783	363	1409	2671	868	700	460	632	432	417	333	772	313	425	473	492	497	403
DBLAB-opt GCC	237	324	382	333	508	207	513	551	532	392	261	321	267	294	266	348	303	342	390	424	390	316

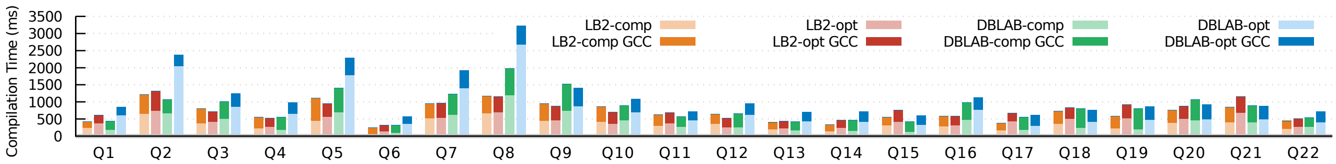


Figure 13: Code generation and compilation for LB2 and DBLAB

Code generation time increases with the number of operators and subqueries, e.g., in Q2, Q5, Q8 and Q21. Compilation times are constant for any dataset size, and can often be amortized if queries are precompiled and used multiple times.

At a closer look, LB2 is faster than DBLAB in TPC-H compliant mode for code generation by an average of 3%-80%. On the other hand, LB2 is slower for Q1, Q4, Q11-Q13, Q15, and Q18-Q19 by an average of 15%-2.6×. For GCC compilation, LB2 is between 40% slower and 75% faster than DBLAB. In the optimal mode, LB2 is slower for Q15 and Q17-Q21 by up to 40% and faster for the remaining queries by up to 3.8×. For GCC compilation, LB2 is between 30% slower and 35% faster than DBLAB.

A.2 Productivity Evaluation

Table 1 summarizes the line of code (LOC) that have been added to the compliant LB2 compiler for each optimization. The original core engine of HyPer is reported to consist of around 11k lines of code [33] that uses low-level LLVM APIs to produce code in LLVM's intermediate language.

DBLAB is based on a specifically developed compiler framework called SC [44], with about 30k lines of code. Most or all of the facilities provides by this framework appear necessary. While the DBLAB developers claim that SC can serve as a *generic* compiler framework, there is no evidence that it is used anywhere else. By contrast, LB2 uses the LMS compiler framework [40], which has been used by a variety of groups in academia and industry for applications ranging from generating JavaScript to C, CUDA, and VHDL. LMS is only 7k lines in total, and of that, LB2 uses only a small fraction. While LMS does provide sophisticated support for multiple intermediate languages and transformation passes, LB2 does not use any of these facilities. Removing this functionality from LMS leads to a core framework of only 2k lines that is enough to support LB2.

	LB2	DBLAB
Base	1800	3700
Index data structures	200	505
Compliant Indexing compilation	80	318
Non-compliant String Dictionary	150	456
Non-compliant Date Indexing	50	400
Memory Allocation Hoisting	30	186
Other	600	1000
Total modified	1100	2800

Table 1: Lines of code needed to add each optimization

B VISUALIZING QUERY CODE GENERATION

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., DBLAB) *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:

```
def power(x:Int, n:Int): Int =
  if (n == 0) 1 else x * power(x, n - 1)
// symbolic integer type
class MyInt(ref: String) {
  def *(y: MyInt) = {
    val id = freshName();
    println(s"int $id = $ref * ${y.ref};");
    new MyInt(id)
} }
// implicit conversion Int -> MyInt
implicit def 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("in"), 4) ⟧
2 ⟦ if (4 == 0) 1 else new MyInt("in") * power(new MyInt("in"), 3)) ⟧
3 ⟦ new MyInt("in") * power(new MyInt("in"), 3) ⟧
4 ⟦ new MyInt("in") * (
    if (3 == 0) 1 else new MyInt("in") * power(new MyInt("in"), 2)) ⟧
5 ⟦ new MyInt("in") *  (new MyInt("in") * power(new MyInt("in"), 2)) ⟧
6 ⟦ new MyInt("in") *  (new MyInt("in") * (
    if (2 == 0) 1 else new MyInt("in") * power(new MyInt("in"), 1)) ⟧
7 ⟦ new MyInt("in") *  (new MyInt("in") * (new MyInt("in") * power(new MyInt("in"), 1))) ⟧
8 ⟦ new MyInt("in") *  (new MyInt("in") * (new MyInt("in") * (
    if (1 == 0) 1 else new MyInt("in") * power(new MyInt("in"), 0)))) ⟧
```

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.

```
9 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * (new MyInt("in") *
    power(new MyInt("in"), 0)))) ⟧
10 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * (new MyInt("in") *
     if (0 == 0) 1 else power(new MyInt("in"), -1)))) ⟧
```

The base case of `power` returns the value 1. Next, the multiplication operator in `MyInt("in") * 1` emits the first line `int x0 = in * 1`, as side effect, and returns `MyInt("x0")`.

```
11 ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * (new MyInt("in") * 1))) ⟧
12 int x0 = in * 1
   ⟦ new MyInt("in") * (new MyInt("in") * (new MyInt("in") * new MyInt("x0"))) ⟧
```

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

```
13 int x0 = in * 1
   int x1 = in * x0
   ⟦ new MyInt("in") * (new MyInt("in") * new MyInt("x1")) ⟧
14 int x0 = in * 1
   int x1 = in * x0
   int x2 = in * x1
   ⟦ new MyInt("in") * new MyInt("x2") ⟧
15 int x0 = in * 1
   int x1 = in * x0
   int x2 = in * x1
   int x3 = in * x2
   ⟦ new MyInt("x3") ⟧
```

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:

```
select ename count(*)        Π          Print(
from Emp                     |            Agg(Scan("Emp"))
group by ename               Γ              (x => x("edname"))(0)
                             |                ((agg,x) => agg + 1))
                            Emp
```

We first consider the basic programmatic specialization approach that generates `Scala`. After that, we walk through the optimized programmatic specialization approach that generates native `C` code, complete with specialized data structure implementations.

Programmatic Specialization (Scala). The first instance uses the simplified `Record` and `HashMap` implementations discussed in the beginning of Section 4 to generate code in Scala. The first expansion step specializes the engine with static input (e.g., attribute names, aggregate parameters, etc.) and evaluates `Print.exec` with an empty callback (since `Print` is the root operator, there is no remaining processing at this point).

```
// Schema edname: String, eid: Int
// generating Emp table as Buffer
val Emp = Buffer() // preloaded
1 ⟦ Print(Agg(Scan("Emp"))(x => x("edname"))(0)((agg,x) => agg + 1)).exec(t => ()) ⟧
```

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

```
2 ⟦ Agg(Scan("Emp"))(x => x("edname"))(0)(
  (agg,x) => agg + 1).exec(t => printRec(t)) ⟧
```

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.

```
// generating a hash map data structure
3 val hm = new HashMap(Seq(StringField("edname")), Seq(IntField("count")))
  ⟦ Scan("Emp").exec{t =>
    val key = t("edname")
    hm.update(key, 0) { c => c + 1 }
  }
  for (t <- hm) printRec(t) ⟧
```

The `Scan.exec` method is then evaluated and generates the scan of all the tuples in the table. The second part of the `Agg` method is also evaluated, the `for (t <- hm)` comprehension calls the `foreach` method of the `HashMap` and generates a for loop:

```
// generating code for the scan method, hash map update and printing tuples
4 val hm = new HashMap(Seq(StringField("edname")), Seq(IntField("count")))
  for (tuple <- Emp) {
    val key = tuple("edname")
    hm.update(key, 0) { c => c + 1 }
  }
  for (tuple <- hm)
    printf("%s|%d\n", tuple("edname"), tuple("count"))
```

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.

```
// 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.

```
1 // generating Emp table as two flat arrays: ednames and eids
  int table_length = ... // preloaded
  char** ednames = ...
  int* eids = ...
  ⟦ Print(Agg(Scan("Emp"))(x => x("edname"))(0)((agg,x) => agg + 1)).exec(t => ()) ⟧
2 // table
  int table_length = ... // preloaded
  char** ednames = ...
  int* eids = ...
  // expanding the aggregate operator with Scan as a child and printRec as a callback
  ⟦ Agg(Scan("Emp"))(x => x("edname"))(0)(
    (agg,x) => agg + 1).exec(t => printRec(t)) ⟧
```

The following step, i.e., evaluating `Agg.exec`, instantiates a new generation-time `HashMap` instance. As discussed in Section 4.2, a high-level hash map specializes to flat arrays (i.e., `key_ednames` and `used`) as illustrated in the following steps.

```
3 // table
  int table_length = ... // preloaded
  char** ednames = ...
  int* eids = ...
⟦ val hm = new HashMap(Seq(StringField("edname")), Seq(IntField("count")))
  Scan("Emp").exec{t =>
    val key = t("edname")
    hm.update(key, 0) { c => c + 1 }
  }
  for (t <- hm) printRec(t) ⟧
4 // table
  int table_length = ... // preloaded
  char** ednames = ...
  int* eids = ...
  // generating HashMap - hm as flat arrays: key_edname and used
  int* agg_count = malloc(size * sizeof(int));
  char* key_edname = calloc(size * sizeof(char*));
  int* used = malloc(size * sizeof(int));
  int next = 0;
  ⟦ Scan("Emp").exec { t =>
    val key = t("edname")
    hm.update(key, 0) { c => c + 1 }
  }
  for (t <- hm) printRec(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 definitions
struct Anon1017206272 { // string
  char* str;
  int length;
};
struct Anon711418583 {
  struct Anon1017206272 edname_166;
};
struct Anon2377293 {
  long _0;
};
struct Anon327205970 { // Emp record
  int eid_165;
  struct Anon1017206272 edname_166;
};
struct Anon1877930583 { // Agg entry
  bool exists;
  struct Anon711418583 key;
  struct Anon2377293 aggs;
};
// main function
int main(int x0, char** x1) {
  long x2 = DEFAULT_INPUT_SIZE;
  long x3 = x2;
  long x4 = 0L;
  char** x7 = (char**)malloc(x2 *
              sizeof(char*));
  char** x8 = x7;
  int* x9 = (int*)malloc(x2 *
            sizeof(int));
  int* x10 = x9;
  long x14 = 0L;
  bool x15 = false;
  int x11 = open("Emp.tbl",0);
  long x12 = fsize(x11);
  char* x13 = mmap(0, x12, PROT_READ,
        MAP_FILE | MAP_SHARED, x11, 0);
  bool x101 = !true;
  for (;;) {
    // ... parse and load data elided ...
    char** x96 = x8;
    char* x53 = x13+x42;
    x96[x76] = x53;
    int* x98 = x10;
    long x51 = x50 - x42;
    int x54 = (int)x51;
    x98[x76] = x54;
    x4 = x4 + 1;
    x15 = x101;
  }
  long x118 = DEFAULT_AGG_HASH_SIZE ;
  long x133 = x118 - 1L;
  long x158 = 0L << 6;
  struct Anon2377293 x222;
  x222._0 = 1L;
  bool x252 = true == false;
  int x106;
  for(x106=0; x106 < 1; x106++) {
    bool x107 = false;
    long x108 = 0L;
    char** x111 = x8;
    char** x112 = x111;
    int* x113 = x10;
    int* x114 = x113;
    struct Anon1877930583* x119 = (struct
       Anon1877930583*)malloc(x118 *
       sizeof(struct Anon1877930583));
    struct Anon1877930583* x120 = x119;
    long x121 = 0L;
    long* x122 = (long*)malloc(x118 *
                 sizeof(long));
    long x124;
    for(x124=0L; x124 < x118; x124++) {
      // ... initialize hash map elided ...
    }
    long x136 = 0L;
    // scan loop over Emp table
    for (;;) {
      bool x139 = x107;
      bool x140 = !x139;
      bool x145 = x140;
      if (x140) {
        long x141 = x108;
        long x142 = x4;
        bool x143 = x141 < x142;
        x145 = x143;
      }
      bool x146 = x145;
      if (!x146) break;
      long x148 = x108;
      x108 = x108 + 1;
      char* x150 = x112[x148];
      int x151 = x114[x148];
      long x157 = 5381;
      int x156 = 0;
      for(; x156 < x151; x156++) {
       x157 = ((x157 << 5) + x157)
              + x150[x156]; } // hash
      long x159 = x158 + x157;
      long x160 = x159 & x133;
      long x161 = x160;
      struct Anon1877930583* x162 = x120;
      struct Anon1877930583 x163 = x162[x160];
      struct Anon1877930583 x164 = x163;
      bool x165 = x163.exists;
      bool x178 = x165;
      if (x165) { // hash lookup: fast path
        struct Anon1877930583 x166 = x164;
        struct Anon711418583 x167 = x166.key;
        struct Anon1017206272 x168 = x167.edname_166;
        int x171 = x168.length;
        bool x172 = x171 == x151;
        bool x175 = x172;
        if (x172) {
          char* x170 = x168.str;
          bool x174 = strncmp(x170,x150,x171) == 0;
          x175 = x174;
        }
        bool x176 = x175;
        x178 = x176;
      }
      bool x179 = x178;
      if (x179) {
        struct Anon2377293 x180 = x163.aggs;
        struct Anon711418583 x184 = x163.key;
        long x181 = x180._0;
        long x182 = x181 + 1L;
        struct Anon2377293 x183;
        x183._0 = x182;
        struct Anon1877930583 x185;
        x185.exists = true;
        x185.key = x184;
        x185.aggs = x183;
        x162[x160] = x185;
      } else { // open addressing loop (slow path)
        struct Anon1017206272 x152;
        x152.str = x150;
        x152.length = x151;
        struct Anon711418583 x155;
        x155.edname_166 = x152;
        struct Anon1877930583 x223;
        x223.exists = true;
        x223.key = x155;
        x223.aggs = x222;
        for (;;) {
          struct Anon1877930583 x188 = x164;
          bool x189 = x188.exists;
          bool x232;
          if (x189) {
            struct Anon711418583 x190 = x188.key;
            struct Anon1017206272 x191 =
                     x190.edname_166;
            int x194 = x191.length;
            bool x195 = x194 == x151;
            bool x198 = x195;
            if (x195) {
              char* x193 = x191.str;
              bool x197 = strncmp(x193,x150,x194) == 0;
              x198 = x197;
            }
            bool x199 = x198;
            bool x219;
            if (x199) {
              struct Anon2377293 x200 = x188.aggs;
              long x204 = x161;
              struct Anon1877930583* x206 = x120;
              long x201 = x200._0;
              long x202 = x201 + 1L;
              struct Anon2377293 x203;
              x203._0 = x202;
              struct Anon1877930583 x205;
              x205.exists = true;
              x205.key = x190;
              x205.aggs = x203;
              x206[x204] = x205;
              x219 = false;
            } else {
              long x209 = x161;
              long x210 = x209 + 1L;
              long x211 = x210 & x133;
              x161 = x211;
              struct Anon1877930583* x213 = x120;
              struct Anon1877930583 x214 = x213[x211];
              x164 = x214;
              x219 = true;
            }
            x232 = x219;
          } else {
            long x221 = x161;
            struct Anon1877930583* x224 = x120;
            x224[x221] = x223;
            long x226 = x121;
            x122[x226] = x221;
            x121 = x121 + 1L;
            x232 = false;
          }
          if (!x232) break;
        }
      }
    }
    long x241 = x121;
    long x243;
    for(x243=0L; x243 < x241; x243++) {
      // ... print output elided ...
    }
    free(x119);
    free(x122);
  }
  return 0;
}
```

Figure 14: The final C code LB2 generates for the aggregate query shown at the beginning of Appendix B.2

```
5 // table
  int table_length = ... // preloaded
  char** ednames = ...
  int* eids = ...
  // HashMap - hm
  int* agg_count = malloc(size * sizeof(int));
  char* key_edname = calloc(size * sizeof(char*));
  int* used = malloc(size * sizeof(int));
  int next = 0;
  // generating the Scan loop
  for (int i = 0 ; i < table_length ; i++) {
⟦ val tuple = Record(Seq(ednames(i),eids(i)), Seq(StringField("edname"),IntField("eid")))
   val idx = defaultHash(tuple("edname")) % size
   if (isEmpty(key_ednames(idx))) { // disregard possibility of hash collisions
     used(next) = idx; next += 1
     key_ednames(idx) = tuple("edname"); agg_count(idx) = 0 + 1
   } else agg_count(idx) = agg_count(idx) + 1 ⟧
  }
 ⟦ for (tuple <- hm)
    printf("%s|%d\n", tuple("edname"), tuple("count")) ⟧
```

Finishing execution of the last remaining Scala bracket dissolves the `Record` access and hash map to individual values i.e., `ednames[i]` and arrays i.e., `used` and `key_ednames`. Lastly, the `for (tuple <- hm)` comprehension completes the generated C code:

```
6 // table
  int table_length = ... // preloaded
  char** ednames = ...
  int* eids = ...
  // HashMap - hm
  int* agg_count = malloc(size * sizeof(int));
  char* key_edname = calloc(size * sizeof(char*));
  int* used = malloc(size * sizeof(int));
  int next = 0;
  // Scan loop, update the hash map with aggregate computation
  for (int i = 0 ; i < table_length ; i++) {
    int idx = stringHash(ednames[i]) % size;
    if (key_dnames[idx] == 0) { // disregard possibility of hash collisions
      used[next] = idx; next += 1
      key_ednames[idx] = ednames[i]; agg_count[idx] = 0 + 1
    } else agg_count[idx] = agg_count[idx] + 1
  }
 ⟦ for (tuple <- hm)
    printf("%s|%d\n", tuple("edname"), tuple("count")) ⟧
```

```
7 // table
  int table_length = ...
  char** ednames = ...
  int* eids = ...
  // HashMap
  int* agg_count = malloc(size * sizeof(int));
  char* key_edname = calloc(size * sizeof(char*));
  int* used = malloc(size * sizeof(int));
  int next = 0;
  // Scan loop, update the hash map with aggregate computation
  for (int i = 0 ; i < table_length ; i++) {
    int idx = stringHash(ednames[i]) % size;
    if (key_dnames[idx] == 0) { // disregard possibility of hash collisions
      used[next] = idx; next += 1
      key_ednames[idx] = ednames[i]; agg_count[idx] = 0 + 1
    } else agg_count[idx] = agg_count[idx] + 1
  }
  for (int idx = 0; idx < used ; idx++) {
    int pos = used[idx];
    printf("%s|%d|\n", key_edname[pos], agg_count[pos]);
  }
```

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.