JFP **29**, e9, 31 pages, 2019. © Cambridge University Press 2019
doi:10.1017/S0956796819000054

A SQL to C compiler in 500 lines of code

TIARK ROMPF
Purdue University, West Lafayette, Indiana, USA
(*e-mail:* tiark@purdue.edu)

NADA AMIN
University of Cambridge, Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
(*e-mail:* na482@cl.cam.ac.uk)

Abstract

We present the design and implementation of a SQL query processor that outperforms existing database systems and is written in just about 500 lines of Scala code – a convincing case study that high-level functional programming can handily beat C for systems-level programming where the last drop of performance matters. The key enabler is a shift in perspective toward generative programming. The core of the query engine is an interpreter for relational-algebra operations, written in Scala. Using the open-source lightweight modular staging framework, we turn this interpreter into a query compiler with very low effort. To do so, we capitalize on an old and widely known result from partial evaluation: the first Futamura projection, which states that a process that can specialize an interpreter to any given input program is equivalent to a compiler. In this context, we discuss lightweight modular staging programming patterns such as mixed-stage data structures (e.g., data records with static schema and dynamic field components) and techniques to generate low-level C code, including specialized data structures and data loading primitives.

1 Introduction

Let us assume that we want to implement a serious, performance-critical, piece of system software, like a database engine that processes SQL queries. Would it be a good idea to pick a high-level language and a mostly functional programming style? Most people would reply within a range from 'probably not' to 'surely you must be joking': for systems-level programming, C reigns supreme as the language of choice.

But let us do a quick experiment. We download a data set from the Google Books NGram Viewer project: a 1.7 GB file in CSV format that contains book statistics of words starting with the letter 'a'. As a first step to perform further data analysis, we load this file into a database system, e.g., MySQL:

```
mysqlimport --local mydb 1gram_a.csv
```

When we run this command we can safely take a coffee break, as the import will take a good 5 minutes on a decently modern laptop. Once our data have loaded and we have returned from the break, we would like to run a simple SQL query, perhaps to find all entries that match a given keyword:

```
select * from 1gram_a where phrase = 'Attention'
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

Unfortunately, we will have to wait another 50 s for an answer. While we are waiting, we may start to look for alternative ways to analyze our data file. We can write an AWK script to process the CSV file directly, which will take 45 s to run. Implementing the same query as a Scala program will get us to 13 s. If we are still not satisfied and rewrite it in C using memory-mapped I/O, we can get down to 3.2 s.

Of course, this comparison may not seem entirely fair. The database system is generic. It can run many kinds of queries, possibly in parallel, and with transaction isolation in the presence of updates. It may also automatically create caches and indexes to speed up future queries once the data are loaded. In contrast to general-purpose systems, hand-written queries run faster, but they are one-off, specialized solutions, unsuited to rapid exploration of a data set. In fact, this gap between general-purpose systems and specialized solutions has been noted many times in the database community (Zukowski *et al.*, 2005; Stonebraker *et al.*, 2007), with prominent researchers arguing that 'one size fits all' is an idea whose time has come and gone (Stonebraker & Çetintemel, 2005). While specialization is clearly necessary for performance, would it not be nice to have the best of both worlds: being able to write generic high-level code while programmatically deriving the specialized, low-level code that is executed?

In this article, we show the following:

- Despite common database systems consisting of millions of lines of code, the essence of a SQL engine is nice, clean, and elegantly expressed as a functional interpreter for relational algebra – at the expense of performance compared to hand-written queries. We present the pieces step by step in Section 2.
- While the straightforward interpreted engine is rather slow, we show how we can turn it into a query compiler that generates fast code with very little modification to the code. The key technique is to *stage* the interpreter using lightweight modular staging (LMS) Rompf & Odersky, 2012, which enables specializing the interpreter for any given query (Section 3).
- Implementing a fast database engine requires techniques beyond simple code generation. Efficient data structures are a key concern, and we show how we can use staging to support specialized hash tables and efficient data layouts, in particular column storage (Section 4).
- Another key concern is low-level control over IO and data representations. By using LMS to generate C code, we can support specialized type representations and memory-mapped IO to eliminate data copying (Section 5).
- We compare the performance of our query engine to the widely used PostgreSQL database system, showing speedups from 4× to 7× for *filter*, *join*, and *group-by* queries over a data set of Amazon food and movie reviews (Section 6).
- We present a glimpse of how this query engine can be grown into a full-scale system, summarizing results from SIGMOD'18 (Tahboub *et al.*, 2018) and OSDI'18 (Essertel *et al.*, 2018), including experimental results on the standard TPC-H benchmark (Section 7).

The SQL engine presented here is deliberately simple, but can be extended in straightforward ways. The database community, too, has significant work on query-to-native compilation, mostly based on LLVM (Neumann, 2011). A first attempt at building a SQL

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
tid,time,title,room
1,09:00 AM,Erlang 101 - Actor and Multi-Core Programming,New York Central
2,09:00 AM,Program Synthesis Using miniKanren,Illinois Central
3,09:00 AM,Make a game from scratch in JavaScript,Frisco/Burlington
4,09:00 AM,Intro to Cryptol and High-Assurance Crypto Engineering,Missouri
5,09:00 AM,Working With Java Virtual Machine Bytecode,Jeffersonian
6,09:00 AM,Let's build a shell!,Grand Ballroom E
7,12:00 PM,Golang Workshop,Illinois Central
8,12:00 PM,Getting Started with Elasticsearch,Frisco/Burlington
9,12:00 PM,Functional programming with Facebook React,Missouri
10,12:00 PM,Hands-on Arduino Workshop,Jeffersonian
11,12:00 PM,Intro to Modeling Worlds in Text with Inform 7,Grand Ballroom E
12,03:00 PM,Mode to Joy - Diving Deep Into Vim,Illinois Central
13,03:00 PM,Get 'go'ing with core.async,Frisco/Burlington
14,03:00 PM,What is a Reactive Architecture,Missouri
15,03:00 PM,Teaching Kids Programming with the Intentional Method,Jeffersonian
16,03:00 PM,Welcome to the wonderful world of Sound!,Grand Ballroom E
```

Fig. 1. Input file `talks.csv` for running example.

engine in LMS won a best paper award at VLDB'14 (Klonatos *et al.*, 2014). The present paper originated in a tutorial given at CUFP'14, in an attempt to distill the essence of the VLDB work and present a more elegant design. A previous version was published as a functional pearl at ICFP'15 (Rompf & Amin, 2015). The present version adds Section 6 (performance evaluation), Section 7 (scaling to a realistic SQL engine), and Section 8 (related work) as new material. Since then, the approach has been extended by the first author's group at Purdue into a complete SQL engine, which, in about 2,500 lines of code, includes support for indexes and parallelism, and is able to run the full TPC-H benchmark (The Transaction Processing Council, 2002). Results are competitive with the best SQL compilers from the database community and were published at SIGMOD'18 (Tahboub *et al.*, 2018) and OSDI'18 (Essertel *et al.*, 2018). Section 7 summarizes these results. The full code accompanying the present article is available at:

`scala-lms.github.io/tutorials/query.html`

2 A SQL interpreter, step by step

We start with a small data file for illustration purposes (see Figure 1). This file, `talks.csv`, contains a list of talks from a recent conference, with identifier, time, title of the talk, and room where it takes place.

It is not hard to write a short program in Scala that processes the file and computes a simple query result. As a running example, we want to find all talks at 9 am and print out their rooms and titles. Here is the code:

```
printf("room,title")
val in = new Scanner("talks.csv")
in.next('\n')
while (in.hasNext) {
  val tid   = in.next(',')
  val time  = in.next(',')
  val title = in.next(',')
  val room  = in.next('\n')
  if (time == "09:00 AM")
    printf("%s,%s\n",room,title)
}
in.close
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

We use a `Scanner` object from the standard library to tokenize the file into individual data fields and print out only the records and fields we are interested in.

Running this little program produces the following result, just as expected:

```
room,title
New York Central,Erlang 101 - Actor and Multi-Core Programming
Illinois Central,Program Synthesis Using miniKanren
Frisco/Burlington,Make a game from scratch in JavaScript
Missouri,Intro to Cryptol and High-Assurance Crypto Engineering
Jeffersonian,Working With Java Virtual Machine Bytecode
Grand Ballroom E,Let's build a shell!
```

While it is relatively easy to implement very simple queries in such a way, and the resulting programs will run fast, the complexity gets out of hand quickly. So let us go ahead and add some abstractions to make the code more general.

The first thing we add is a class to encapsulate data records:

```
case class Record(fields: Fields, schema: Schema) {
  def apply(name:  String) = fields(schema indexOf name)
  def apply(names: Schema) = names map (apply _)
}
```

And some auxiliary type definitions:

```
type Fields = Vector[String]
type Schema = Vector[String]
```

Each records contains a list of field values and a *schema*, for now only a heading, i.e., a list of field names. With that, it provides a method to look up a field value, given a field name, and another version of this method that return a list of values, given a list of names. This will make our code independent of the order of fields in the file. Another thing that is bothersome about the initial code is that I/O boilerplate such as the scanner logic is intermingled with the actual data processing. To fix this, we introduce a method `processCSV` that encapsulates the input handling:

```
def processCSV(file: String)(yld: Record => Unit): Unit = {
  val in = new Scanner(file)
  val schema = in.next('\n').split(",").toVector
  while (in.hasNext) {
    val fields = schema.map(n => in.next(if (n == schema.last)
        '\n' else ','))
    yld(Record(fields, schema))
  }
}
```

This method fully abstracts over all file handling and tokenization. It takes a file name as input, along with a callback that it invokes for each line in the file with a freshly created record object. The schema is read from the first line of the file.

With these abstractions in place, we can express our data-processing logic in a much nicer way:

```
printf("room,title")
processCSV("talks.csv") { rec =>
  if (rec("time") == "09:00 AM")
    printf("%s,%s\n",rec("room"),rec("title"))
}
```

The output will be exactly the same as before.

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
// relational-algebra ops
sealed abstract class Operator
case class Scan(name: Table) extends Operator
case class Print(parent: Operator) extends Operator
case class Project(out: Schema, in: Schema, parent: Operator) extends Operator
case class Filter(pred: Predicate, parent: Operator) extends Operator
case class Join(parent1: Operator, parent2: Operator) extends Operator
case class HashJoin(parent1: Operator, parent2: Operator) extends Operator
case class Group(keys: Schema, agg: Schema, parent: Operator) extends Operator

// filter predicates
sealed abstract class Predicate
case class Eq(a: Ref, b: Ref) extends Predicate
case class Ne(a: Ref, b: Ref) extends Predicate

sealed abstract class Ref
case class Field(name: String) extends Ref
case class Value(x: Any) extends Ref
```

Fig. 2. Query-plan language (relational-algebra operators).

2.1 Parsing SQL queries

While the programming experience has much improved, the query logic is still essentially hard-coded. What if we want to implement a system that can itself answer queries from the outside world – say, respond to SQL queries it receives over a network connection?

We will build a SQL interpreter on the top of the existing abstractions next. But first we need to understand what SQL queries *mean*. We follow the standard approach in database systems of translating SQL statements to an internal *query execution plan* representation – a tree of relational-algebra operators. The `Operator` data type is defined in Figure 2, and we will implement a function `parseSql` that produces instances of that type.

Here are a few examples. For a query that returns its whole input, we get a single table-scan operator:

```
parseSql("select * from talks.csv")
↪ Scan("talks.csv")
```

If we select specific fields, with possible renaming, we obtain a *projection* operator with the table scan as parent:

```
parseSql("select room as where, title as what from talks.csv")
↪ Project(Vector("where","what"),Vector("room","title"),
      Scan("talks.csv"))
```

And if we add a condition, we obtain an additional *filter* operator:

```
parseSql("select room, title from talks.csv where time='09:00 AM'")
↪ Project(Vector("room","title"),Vector("room","title"),
      Filter(Eq(Field("time"),Value("09:00 AM")),
        Scan("talks.csv")))
```

Finally, we can use joins, aggregations (groupBy), and nested queries. Here is a more complex query that finds all different talks that happen at the same time in the same room (hopefully there are none!):

```
parseSql("select *
  from (select time, room, title as title1 from talks.csv)
  join (select time, room, title as title2 from talks.csv)
  where title1 <> title2")
↪ Filter(Ne(Field("title1"),Field("title2")),
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
def stm: Parser[Operator] =
  selectClause ~ fromClause ~ whereClause ~ groupClause ^^ {
    case p ~ s ~ f ~ g => g(p(f(s))) }
def selectClause: Parser[Operator=>Operator] =
  "select" ~> ("*" ^^^ idOp | fieldList ^^ {
    case (fs,fs1) => Project(fs,fs1,_:Operator) })
def fromClause: Parser[Operator] =
  "from" ~> joinClause
def whereClause: Parser[Operator=>Operator] =
  opt("where" ~> predicate ^^ { p => Filter(p, _:Operator) })
def joinClause: Parser[Operator] =
  repsep(tableClause, "join") ^^ { _.reduce((a,b) => Join(a,b)) }
def tableClause: Parser[Operator] =
  tableIdent ^^ { case table => Scan(table, schema, delim) } |
  ("(" ~> stm <~ ")")
// 30 lines elided
```

Fig. 3. Combinator parsers for SQL grammar.

```
Join(
  Project(Vector("time","room","title1"),Vector(...),
    Scan("talks.csv")),
  Project(Vector("time","room","title2"),Vector(...),
    Scan("talks.csv")))
```

2.2 Parser combinators

In good functional programming style, we use Scala's combinator-parser library (Odersky & Rompf, 2014; Jonnalagedda *et al.*, 2014) to define our SQL parser. The details are not overly illuminating, but we show an excerpt in Figure 3. While the code may look dense on first glance, it is rather straightforward when read top-to-bottom. The important bit is that the result of parsing a SQL query is an `Operator` object, a representation we will focus on next.

2.3 Interpreting relational-algebra operators

Given that the result of parsing a SQL statement is a query execution plan, we need to specify how to turn such a plan into actual query execution. The classical database model would be to define a stateful iterator interface with `open`, `next`, and `close` functions for each type of operator (also known as *volcano model* (Graefe, 1994)). In contrast to this traditional *pull-driven* execution model, recent database work proposes a *push-driven* model to reduce indirection (Neumann, 2011).

Working in a higher-order functional language and coming from a background informed by PL theory, we consider a push model to be a more natural fit from the start: we would like to give a compositional account of what an operator does, and it is easy to describe the semantics of each operator in terms of what records it emits to its caller. This means that we can define a semantic domain as type

```
type Semant = (Record => Unit) => Unit
```

with the idea that the argument is a callback that is invoked for each emitted record. With that, we describe the meaning of each operator through a function `execOp` with the following signature:

```
def execOp: Operator => Semant
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

Even without these considerations, we might pick the push mode of implementation for completely pragmatic reasons: the executable code corresponds almost directly to a textbook definition of the query operators, and it would be hard to imagine an implementation that is clearer or more concise. The following code might therefore serve as a definitional interpreter in the spirit of Reynolds (1972,1998):

```
def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
  case Scan(filename) =>
    processCSV(filename)(yld)
  case Print(parent) =>
    execOp(parent) { rec =>
      printFields(rec.fields) }
  case Filter(pred, parent) =>
    execOp(parent) { rec =>
      if (evalPred(pred)(rec)) yld(rec) }
  case Project(newSchema, parentSchema, parent) =>
    execOp(parent) { rec =>
      yld(Record(rec(parentSchema), newSchema)) }
  case Join(left, right) =>
    execOp(left) { rec1 =>
      execOp(right) { rec2 =>
        val keys = rec1.schema intersect rec2.schema
        if (rec1(keys) == rec2(keys))
          yld(Record(rec1.fields ++ rec2.fields,
                     rec1.schema ++ rec2.schema)) }}
}
```

So what does each operator do? A table scan just means that we are reading an input file through our previously defined `processCSV` method. A print operator prints all the fields of every record that its parent emits. A filter operator evaluates the predicate, for each record its parent produces, and if the predicate holds, it passes the record on to its own caller. A projection rearranges the fields in a record before passing it on. A join, finally, matches every single record it receives from the left against all records from the right, and if the fields with a common name also agree on the values, it emits a combined record. Of course this is not the most efficient way to implement a join, and adding an efficient hash-join operator is straightforward. The same holds for the group-by operator, which we have omitted so far. We will come back to this in Section 4.

To complete this section, we show the auxiliary functions used by `execOp`. Function `evalRef` evaluates a reference, which may denote either a constant value or a record-field reference:

```
def evalRef(p: Ref)(rec: Record) = p match {
  case Value(a: String) => a
  case Field(name) => rec(name)
}
```

Function `evalPred` evaluates an equality or disequality predicate on a given `Record`:

```
def evalPred(p: Predicate)(rec: Record) = p match {
  case Eq(a,b) => evalRef(a)(rec) == evalRef(b)(rec)
  case Ne(a,b) => evalRef(a)(rec) != evalRef(b)(rec)
}
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

Function `printFields` prints a list of fields as formatted output:

```
def printFields(fields: Fields) =
  printf(fields.map(_ => "%s").mkString("",",","\n"), fields: _*)
```

Finally, to put everything together, we provide a main object that integrates parsing and execution, and that can be used to run queries against CSV files from the command line:

```
object Engine {
  def main(args: Array[String]) {
    if (args.length != 1)
      return println("usage: engine <sql>")
    val ops = parseSql(args(0))
    execOp(Print(ops)) { _ => }
  }
}
```

With the code in this section, which is about 100 lines combined, we have a fully functional query engine that can execute a practically relevant subset of SQL.

But what about performance? We can run the Google Books query on the 1.7 GB data file from Section 1 for comparison, and the engine we have built will take about 45 s. This is about the same as an AWK script, which makes sense, as AWK is also an interpreted language. Compared to our starting point, handwritten scripts that ran in 10 s, the interpretive overhead we have added is clearly visible.

3 From interpreter to compiler

We will now show how we can turn our relatively slow query interpreter into a query compiler that produces Scala or C code that is practically identical to the handwritten queries that were the starting point of our development in Section 2.

3.1 Futamura projections

The key idea behind our approach goes back to early work on partial evaluation in the 1970s, namely the notion of *Futamura projections* (Futamura, 1971). The setting is to consider programs with two inputs, one designated as static and one as dynamic. A program specializer or partial evaluator `mix` is then able to specialize a program p with respect to a given static input. The key use case is if the program is an interpreter:

```
result   = interpreter(source, input)
```

Then specializing the interpreter with respect to the source program yields a program that performs the same computation on the dynamic input, but faster:

```
target   = mix(interpreter, source)
result   = target(input)
```

This application of a specialization process to an interpreter is called the first Futamura projection. In total, there are three Futamura projections:

```
target   = mix(interpreter, source)     (1)
compiler = mix(mix, interpreter)        (2)
cogen    = mix(mix, mix)                (3)
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

The second Futamura projection observes that if we can automate the process of specializing an interpreter to any static input, we obtain a program equivalent to a compiler. Finally the third Futamura projection observes that specializing a specializer with respect to itself yields a system that can generate a compiler from any interpreter given as input (Consel & Danvy, 1993).

In our case, we do not rely on a fully automatic program specializer, but we delegate some work to the programmer to change our query interpreter into a program that specializes itself by treating queries as static data and data files as dynamic input. In particular, we use the following variant of the first Futamura projection:

```
target   = staged-interpreter(source)
```

Here, `staged-interpreter` is a version of the interpreter that has been *annotated* by the programmer. This idea was also used in bootstrapping the first implementation of the Futamura projections by Neil Jones and others in Copenhagen (Jones *et al.*, 1993). The role of the programmer can be understood as being part of the `mix` system, but we will see that the job of converting a straightforward interpreter into a staged interpreter is relatively easy.

3.2 Lightweight modular staging

Staging or multistage programming describes the idea of making different computation stages explicit in a program, where the *present stage* program generates code to run in a *future stage*. The concept goes back at least to Jørring and Scherlis (Jørring & Scherlis, 1986), who observed that many programs can be separated into stages, distinguished by frequency of execution or by availability of data. Taha & Sheard (2000) introduced the language MetaML and made the case for making such stages explicit in the programming model through the use of quotation operators, as known from Lisp and Scheme macros.

LMS (Rompf & Odersky, 2012) is a staging technique based on types: instead of syntactic quotations, we use the Scala type system to designate future stage expressions. Where any regular Scala expression of type `Int`, `String`, or in general `T` is executed normally, we introduce a special type constructor `Rep[T]` with the property that all operations on `Rep[Int]`, `Rep[String]`, or `Rep[T]` objects will generate code to perform the operation later.

Here is a simple example of using LMS:

```
val power4 = new LMS_Driver[Int,Int] {

  def power(b: Rep[Int], x: Int): Rep[Int] =
    if (x == 0) 1 else b * power(b, x - 1)

  def snippet(x: Rep[Int]): Rep[Int] = {
    power(x,4)
  }
}
power4(3)
↪ 81
```

We create a new `LMS_Driver` object. Inside its scope, we can use `Rep` types and corresponding operations. Method `snippet` is the ‘main’ method of this object. The driver will execute `snippet` with a symbolic input. This will completely evaluate the recursive

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

power invocations (since it is a present-stage function) and record the individual expressions in the intermediate representation (IR) as they are encountered. On exit of `snippet`, the driver will compile the generated source code and load it as an executable into the running program.

Here, the generated code corresponds to the following:

```
class Anon12 extends ((Int)=>(Int)) {
  def apply(x0:Int): Int = {
    val x1 = x0*x0
    val x2 = x0*x1
    val x3 = x0*x2
    x3
  }
}
```

The performed specializations are immediately clear from the types: in the definition of `power`, only the base `b` is dynamic (type `Rep[Int]`); everything else will be evaluated statically, at code-generation time. The expression `driver(3)` will then execute the generated code and return the result 81.

3.3 Some LMS internals

While not strictly needed to understand the rest of this paper, familiarity with some of the internals might be useful.

LMS is called *lightweight* because it is implemented as a library instead of baked into a language, and it is called *modular* because there is complete freedom to define the available operations on `Rep[T]` values. To user code, LMS provides just an abstract interface that lifts (selected) functionality of types `T` to `Rep[T]`:

```
trait Base {
  type Rep[T]
}
trait IntOps extends Base {
  implicit def unit(x: Int): Rep[Int]
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
  def infix_*(x: Rep[Int], y: Rep[Int]): Rep[Int]
}
```

Internally, this API is wired to create an IR which can be further transformed and finally unparsed to target code:

```
trait BaseExp {
  // IR base classes: Exp[T], Def[T]
  type Rep[T] = Exp[T]
  def reflectPure[T](x:Def[T]): Exp[T] = .. // insert x into IR graph
}
trait IntOpsExp extends BaseExp {
  case class Plus(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  case class Times(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  implicit def unit(x: Int): Rep[Int] = Const(x)
  def infix_+(x: Rep[Int], y: Rep[Int]) = reflectPure(Plus(x,y))
  def infix_*(x: Rep[Int], y: Rep[Int]) = reflectPure(Times(x,y))
}
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

Another way to look at this structure is as combining a shallow and a deep embedding for an IR object language (Svenningsson & Axelsson, 2012). Methods like `infix_+` can serve as smart constructors that perform optimizations on the fly while building the IR (Rompf *et al.*, 2013). With some tweaks to the Scala compiler (or alternatively using Scala macros (Rompf, 2016b)) we can extend this approach to lift language built-ins like conditionals or variable assignments into the IR, by redefining them as method calls (Rompf *et al.*, 2012).

3.4 Mixed-stage data structures

We have seen above that LMS can be used to unfold functions and generate specialized code based on static values. One central design pattern that will drive the specialization of our query engine is the notion of mixed-stage data structures, which have both static and dynamic components.

Looking again at our earlier `Record` abstraction:

```
case class Record(fields: Vector[String], schema: Vector[String]) {
  def apply(name: String): String = fields(schema indexOf name)
}
```

We would like to treat the schema as static data and treat only the field values as dynamic. The field values are read from the input and vary per row, whereas the schema is fixed per file and per query. We thus go ahead and change the definition of `Records` like this:

```
case class Record(fields: Vector[Rep[String]], schema:
    Vector[String]) {
  def apply(name: String): Rep[String] = fields(schema indexOf name)
}
```

Now the individual fields have type `Rep[String]` instead of `String` which means that all operations that touch any of the fields will need to become dynamic as well. On the other hand, all computations that only touch the schema will be computed at code-generation time. Moreover, `Record` objects are static as well. This means that the generated code will manipulate the field values as individual local variables, instead of through a record indirection. This is a strong guarantee (enforced by the type-checker): records cannot exist in the generated code, unless we provide an API for `Rep[Record]` objects.

3.5 Staged interpreter

As it turns out, this simple change to the definition of records is the only creative one we need to make to obtain a query compiler from our previous interpreter. All other modifications follow by fixing the type errors that arise from this change. We show the full code again in Figure 4. Note that we are now using a staged version of the `Scanner` implementation, which needs to be provided as an LMS module.

3.6 Results

Let us compare the generated code to the one that was our starting point in Section 2. Our example query was the following:

```
select room, title from talks.csv where time = '09:00 AM'
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
val driver = new LMS_Driver[Unit,Unit] {
  type Fields = Vector[Rep[String]]
  type Schema = Vector[String]
  case class Record(fields: Fields, schema: Schema) {
    def apply(name: String):  Rep[String] = fields(schema indexOf name)
    def apply(names: Schema): Fields      = names map (this apply _)
  }
  def processCSV(file: String)(yld: Record => Unit): Unit = {
    val in = new Scanner(file)
    val schema = in.next('\n').split(",").toVector
    while (in.hasNext) {
      val fields = schema.map(n=>in.next(if(n==schema.last)'\n'else','))
      yld(Record(fields, schema))
    }
  }
  def evalRef(p: Ref)(rec: Record): Rep[String] = p match {
    case Value(a: String) => a
    case Field(name) => rec(name)
  }
  def evalPred(p: Predicate)(rec: Record): Rep[Boolean] = p match {
    case Eq(a,b) => evalRef(a)(rec) == evalRef(b)(rec)
    case Ne(a,b) => evalRef(a)(rec) != evalRef(b)(rec)
  }
  def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
    case Scan(filename) =>
      processCSV(filename)(yld)
    case Print(parent) =>
      execOp(parent) { rec =>
        printFields(rec.fields) }
    case Filter(pred, parent) =>
      execOp(parent) { rec =>
        if (evalPred(pred)(rec)) yld(rec) }
    case Project(newSchema, parentSchema, parent) =>
      execOp(parent) { rec =>
        yld(Record(rec(parentSchema), newSchema)) }
    case Join(left, right) =>
      execOp(left) { rec1 =>
        execOp(right) { rec2 =>
          val keys = rec1.schema intersect rec2.schema
          if (rec1(keys) == rec2(keys))
            yld(Record(rec1.fields ++ rec2.fields, rec1.schema ++ rec2.schema)) }}
  }
  def printFields(fields: Fields) =
    printf(fields.map(_ => "%s").mkString("",",","\n"), fields: _*)
  def snippet(x: Rep[Unit]): Rep[Unit] = {
    val ops = parseSql("select room,title from talks.csv where time = '09:00 AM'")
    execOp(PrintCSV(ops)) { _ => }
  }
}
```

Fig. 4. (Staged query interpreter) = compiler. Changes are underlined.

And here is the handwritten code again:

```
printf("room,title")
val in = new Scanner("talks.csv")
in.next('\n')
while (in.hasNext) {
  val tid   = in.next(',')
  val time  = in.next(',')
  val title = in.next(',')
  val room  = in.next('\n')
  if (time == "09:00 AM")
    printf("%s,%s\n",room,title)
}
in.close
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

The generated code from the compiling engine is this:

```
val x1 = new scala.lms.tutorial.Scanner("talks.csv")
val x2 = x1.next('\n')
val x14 = while ({
  val x3 = x1.hasNext
  x3
}) {
  val x5 = x1.next(',')
  val x6 = x1.next(',')
  val x7 = x1.next(',')
  val x8 = x1.next('\n')
  val x9 = x6 == "09:00 AM"
  val x12 = if (x9) {
    val x10 = printf("%s,%s\n",x8,x7)
  } else {
  }
  x1.close
}
```

So, modulo syntactic differences, we have generated exactly the same code. And, of course, this code will run just as fast. Looking again at the Google Books query, where the interpreted engine took 45 s to run the query, we are down again to 10 s but this time *without giving up on generality*.

3.7 Error handling

Both our initial query interpreter and the staged query compiler operate on an untyped syntax tree of query operators. Hence, there is the possibility of runtime errors for queries that refer to nonexistent fields, for example. Such errors manifest as runtime exceptions during execution in the interpreter, and, since the data schema is static, in most cases as exceptions during staging in the query compiler. It is an easy exercise to add a separate semantic analysis pass to checks for such errors, and such a pass could serve as a basis for a proper type-checking pass once other data types than strings are supported (see Section 5). Other errors may of course still occur during execution, caused, e.g., by unexpected IO conditions such as a premature end-of-file.

4 Specializing data structures

In this section, we look at efficient join algorithms that require auxiliary data structures and we show how can leverage generative techniques for this purpose as well, going beyond simple compilation.

4.1 Hash joins and aggregates

A full-fledged SQL engine may include many different kinds of join operators (e.g., inner, outer, anti-, and semi-joins) and a high-level query optimizer that picks a physical realization (e.g., hash join or sort-merge join). As before, we content ourselves with standard inner joins on equality predicates, but we add a variant implemented as hash joins. We also add grouped summation, implemented using hash aggregation. We can either add a simple query optimizer to pick between the nested-loops join operator introduced above and the new hash join operator, or we expose the choice in the SQL syntax. Maintaining

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

the style of a definitional interpreter, we would like to implement the new operators by extending execOp in the following way:

```
def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
  // ... pre-existing operators elided
  case Group(keys, agg, parent) =>
    val hm = new HashMapAgg(keys, agg)
    execOp(parent) { rec =>
      hm(rec(keys)) += rec(agg)
    }
    hm foreach { (k,a) =>
      yld(Record(k ++ a, keys ++ agg))
    }
  case HashJoin(left, right) =>
    val keys = resultSchema(left) intersect resultSchema(right)
    val hm = new HashMapBuffer(keys, resultSchema(left))
    execOp(left) { rec1 =>
      hm(rec1(keys)) += rec1.fields
    }
    execOp(right) { rec2 =>
      hm(rec2(keys)) foreach { rec1 =>
        yld(Record(rec1.fields ++ rec2.fields,
                   rec1.schema ++ rec2.schema))
      }
    }
}
```

An aggregation will collect all records from the parent operator into buckets and accumulate sums in a hash table. Once all records are processed, all key-value pairs from the hash map will be emitted as records. A hash join will insert all records from the left parent into a hash map, indexed by the join key. Afterwards, all the records from the right will be used to look up matching left records from the hash table, and the operator will pass combined records on to its callback. This approach is much more efficient for larger data sets than the naive nested loops join from Section 2.

4.2 Data-structure specialization

What are the implementations of hash tables that we need to provide to make the above code work? We could opt to just use lifted versions of the regular Scala hash tables, i.e., Rep[HashMap[K,V]] objects. However, these are not the most efficient for our case, since they have to support a very generic programming interface. We want to execute only the actual data-dependent operations such as computing hash values, but we do not want to incur any dispatch overhead that, e.g., has to look up a method to perform those hash computations. Moreover, recall our staged Record definition:

```
case class Record(fields: Vector[Rep[String]], schema:
    Vector[String]) {
  def apply(name: String): Rep[String] = fields(schema indexOf name)
}
```

A crucial design choice was to treat records as a purely staging-time abstraction. If we were to use Rep[HashMap[K,V]] objects, we would have to use Rep[Record] objects

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

as well, or at least `Rep[Vector[String]]`. The choice of using `Vector[Rep[String]]` means that all field values will be mapped to individual entities in the generated code. This property naturally leads to a design for data structures in *column-oriented* instead of *row-oriented* order. Instead of working with

```
Collection[ { Field1, Field2, Field3 } ]
```

we work with

```
{ Collection[Field1], Collection[Field2], Collection[Field3] }
```

This layout has other important benefits, e.g., in terms of memory-bandwidth utilization, and is becoming increasingly popular in contemporary in-memory database systems.

Usually, programming in a columnar style is more cumbersome than in a record-oriented manner. But fortunately, we can completely hide the column-oriented nature of our internal data structures behind a high-level record-oriented interface. Let us go ahead and implement a growable `ArrayBuffer` (following the corresponding class from the standard Scala collections library), which will serve as the basis for our `HashMaps`:

```
class ArrayBuffer[T:Typ](dataSize: Int, schema: Schema) {
  val buf = schema.map(f => NewArray[T](dataSize))
  var len = 0
  def +=(x: Seq[Rep[T]]) = {
    this(len) = x
    len += 1
  }
  def update(i: Rep[Int], x: Seq[Rep[T]]) = {
    (buf,x).zipped.foreach((b,x) => b(i) = x)
  }
  def apply(i: Rep[Int]) = {
    buf.map(b => b(i))
  }
}
```

The array buffer is passed a schema on creation, and it sets up one buffer for each of the columns. Here, we keep it simple, yet it would be very easy to introduce specialization, e.g., specialized columns (see Section 5), or sparse or compressed columns for cases where we know that most values will be zero. The `update` and `apply` methods of `ArrayBuffer` still provide a row-oriented interface, working on a set of `Fields` together, but internally access the distinct column buffers.

With this definition of array buffers at hand, we can define a class hierarchy of hash maps, with a common base class and then derived classes for aggregations (storing scalar values) and joins (storing collections of objects):

```
class HashMapBase(keySchema: Schema, schema: Schema) {
  val keys = new ArrayBuffer(keysSize, keySchema)
  val htable = NewArray[Int](hashSize)
  def lookup(k: Fields) = ...
  def lookupOrUpdate(k: Fields)(init: Rep[Int]=>Rep[Unit]) = ...
}
// hash table for groupBy, storing scalar sums
class HashMapAgg(keySchema: Schema, schema: Schema) extends
      HashMapBase(keySchema: Schema, schema: Schema) {
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
    val values = new ArrayBuffer(keysSize, schema)
    def apply(k: Fields) = new {
      def +=(v: Fields) = {
        val keyPos = lookupOrUpdate(k) { keyPos =>
          values(keyPos) = schema.map(_ => 0)
        }
        values(keyPos) = (values(keyPos) zip v) map (_ + _)
      }}
    def foreach(f: (Fields,Fields) => Rep[Unit]): Rep[Unit] =
      for (i <- 0 until keyCount)
        f(keys(i),values(i))
  }
  // hash table for joins, storing lists of records
  class HashMapBuffer(keySchema: Schema, schema: Schema) extends
        HashMapBase(keySchema: Schema, schema: Schema) {
          // ... details elided
  }
```

Note that the hash-table implementation is oblivious of the storage format used by the array buffers. Furthermore, we're freely using object-oriented techniques like inheritance without the usually associated overheads because all these abstractions exist only at code-generation time.

5 Switching to C and optimizing IO

While we have seen impressive speedups just through compilation of queries, let us recall from Section 1 that we can still go faster. By writing our query by hand in C instead of Scala, we were able to run it in 3 s instead of 10 s. The technique there was to use the `mmap` system call to map the input file into memory, so that we could treat it as a simple array instead of copying data from read buffers into string objects.

5.1 Memory-mapped IO and data representations

LMS provides code-generation facilities not only for Scala but also for C. The C back-end provides low-level APIs for pointers and memory management, which we will use to implement memory-mapped IO. One key benefit will be to eliminate data copies and represent strings just as pointers into the memory-mapped file, instead of first copying data into another buffer. But there is a problem: the standard C API assumes that strings are null-terminated, but in our memory-mapped file, strings will be delimited by commas or line breaks.

To this end, we introduce our own operations and data types for data fields. Instead of the previous definition of `Fields` as `Vector[Rep[String]]`, we introduce a small class hierarchy `Value` with the necessary operations:

```
type Fields = Vector[Value]
abstract class Value {
  def print()
  def compare(o: Value): Rep[Boolean]
  def hash: Rep[Long]
}
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
case class StringValue(data: Rep[Pointer[Char]], len: Rep[Int])
    extends Value {
  def print() = ...
  def compare(o: Value) = ...
  def hash = ...
}
case class IntValue(value: Rep[Int]) extends Value {
  def print() = printf("%d",value)
  def compare(o: Value) = o match { case IntValue(v2) => value
    == v2 }
  def hash = value.asInstanceOf[Rep[Long]]
}
```

The new string type StringValue consists of a raw pointer into a buffer of characters and a length. Note that this change is again completely orthogonal to the actual query-interpreter logic. However, we generalize our specialized ArrayBuffer data structure from Section 4 to work with these specialized field representations.

```
abstract class ColBuffer
case class IntColBuffer(data: Rep[Array[Int]]) extends ColBuffer
case class StringColBuffer(data: Rep[Array[Pointer[Char]]],
                           len: Rep[Array[Int]]) extends ColBuffer

class ArrayBuffer(dataSize: Int, schema: Schema) {
  val buf = schema.map {
    case hd if isNumericCol(hd) =>
      IntColBuffer(NewArray[Int](dataSize))
    case _ =>
      StringColBuffer(NewArray[Pointer[Char]](dataSize),
                      NewArray[Int](dataSize))
  }
  ...
  def update(i: Rep[Int], x: Fields) = (buf,x).zipped.foreach {
    case (IntColBuffer(b), IntValue(x)) => b(i) = x
    case (StringColBuffer(b,l), StringValue(x,y)) => b(i) = x;
        l(i) = y
  }
  def apply(i: Rep[Int]): Fields = buf.map {
    case IntColBuffer(b) => IntValue(b(i))
    case StringColBuffer(b,l) => StringValue(b(i),l(i))
  }
}
```

The array buffer now sets up one ColBuffer object for each of the columns. In this version of our query engine we also introduce typed columns, treating a column whose name starts with '#' as numeric. This enables us to use primitive integer arrays for storage of numeric columns instead of a generic binary format.

As the final piece in the puzzle, we provide our own specialized Scanner class that generates mmap calls (supported by a corresponding LMS IR node) and creates Value instances when reading the data:

```
class Scanner(name: Rep[String]) {
  val fd = open(name)
  val fl = filelen(fd)
  val data = mmap[Char](fd,fl)
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
Movies Table:                                   Food Table:
     productid    VARCHAR                            productid    VARCHAR
     userid       VARCHAR                            userid       VARCHAR
     helpful      VARCHAR                            helpful      VARCHAR
     score        INTEGER                            score        INTEGER
     time         INTEGER                            time         INTEGER
```

Fig. 5. Movies and Food tables used in experiments.

```
    var pos = 0
    def next(d: Rep[Char]) = {
      //...
      StringValue(data + start, len)
    }
    def nextInt(d: Rep[Char]) = {
      //...
      IntValue(num)
    }
  }
```

With this, we are able to generate tight C code that executes the Google Books query in 3 s, just like the hand-written optimized C code. The total size of the code is just under 500 (non-blank, non-comment) lines.

The crucial point here is that while we cannot hope to beat hand-written *specialized* C code for a particular query – after all, anything we generate could also be written by hand – we are beating, by a large margin, the highly optimized *generic* C code that makes up the bulk of MySQL, PostgreSQL, and other traditional database systems. By changing the perspective to embrace a generative approach, we are able to raise the level of abstraction, and to leverage high-level functional programming techniques to achieve excellent performance with very concise code.

6 Performance evaluation

We have seen impressive speedups for the Google Books query, but what about more complex queries, including joins and aggregates? To substantiate our claims to performance further, we present additional experiments in the following. Performance results on the standard TPC-H benchmark are discussed in Section 7, summarizing a paper from SIGMOD'18 that describes a more complete SQL engine developed in the same style (Tahboub *et al.*, 2018).

6.1 Amazon reviews

We pick a data set based on Amazon reviews for movies and food products. Figure 5 shows the schema description of the Movies table (1,000,000 tuples) and the Food table (500,000 tuples). The experiment runs three different queries (Filter, Join, and Group-by, shown in Figure 6) and measures the running time of PostgreSQL, our unstaged query interpreter in Scala, and our staged query compiler generating Scala as well as optimized C.

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

Filter	`select * from Movies where helpful = '10/10'`
Join	`select * from Food, Movies where Food.userid = Movies.userid`
Group by	`select productid, sum(score) from Movies group by productid`

Fig. 6. Queries used in experiments.

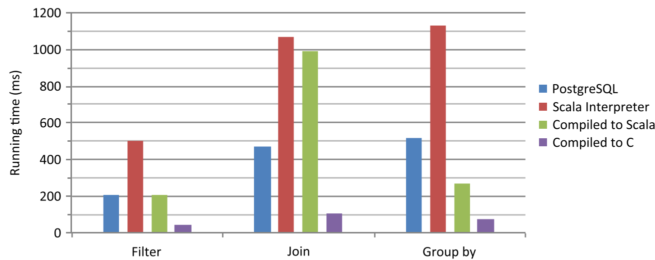


Fig. 7. Performance evaluation: Compiled C code is 4–7× faster than PostgreSQL and 10–15× faster than the Scala interpreter.

Figure 7 shows the results. The access path in all queries is a file scan – i.e., data are processed *in situ* without a pre-loading phase, and without using indices in PostgreSQL. The reported numbers are minimums of five consecutive runs. The filter query scans the Movies table and filters out tuples where the value of the `helpful` attribute is equal to '10/10'. The compiled C code outperforms the Scala interpreter by 11× and PostgreSQL by 4.75×. Similar to the Google Books query, the improvement in Filter is mainly due to removing the overhead of iterator-style query evaluation by operating in push mode rather than pull mode, and removing further interpretive overhead via code generation. The Join query finds Movies and Food reviews written by the same person. The Hash Join implementation builds a hash table on `Food.userid`. The compiled C code is one order of magnitude faster than the unstaged Scala and 4.47× faster than PostgreSQL. Finally, the Group-by query groups data based on the `productid` and then calculates the sum of scores for each group. The optimized C code outperforms unstaged Scala by 15× and PostgreSQL by 7.24×. The large performance improvement in Group-by relative to PostgreSQL is mainly due to compiling specialized data structures, where PostgreSQL has to use a generic representation.

For completeness, Figure 7 also shows the performance of compiling to Scala. While generally in between the Scala interpreter and the compiled C version, the Join query shows only a minimal speedup when generating Scala. This somewhat surprising result exposes one of the pitfalls of running on a managed runtime such as the JVM: the generated code is larger than a certain internal JVM threshold and therefore not considered for just-in-time compilation.

6.2 Environment

All experiments were run on an HP compute node with two 10-core Intel Xeon-E5 processors and 256 GB of memory. The operating system is Red Hat Enterprise Linux 6. We used Scala version 2.11.2, PostgreSQL 9.4 with 50 GB of shared_buffers, and gcc 4.9 with optimization flags `-O3`. The Java Virtual Machine for running Scala programs was configured with 20 GB of heap space.

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

7 Scaling to a realistic SQL engine

LB2 is a more complete SQL engine presented at SIGMOD'18 (Tahboub *et al.*, 2018) and OSDI'18 (Essertel *et al.*, 2018), implemented by the first author's group at Purdue, in particular his graduate students Grégory Essertel, Ruby Tahboub, and James Decker. LB2 was implemented as a full-scale extension of the 500 line engine presented in the preceding sections. It supports indexes and parallelism, and supports enough SQL to run the industry-standard TPC-H benchmark. Performance-wise, LB2 matches and sometimes beats the best query compilers from the database community. This section summarizes key technical details and experimental results from the corresponding publications (Tahboub *et al.*, 2018; Essertel *et al.*, 2018), to which we refer readers for a more complete exposition.

LB2 does not implement a SQL parser of its own. Instead, it uses the SQL parser, type checker, and high-level query optimizer from the Spark SQL project (Armbrust *et al.*, 2015). Flare (Essertel *et al.*, 2018) is an extension of LB2 that integrates directly with Spark and acts as a drop-in accelerator that provides increased performance for Spark workloads when executed on a single node. The inputs to LB2 and Flare are query plans produced by Spark's Catalyst query planner, which LB2 translates to its own operator representation. Data can be stored in multiple formats, including CSV, Apache Parquet, and as straight binary dump of LB2's own internal columnar or row-oriented representation, which can be directly `mmap`'ed back into memory. This way, LB2 delegates caching and paging decisions to the operating system without having to implement a persistence layer of its own.

LB2's internal operator representation is a direct extension of the `Operator` type from Sections 2 and 3. The difference of course is that LB2 needs to support a larger number of operators, including semi joins, anti joins, outer joins, sorting, etc., as well as operators with more complex interfaces, such as group-by with multiple aggregates, non-equi-joins with complex predicates, etc. To handle this larger number of operators, LB2 implements the main operator logic in an instance method `exec` on class `Operator` instead of in an external function `execOp`. The core interface and implementations of `Select` and `HashJoin` are shown as follows:

```
type Pred   = Record => Rep[Boolean]
type KeyFun = Record => Record
// object-oriented operator interface for increased modularity
class Operator {
  def exec(f: Record => Unit): Unit
}
// select operator
class Select(op: Operator)(pred: Pred) extends Operator {
  def exec(cb: Record => Unit) = {
    op.exec { tuple =>
      if (pred(tuple)) cb(tuple)
    }
  }
}
// hash join operator
class HashJoin(left: Operator, right: Operator)(lkey: KeyFun)
    (rkey: KeyFun) extends Operator {
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
  def exec(cb: Record => Unit) = {
    val hm = new HashMapBuffer(resultSchema())
    left.exec { rec =>
      hm += (lkey(rec), rec)
    }
    right.exec { rec =>
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

Note that `HashJoin` adds key selector functions `lkey` and `rkey` to support joins other than natural joins, i.e., joining fields with different names.

Internally, LB2 supports both row-oriented and column-oriented records. To abstract over this choice under a clean interface, `Record` becomes an abstract base class with two concrete implementation classes:

```
// support both row-oriented and column-oriented records with
    a common interface
abstract class Record { def schema: Seq[Field]; def apply(name:
    String): Value }
// implementation subclasses
case class NativeRec(pt: Rep[Pointer[Byte]], 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))
}
```

LB2 also needs to support a variety of data types. Section 5 already introduced a split between field types `Int` and `String` using a common base class `Value`:

```
abstract class Value                // models an attribute's value
case class IntValue(value: Rep[Int]) extends Value { ... }
case class StringValue(data: Rep[Pointer[Char]], length: Rep[Int])
    extends Value { ... }
```

But in this previous setting we used the column name to encode the type, treating columns whose name started with '#' as numeric. For a full-scale SQL engine this is not an option, so LB2 introduces a proper hierarchy of field descriptors to model the types of columns and potentially other column attributes, such as size bounds, nullability, foreign-key constraints, or the presence of indexes.

```
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
```

As in Section 5, the implicit lesson here is that object-oriented abstraction and staging work together just fine.

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

7.1 Index structures

It is often desirable to speed up query execution by using indexes. This way, search over an input relation can be replaced by a lookup operation on an index data structure, reducing a linear-time operation to logarithmic time with tree indexes or expected constant time with hash indexes. Decisions when and how to use an index are typically made during the query planning and optimization phase, based on statistics and metadata. To add index capabilities, LB2 provides a corresponding set of indexed query operators in the same style as other operators. The following code 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 implements sparse and dense index data structures for primary and foreign keys behind a uniform `Index` interface. An `IndexEntryView` class enables iterating over index lookups via `foreach`. An additional method `exists` is used by `IndexSemiJoin` and `IndexAntiJoin` operators. Indexes are implemented in the same way as other data structures (Section 4).

In addition to query evaluation, LB2 generates data loading code for different storage formats, which is extended to create index structures. These can serve as additional access paths on top of underlying data, or as primary partitioned and/or replicated data format, e.g., when there are multiple foreign keys.

7.1.1 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.

7.1.2 String dictionaries

Another form of indexing is compressed columns and dictionary encodings. LB2 implements string dictionaries to optimize operations such as `startsWith`. Individual columns can be marked as dictionary-compressed in the database schema. Building on the `StringValue` and `ColBuffer` abstractions discussed in Section 5, LB2 defines an alternative string representation class `DictValue` and a class `StringDictionary` that stores and provides access to compressed string values.

```
class DictField(name: String) extends Field {
  def dict: StringDictionary = ...
}
```

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

```
class DictValue(idx: Rep[Long]) extends Value {
  def startsWith(p: DictValue) = p.idx <= idx
  ...
}
```

Let us consider the simple case of compressing a single string column. At loading time, the `StringDictionary` is loaded from memory or `mmap`'ed from secondary storage. When the loader reads a string via `Scanner.next`, it creates a `StringValue` and uses the `StringDictionary` to convert it into its `DictValue` compressed form: the index where the `StringValue` is stored inside the `StringDictionary`. String dictionaries do not add any new query operators – i.e., they operate transparently as part of the data representation layer.

7.2 Parallel execution

Query engines typically realize parallelism either explicitly by implementing special *split* and *merge* operators (Mehta & DeWitt, 1995), or implicitly by modifying the internal operator logic to orchestrate parallel execution. LB2 does the latter, and generates code that uses OpenMP.

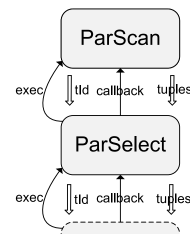
The callback signature for `exec` defined earlier works well in a single-threaded environment, but multi-threaded environments require synchronization and thread-local variables. Thus, LB2 defines a new class `ParOP` with a modified `exec` signature that adds another callback level.

For stateless operators such as `Select`, the parallel implementation is very similar to the single-threaded one. In fact, 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:

```
def ParSelect(op: ParOp)(pred: Pred) =
  parallelPipeline(inner => Select(inner)(pred))(op)
val ps = ParSelect(ParScan("Dep"))(t => t("rank") < 10)
```

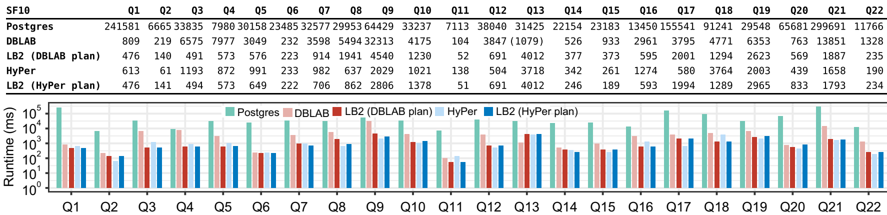
The definition of `ParOp` and `parallelPipeline` is as follows:

```
type ValueCallback = Record => Unit
type DataLoop = ValueCallback => Unit
type ThreadCallback = Rep[Int] => DataLoop => Unit
// parallel operator interface
class ParOp {
  def exec: ThreadCallback => Unit
}
// lifting a sequential operator pipeline
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`

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press



SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Postgres	241581	6665	33835	7980	30158	23485	32577	29953	64429	33237	7113	38040	31425	22154	23183	13450	155541	91241	29548	65681	299691	11766
DBLAB	809	219	6575	7977	3049	232	3598	5494	32313	4175	104	3847	(1079)	526	933	2961	3795	4771	6353	763	13851	1328
LB2 (DBLAB plan)	476	140	491	573	576	223	914	1941	4540	1230	52	691	4012	377	373	595	2001	1294	2623	569	1887	235
HyPer	613	61	1193	872	991	233	982	637	2029	1021	138	504	3718	342	261	1274	580	3764	2003	439	1658	190
LB2 (HyPer plan)	476	141	494	573	649	222	706	862	2806	1378	51	691	4012	246	189	593	1994	1289	2965	833	1793	234

Fig. 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 (results from Tahboub *et al.*, 2018).

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 allows 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, extra work is required for pipeline breakers. For operators such as aggregations, 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. Tahboub *et al.* (2018) present examples of using a `ParHashMap` class as the basis for a parallel aggregation operators. Essertel *et al.* (2018) show the implementation of a parallel hash join operator and also discusses optimizations to take advantage of NUMA (nonuniform memory access) characteristics on large multi-socket machines.

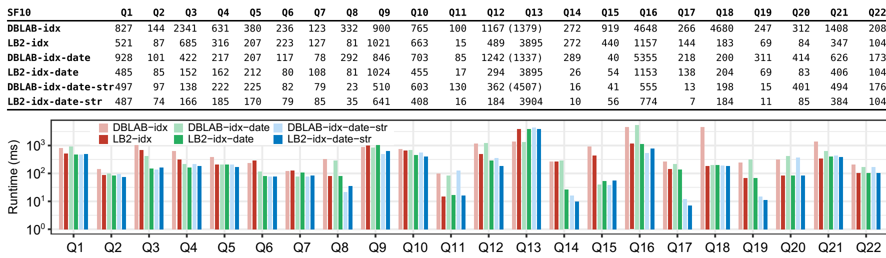
7.3 TPC-H experiments

In this section, we review performance experiments from Tahboub *et al.* (2018), comparing LB2 on the standard TPC-H benchmark to PostgreSQL and two recent state-of-the-art compiled query engines: HyPer (Neumann, 2011) and DBLAB (Shaikhha *et al.*, 2016). HyPer implements code generation using LLVM, and DBLAB is a query compiler that generates C through multiple intermediate languages and lowering passes. Reported numbers are the median of five runs. All experiments are run on a single machine with 4 Xeon E7-88904 CPUs, 18 cores, and 256 GB RAM per socket (1 TB total). We refer readers to (Tahboub *et al.*, 2018) for a full analysis of the experiments.

7.3.1 TPC-H-compliant execution

The first experiment (Figure 8) evaluates the performance of LB2 with only those optimizations that are compliant with the official TPC-H rules, i.e., excluding precomputation and advanced index structures. 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. At first glance, LB2 outperforms Postgres and DBLAB in all queries where query plans are matched. Furthermore, LB2 and HyPer's performance is comparable.

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press



SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
DBLAB-idx	827	144	2341	631	380	236	123	332	900	765	100
LB2-idx	521	87	685	316	207	223	127	81	1021	663	15
DBLAB-idx-date	928	101	422	217	207	117	78	292	846	703	85
LB2-idx-date	485	85	152	162	212	80	108	81	1024	455	17
DBLAB-idx-date-str	497	97	138	222	225	82	79	23	510	603	130
LB2-idx-date-str	487	74	166	185	170	79	85	35	641	408	16

SF10	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
DBLAB-idx	1167	(1379)	272	919	4648	266	4680	247	312	1408	208
LB2-idx	489	3895	272	440	1157	144	183	69	84	347	104
DBLAB-idx-date	1242	(1337)	289	40	5355	218	200	311	414	626	173
LB2-idx-date	294	3895	26	54	1153	138	204	69	83	406	104
DBLAB-idx-date-str	362	(4507)	16	41	555	13	198	15	401	494	176
LB2-idx-date-str	184	3904	10	56	774	7	184	11	85	384	104

Fig. 9. The absolute runtime in milliseconds (ms) after enabling non-TPC-H-compliant indexing, date indexing, and string dictionaries for SF10 using DBLAB plans (results from Tahboub *et al.*, 2018).

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 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. 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. The respective performance gaps can be attributed to various internal implementation choices.

7.3.2 Index optimizations

The second experiment (Figures 9) focuses on evaluating three advanced optimizations that were used by DBLAB to justify a multi-pass compiler pipeline (Shaikhha *et al.*, 2016); primary and foreign key indexes, date indexes, and string dictionaries. In their full generality, these optimizations are not compliant with the TPC-H rules (Chiba & Onodera, 2015) since they incur pre-computation and a duplication of data.

On a query by query analysis, LB2 outperforms DBLAB in join query Q3 by 3× and by 15% and 80% in Q10 and Q5, respectively. Similarly, LB2 is 2–4× faster in semi join and anti join queries Q4, Q22, Q16, and Q21. On the other hand, DBLAB is faster than LB2 in Q7 and Q9 by 3% and 13%, respectively.

The date indexing optimization is used when a table is filtered on a date attribute. This optimization is always beneficial in both systems. DBLAB and LB2 create string dictionaries to speed up commonly used string operations: `equality`, `startsWith`, `endsWith`, and certain forms of `like`. LB2 is 35%–95% faster in Q12, Q17, and Q19 whereas DBLAB is 20% and 50% faster in Q3 and Q8, respectively. Moreover, Q2 and Q14 use `startsWith` and `endsWith`. LB2 is 30% and 60% faster in these queries. Finally, Q9, Q13, and Q16 use `like`, which LB2 does not optimize.

7.3.3 Parallelism

This experiment compares the scalability of LB2 with HyPer (DBLAB does not support parallelism). The five selected queries represent aggregates and join variants. Figure 10

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

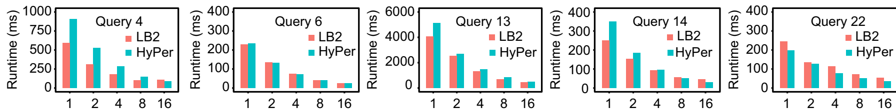


Fig. 10. The absolute runtime in milliseconds (ms) for parallel scaling of LB2 and HyPer with SF10 on 2, 4, 8, and 16 cores (results from Tahboub *et al.*, 2018).

gives the absolute runtime for scaling up LB2 and HyPer for Q4, Q6, Q13, Q14, and Q22 in SF10. 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 two to eight 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.

7.3.4 Conclusion

The experiments show that LB2 can compete against state-of-the-art query compilers from the database community. However, LB2's design is simpler than both DBLAB and HyPer; it is derived from a straightforward query interpreter design and does neither require low-level coding with LLVM nor multiple compiler passes or additional intermediate languages.

8 Related work

8.1 Query compilation

Query compilation itself is not a new idea. Historically, the very first relational database, IBM's System R (Astrahan *et al.*, 1976), was initially designed around a form of templated code generation. However, before the first commercial release, the code generator was replaced by interpreted execution (Chamberlin *et al.*, 1981), since at the time, the benefits of code generation were outweighed by its complexity and issues such as code portability, cost of maintenance, and prevailing I/O-intensive workloads. Indeed, compiling code for query-evaluation pipelines is a nontrivial task. First, code-generation mechanisms need to consider database-level optimizations, compiler-level optimizations, and handling nontraditional data types. Second, code generators need to be extensible and expressive. Third, the generated code should be sufficiently portable, i.e., be easily mapped to a variety of target platforms.

In recent years, query compilation has received renewed interest, mainly because with the decline of Moore's law, I/O is no longer the key bottleneck in data-processing systems, and compute performance has become much more important. Traditional query engines are built around an iterator model (Graefe & McKenna, 1993), which pipelines processing of tuples between operators and hence eliminates unnecessary I/O blocking at the expense of computational overhead. In the new era of large main memory, which reduces the necessary disk I/O, this model loses much of its attractiveness. MonetDB (Boncz *et al.*, 2006) vectorizes query processing by processing blocks of intermediate results, instead of repeatedly invoking the operator interface for each single tuple. On recent query-compilation

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

work, Roa *et al.* (2006) compile queries to JVM bytecode. HIQUE (Krikellas *et al.*, 2010) realized query compilation using code templates, and HyPer (Neumann, 2011) uses LLVM to generate code from a producer/consumer query-execution model, translating traditional query plans into push-based operators.

Hekaton (Diaconu *et al.*, 2013), DryadLINQ (Isard *et al.*, 2007), Impala (Kornacker *et al.*, 2015), and Spade (Gedik *et al.*, 2008) are examples of commercial query compilers. Tupleware (Crotty *et al.*, 2015) focuses on support for user-defined functions. DBLAB (Shaikhha *et al.*, 2016) advocates a many-pass compiler design. The first system to use LMS for query compilation was Legobase (Klonatos *et al.*, 2014). More recent systems that are directly based on the conference version of this paper (Rompf & Amin, 2015) are LB2 (Tahboub *et al.*, 2018), LB2-Spatial (Tahboub & Rompf, 2016), and Flare (Essertel *et al.*, 2018), a native compiler back-end for Apache Spark.

8.2 Generative programming

Multistage programming (*staging* for short), as established by Taha & Sheard (2000), enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml (Calcagno *et al.*, 2003) implements a classic staging system based on quasi-quotation. LMS (Rompf & Odersky, 2010; Rompf & Odersky, 2012) uses types instead of syntax to identify binding times and generates an IR instead of target code (Rompf, 2012). LMS draws inspiration from earlier work such as TaskGraph (Beckmann *et al.*, 2003), a C++ framework for program generation and optimization. Delite is a compiler framework for embedded DSLs that provides parallelization and heterogeneous code generation on top of LMS (Rompf *et al.*, 2013; Brown *et al.*, 2011; Rompf *et al.*, 2011; Lee *et al.*, 2011; Ackermann *et al.*, 2012; Sujeeth *et al.*, 2013a, 2013b; Brown *et al.*, 2016).

While this line of work demonstrates that Scala is a good choice as host environment for generative programming (Odersky & Rompf, 2014), other expressive modern languages can be used just as well, as demonstrated by Racket macros (Tobin-Hochstadt *et al.*, 2011); DSLs Accelerate (McDonell *et al.*, 2013), Feldspar (Axelsson *et al.*, 2011), and Nikola (Mainland & Morrisett, 2010) in Haskell; the Copperhead (Catanzaro *et al.*, 2011) system in Python; and Terra (DeVito *et al.*, 2013, 2014) as multistage extension of Lua. Various patterns for realizing program transformations for increased performance through generative programming have been identified by the high-performance-computing community (Cohen *et al.*, 2006; Ofenbeck *et al.*, 2017), but there is comparatively fewer work in the context of systems-oriented software. The essence of LMS has been described as a combination of techniques such as operator overloading and eager let-insertion (Rompf, 2016a, 2016b), which can be realized in different ways in both statically and dynamically typed languages (Amin & Rompf, 2018; Moldovan *et al.*, 2019).

8.3 Partial evaluation

Partial evaluation (Jones *et al.*, 1993) is an automatic program-specialization technique. Some notable systems include DyC (Grant *et al.*, 2000) for C, JSpec and Tempo (Schultz *et al.*, 2003), the JSC Java Supercompiler (Klimov, 2009), Civet (Shali & Cook, 2011), and Lancet (Rompf *et al.*, 2014) for Java. Preserving proper semantics in the presence of state has been an important goal (Bondorf, 1990; Hatcliff & Danvy, 1997; Lawall & Thiemann,

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

1997; Thiemann & Dussart, 1999). Further work has studied partially static structures (Mogensen, 1988) and partially static operations (Thiemann, 2013), and compilation based on combinations of partial evaluation, staging, and abstract interpretation (Sperber & Thiemann, 1996; Consel & Khoo, 1993; Kiselyov *et al.*, 2004).

9 Perspectives

This paper is a case study in 'abstraction without regret': achieving high performance from very high-level code. More generally, we argue for a radical rethinking of the role of high-level languages in performance-critical code (Rompf *et al.*, 2015).

Our case study illustrates a few common generative design patterns: higher-order functions for composition of code fragments, objects and classes for mixed-staged data structures and for modularity at code-generation time. While these patterns have emerged and proven useful in several projects, the field of practical generative programming is still in its infancy and is lacking an established canon of programming techniques. Thus, our plea to language designers and to the wider PL community is to ask, for each language feature or programming model: 'how can it be used to good effect in a generative style?'

Acknowledgments

We thank Grégory Essertel and Ruby Tahboub (Purdue University) for contributing the experiments reported in Section 6 and for scaling up the approach to a realistic SQL engine as summarized in Section 7. Parts of this research were supported by ERC grant 321217, NSF awards 1553471 and 1564207, and DOE award DE-SC0018050.

References

Ackermann, S., Jovanovic, V., Rompf, T. & Odersky, M. (2012) Jet: An embedded DSL for high performance big data processing. In International Workshop on End-to-end Management of Big Data (BigData 2012).

Amin, N. & Rompf, T. (2018) Collapsing towers of interpreters. In *PACMPL*, vol. 2. (POPL). ACM.

Armbrust, M., *et al.* (2015) Spark SQL: Relational data processing in spark. In *SIGMOD*. ACM.

Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J., Griffiths, P. P., III, King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W. & Watson, V. (1976) System R: relational approach to database management. *ACM Trans. Database Syst.* **1**(2), 97–137.

Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D. & Persson, A. (2011) The design and implementation of Feldspar: An embedded language for digital signal processing. In *IFL'10*. Springer.

Beckmann, O., Houghton, A., Mellor, M. R. & Kelly, P. H. J. (2003) Runtime code generation in C++ as a foundation for domain-specific optimisation. *Domain-Specific Program Generation*. Lecture Notes in Computer Science, vol. 3016. Springer.

Boncz, P., Grust, T., Van Keulen, M., Manegold, S., Rittinger, J. & Teubner, J. (2006) MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *SIGMOD*.

Bondorf, A. (1990) *Self-applicable partial evaluation*. Ph.D. thesis, DIKU, Department of Computer Science, University of Copenhagen.

Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M. & Olukotun, K. (2011) A heterogeneous parallel framework for domain-specific languages. In *PACT*.

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

Brown, K. J., Lee, H. J., Rompf, T., Sujeeth, A. K., De Sa, C., Aberger, C. & Olukotun, K. (2016) Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *CGO*. ACM.

Calcagno, C., Taha, W., Huang, L. & Leroy, X. (2003) Implementing multi-stage languages using asts, gensym, and reflection. In *GPCE*. ACM.

Catanzaro, B., Garland, M. & Keutzer, K. (2011) Copperhead: Compiling an embedded data parallel language. In *PPoPP*. ACM.

Chamberlin, D. D., Astrahan, M. M., Blasgen, M. W., Gray, J. N., King, W. F., Lindsay, B. G., Lorie, R., Mehl, J. W., Price, T. G., Putzolu, F., Selinger, P. G., Schkolnick, M., Slutz, D. R., Traiger, I. L., Wade, B. W. & Yost, R. A. (1981) A history and evaluation of System R. *Commun. ACM*, **24**(10).

Chiba, T. & Onodera, T. (2015) *Workload Characterization and Optimization of tpc-h Queries on Apache Spark*. Technical Report RT0968.

Cohen, A., Donadio, S., Garzarán, M. J., Herrmann, C. A., Kiselyov, O. & Padua, D. A. (2006) In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.* **62**(1). Elsevier.

Consel, C. & Danvy, O. (1993) Tutorial notes on partial evaluation. In *POPL*. ACM.

Consel, C. & Khoo, S.-C. (1993) Parameterized partial evaluation. *ACM Trans. Program. Lang. Syst.* **15**(3).

Crotty, A., Galakatos, A., Dursun, K., Kraska, T., Binnig, C., Çetintemel, U. & Zdonik, S. (2015) An architecture for compiling UDF-centric workflows. *PVLDB* **8**(12). ACM.

DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P. & Vitek, J. (2013) Terra: A multi-stage language for high-performance computing. In *PLDI*. ACM.

DeVito, Z., Ritchie, D., Fisher, M., Aiken, A. & Hanrahan, P. (2014) First-class runtime generation of high-performance types using exotypes. In *PLDI*. ACM.

Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N. & Zwilling, M. (2013) Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*. ACM.

Essertel, G. M., Tahboub, R. Y., Decker, J. M., Brown, K. J., Olukotun, K. & Rompf, T. (2018) Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *OSDI*. ACM.

Futamura, Y. (1971) Partial evaluation of computation process—An approach to a compiler-compiler. *Trans. Inst. Electron. Commun. Eng. Jpn* **54-C**(8).

Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S. & Doo, M. (2008) Spade: The system s declarative stream processing engine. In *SIGMOD*. ACM.

Graefe, G. (1994) Volcano - An extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* **6**(1).

Graefe, G. & McKenna, W. J. (1993) The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*. ACM.

Grant, B., Mock, M., Philipose, M., Chambers, C. & Eggers, S. J. (2000) DyC: An expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.* **248**(1–2).

Hatcliff, J. & Danvy, O. (1997) A computational formalization for partial evaluation. *Math. Struct. Comput. Sci.* **7**(5). ACM.

Isard, M., Budiu, M., Yu, Y., Birrell, A. & Fetterly, D. (2007) Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*. ACM.

Jones, N. D., Gomard, C. K. & Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc.

Jonnalagedda, M., Coppey, T., Stucki, S., Rompf, T. & Odersky, M. (2014) Staged parser combinators for efficient data processing. In *OOPSLA*. ACM.

Jørring, U. & Scherlis, W. L. (1986) Compilers and staging transformations. In *POPL*. ACM.

Kiselyov, O., Swadi, K. N. & Taha, W. (2004) A methodology for generating verified combinatorial circuits. In *EMSOFT*. ACM.

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

Klimov, A. V. (2009) A Java supercompiler and its application to verification of cache-coherence protocols. In *Ershov Memorial Conference*. Springer.

Klonatos, Y., Koch, C., Rompf, T. & Chafi, H. (2014) Building efficient query engines in a high-level language. *VLDB* **7**(10). ACM.

Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., Joshi, I., Kuff, L., Kumar, D., Leblang, A., Li, N., Pandis, I., Robinson, H., Rorke, D., Rus, S., Russell, J., Tsirogiannis, D., Wanderman-Milne, S. & Yoder, M. (2015) Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*.

Krikellas, K., Viglas, S. D. & Cintra, M. (2010) Generating code for holistic query evaluation. In *ICDE*. IEEE.

Lawall, J. L. & Thiemann, P. (1997) Sound specialization in the presence of computational effects. *TACS*. Lecture Notes in Computer Science, vol. 1281. Springer.

Lee, H. J., Brown, K. J., Sujeeth, A. K., Chafi, H., Rompf, T., Odersky, M. & Olukotun, K. (2011) Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro* **31**(5).

Mainland, G. & Morrisett, G. (2010) Nikola: Embedding compiled GPU functions in Haskell. In *Haskell Symposium*.

McDonell, T. L., Chakravarty, M. M. T., Keller, G. & Lippmeier, B. (2013) Optimising purely functional GPU programs. In *ICFP*. ACM.

Mehta, M. & DeWitt, D. J. (1995) Managing intra-operator parallelism in parallel database systems. In *SIGMOD*. ACM.

Mogensen, T. A. E. (1988) Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation*, Bjørner, D., Ershov, A. P. & Jones, N. D. (eds).

Moldovan, D., Decker, J. M., Wang, F., Johnson, A. A., Lee, B. K., Nado, Z., S. D., Rompf, T. & Wiltschko, A. B. (2019) AutoGraph: Imperative-style coding with graph-based performance. In *SysML*. Springer.

Neumann, T. (2011) Efficiently compiling efficient query plans for modern hardware. *PVLDB* **4**(9).

Odersky, M. & Rompf, T. (2014) Unifying functional and object-oriented programming with scala. *Commun. ACM* **57**(4).

Ofenbeck, G., Rompf, T. & Püschel, M. (2017) Staging for generic programming in space and time. In *GPCE*. ACM.

Rao, J., Pirahesh, H., Mohan, C. & Lohman, G. (2006) Compiled query execution engine using JVM. In *ICDE*. IEEE.

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72.

Reynolds, J. C. (1998) Definitional interpreters for higher-order programming languages. *Higher-order Symb. Comput.* **11**(4). ACM.

Rompf, T. (2012) *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. Ph.D. thesis, EPFL.

Rompf, T. (2016a) The essence of multi-stage evaluation in LMS. *A List of Successes That Can Change the World*. Lecture Notes in Computer Science, vol. 9600. Springer.

Rompf, T. (2016b) Reflections on LMS: Exploring front-end alternatives. In *Scala Symposium*.

Rompf, T. & Amin, N. (2015) Functional pearl: A SQL to C compiler in 500 lines of code. In *ICFP*. ACM.

Rompf, T. & Odersky, M. (2010) Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM.

Rompf, T. & Odersky, M. (2012) Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* **55**(6).

Rompf, T. Sujeeth, A. K., Lee, H. J., Brown, K. J., Chafi, H., Odersky, M. & Olukotun, K. (2011) Building-blocks for performance oriented DSLs. In *IFIP Working Conference on Domain-Specific Languages (DSL)*. EPTCS, vol. 66. Open Publishing Association.

Rompf, T., Amin, N., Moors, A., Haller, P. & Odersky, M. (2012) Scala-Virtualized: Linguistic reuse for deep embeddings. *Higher-order Symb. Comput.* **25**(1). ACM.

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press

Rompf, T., Sujeeth, A. K., Amin, N., Brown, K., Jovanovic, V., Lee, H. J., Jonnalagedda, M., Olukotun, K. & Odersky, M. (2013) Optimizing data structures in high-level programs. In *POPL*. ACM.

Rompf, T., Sujeeth, A. K., Brown, K. J., Lee, H. J., Chafi, H. & Olukotun, K. (2014) Surgical precision JIT compilers. In *PLDI*. ACM.

Rompf, T., Brown, K. J., Lee, H. J., Sujeeth, A. K., Jonnalagedda, M., Amin, N., Ofenbeck, G., Stojanov, A., Klonatos, Y., Dashti, M., Koch, C., Püschel, M. & Olukotun, K. (2015) Go meta! A case for generative programming and DSLs in performance critical systems. In *SNAPL*.

Schultz, U. P., Lawall, J. L. & Consel, C. (2003) Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.* **25**(4).

Shaikhha, A., Klonatos, I., Parreaux, L. E. V., Brown, L., Dashti Rahmat Abadi, M. & Koch, C. (2016) How to architect a query compiler. In *SIGMOD*. ACM.

Shali, A. & Cook, W. R. (2011) Hybrid partial evaluation. In *OOPSLA*. ACM.

Sperber, M. & Thiemann, P. (1996) Realistic compilation by partial evaluation. In *PLDI*. ACM.

Stonebraker, M. & Çetintemel, U. (2005) “One Size Fits All": An idea whose time has come and gone (abstract). In *ICDE*. IEEE.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N. & Helland, P. (2007) The end of an architectural era (it’s time for a complete rewrite). In *PVLDB*. ACM.

Sujeeth, A. K., Rompf, T., Brown, K. J., Lee, H. J., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M. & Olukotun, K. (2013a) Composition and reuse with compiled domain-specific languages. In *ECOOP*. Springer.

Sujeeth, A. K., Gibbons, A., Brown, K. J., Lee, H. J., Rompf, T., Odersky, M. & Olukotun, K. (2013b) Forge: Generating a high performance DSL implementation from a declarative specification. In *GPCE*. ACM.

Svenningsson, J. & Axelsson, E. (2012) Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming (TFP)*.

Taha, W. & Sheard, T. (2000) MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* **248**(1–2). Elsevier.

Tahboub, R. Y. & Rompf, T. (2016) On supporting compilation in spatial query engines (vision paper). In *SIGSPATIAL*.

Tahboub, R. Y., Essertel, G. M. & Rompf, T. (2018) How to architect a query compiler, revisited. In *SIGMOD*. ACM.

The Transaction Processing Council. (2002) *TPC-H Revision 2*.

Thiemann, P. (2013) Partially static operations. In *PEPM*. ACM.

Thiemann, P. & Dussart, D. (1999) *Partial evaluation for higher-order languages with state*. Technical Report. Germany: Universität Tübingen.

Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M. & Felleisen, M. (2011) Languages as libraries. In *PLDI*. ACM.

Zukowski, M., Boncz, P. A., Nes, N. & Héman, S. (2005) MonetDB/X100 - a DBMS in the CPU cache. *IEEE Data Eng. Bull.* **28**(2).

https://doi.org/10.1017/S0956796819000054 Published online by Cambridge University Press