Building Efficient Query Engines in a High-Level Language

Amir Shaikhha, École Polytechnique Fédérale de Lausanne
Yannis Klonatos, École Polytechnique Fédérale de Lausanne
Christoph Koch, École Polytechnique Fédérale de Lausanne

Abstraction without regret refers to the vision of using high-level programming languages for systems development without experiencing a negative impact on performance. A database system designed according to this vision offers both increased productivity and high performance, instead of sacrificing the former for the latter as is the case with existing, monolithic implementations that are hard to maintain and extend.

In this article, we realize this vision in the domain of analytical query processing. We present LegoBase, a query engine written in the *high-level* programming language Scala. The key technique to regain efficiency is to apply *generative* programming: LegoBase performs source-to-source compilation and optimizes the *entire* query engine by converting the high-level Scala code to specialized, low-level C code. We show how generative programming allows to *easily* implement a wide spectrum of optimizations, such as introducing data partitioning or switching from a row to a column data layout, which are difficult to achieve with existing low-level query compilers that handle *only* queries. We demonstrate that sufficiently powerful abstractions are essential for dealing with the complexity of the optimization effort, shielding developers from compiler internals and decoupling individual optimizations from each other.

We evaluate our approach with the TPC-H benchmark and show that: (a) With all optimizations enabled, our architecture significantly outperforms a commercial in-memory database as well as an existing query compiler. (b) Programmers need to provide just a few hundred lines of high-level code for implementing the optimizations, instead of complicated low-level code that is required by existing query compilation approaches. (c) These optimizations may potentially come at the cost of using more system memory for improved performance. (d) The compilation overhead is low compared to the overall execution time, thus making our approach usable in practice for compiling query engines.

1. INTRODUCTION

During the last decade, we have witnessed a shift towards the use of high-level programming languages for systems development. Examples include the Singularity Operating System [Hunt and Larus 2007], the Spark [Zaharia et al. 2010] and DryadLINQ [Yu et al. 2008] frameworks for efficient, distributed data processing, the FiST platform for specifying stackable file systems [Zadok et al. 2006] and GPUs programming [Holk et al. 2013]. All these approaches collide with the traditional wisdom which calls for using low-level languages like C for building high-performance systems.

This shift is necessary as the productivity of developers is severely diminished in the presence of complicated, monolithic, *low-level* code bases, making their debugging and maintenance very costly. High-level programming languages can remedy this situation in two ways. First, by offering advanced software features (modules, interfaces, object orientation, etc.), they allow the same functionality to be implemented with significantly less code (compared to low-level languages). Second, by providing powerful type systems and well-defined design patterns, they allow programmers not only to create abstractions and protect them from leaking but also to quickly define system modules that are *reusable* (even in contexts very different from the one these were created for) and easily composable [Odersky and Zenger 2005]. All these properties can reduce the number of software errors of the systems and facilitate their verification.

Yet, despite these benefits, database systems are still written using low-level languages.

This work was supported by ERC grant 279804 and NCCR MARVEL of the Swiss National Science Foundation.

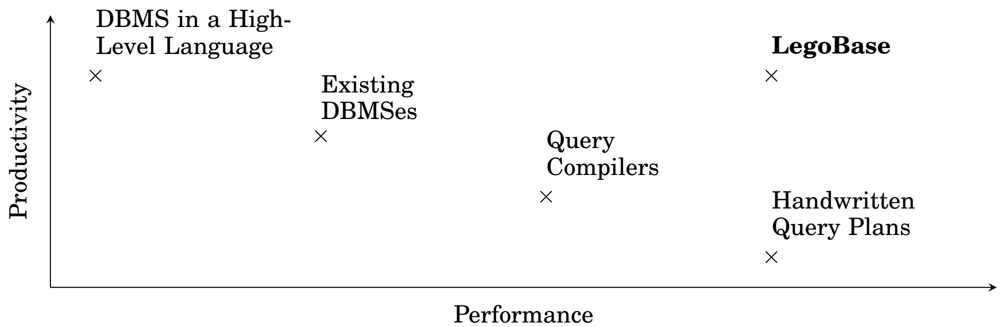

Fig. 1: Comparison of the performance/productivity trade-off for all approaches presented in this article.

The reason is that increased productivity comes at a cost: high-level languages increase indirection, which in turn has a pronounced negative impact on performance. For example, abstraction generally necessitates the need of containers, leading to costly object creation and destruction operations at runtime. Encapsulation is provided through object copying rather than object referencing, thus similarly introducing a number of expensive memory allocations on the critical path. Even primitive types such as integers are often converted to their object counterparts for use with general-purposes libraries. As a result of these overheads, the use of high-level languages for developing high-performance databases seems (deceptively) prohibited.

The *abstraction without regret* vision [Koch 2013; 2014] argues that it is indeed possible to use high-level languages for building database systems that allow for *both* productivity and high performance, instead of trading off the former for the latter. By programming databases in a high-level style and still being able to get good performance, the time saved can be spent implementing more database features and optimizations. In addition, the language features of high-level languages can grant great flexibility to developers so that they can easily experiment with various design choices.

In this article, we realize the abstraction without regret vision on the domain of ad-hoc, analytical query processing. We make the following contributions:

— We present LegoBase, an in-memory query execution engine written in the high-level programming language, Scala, being the first step towards providing a full DBMS written in a high-level language.

  To avoid the overheads of a high-level language (e.g. complicated memory management) while maintaining well-defined abstractions, we opt for using *generative* programming [Taha and Sheard 2000], a technique that allows for programmatic removal of abstraction overhead through source-to-source compilation. This is a key benefit as, in contrast to traditional, general-purpose compilers – which need to perform complicated and sometimes brittle analyses before *maybe* optimizing programs – generative programming in Scala takes advantage of the type system of the language to provide programmers with strong *guarantees* about the structure of the generated code. For example, developers can specify optimizations that are applied during compilation in order to ensure that certain abstractions (e.g. generic data structures and function calls) are definitely optimized away during compilation.

  Generative programming can be used to optimize *any* piece of Scala code. This allows LegoBase to perform *whole-system* specialization and compile *all* components, data structures and auxiliary functions used inside the query engine to efficient

C code. This design significantly contrasts our approach with existing *query* compilation approaches (e.g. the one proposed in [Neumann 2011]) for three reasons. First, a compiler that handles *only* queries cannot optimize and inline their code with the remaining code of the database system (which is typically *precompiled*), thus missing a number of optimization opportunities. Second, in their purest form, query compilation approaches simply optimize or inline the code of *individual* operators in the physical query plan, thus making cross-operator code optimization inside the query compiler impossible. Finally, existing approaches perform compilation using low-level code generation templates. These essentially come in stringified form, making their development and automatic type checking very difficult[1].

— The LegoBase query engine uses a *new* optimizing compiler called SC. When performing *whole-system* compilation, an optimizing compiler effectively needs to specialize *high-level* systems code which will naturally employ a hierarchy of components and libraries from relatively high to very low level of abstraction. To scale to such complex code bases, an optimizing compiler must guarantee two properties, not offered by existing compiler frameworks for applying generative programming.

First, to achieve maximum efficiency, developers must have tight control on the compiler's phases – admitting custom optimization phases and phase orderings. This is necessary as code transformers with different optimization objectives may have to be combined in every possible ordering, depending on architectural, data, or query characteristics. However, existing generative programming frameworks do not offer much control over the compilation process[2]. This absence of control effectively forces developers to provision for *all* possible optimization orderings. This pollutes the code base of individual optimizations, making some of them dependent on other, possibly semantically independent, optimizations. In general, the code complexity grows exponentially with the number of supported transformations[3].

Second, existing optimizing compilers expose a large number of low-level, compiler internals such as nodes of an intermediate representation (IR), dependency information encoded in IR nodes, and code generation templates to their users. This interaction with low-level semantics when coding optimizations, but also the introduction of the IR as an additional level of abstraction, both significantly increase the difficulty of debugging as developers cannot easily track the relationship between the source code, the optimization for it – expressed using IR constructs – and the final, generated code [Jovanović et al. 2014; Sujeeth et al. 2013].

Instead, the SC compiler was designed from the beginning so that it allows developers to have full control over the optimization process without exporting compiler internals such as code generation templates. It does so by delivering sufficiently powerful programming abstractions to developers like those afforded by modern

[1]For example, templates can be used to convert the code of individual query operators – typically written today in C/C++ – to optimized LLVM code. In that case, developers must handle a number of low-level concerns themselves, like register allocation.

[2]For instance, Lightweight Modular Staging (LMS) [Rompf and Odersky 2010] applies *all* user-specified, domain-specific optimizations in a *single* optimization step. It does so to avoid the well-known *phase-ordering* problem in compilers, where applying two (or more) optimizations in an improper order can lead not only to suboptimal performance but also to programs that are semantically incorrect [Rompf 2012]. We analyze how the design of the new optimizing compiler, SC, differs from that of LMS in Section 2 of this article.

[3]As an example, consider the case of a compiler that is to support only two optimizations: 1) data-layout optimizations (i.e. converting a row layout to a column or PAX-like layout [Ailamaki et al. 2001]) and 2) data-structure specialization (i.e. adapting the definition of a data structure to the particular context in which it is used). This means that if the second optimization handles three different types of specialization, one has to provision for $2 \times 3 = 6$ cases to handle all possible combinations of these optimizations.

```
SELECT * FROM
  (SELECT S.D,
    SUM(1-S.B) AS E,
    SUM(S.A*(1-S.B)),
    SUM(S.A*(1-S.B)*(1+S.C))
   FROM S
   GROUP BY S.D) T, R
WHERE T.E=R.Z AND R.Q=3
```

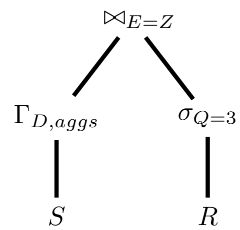

Fig. 2: Motivating example showing missed optimizations opportunities by existing query compilers that use template expansion.

high-level programming languages. The SC compiler along with all optimizations are both written in plain Scala, thus allowing developers to be highly productive when optimizing all components of the query engine.

— We demonstrate the ease of use of the new SC compiler for optimizing system components that differ significantly in structure and granularity of operations. We do so by providing (i) an in-depth presentation of the optimizations applied to the LegoBase query engine and (b) a description of the *high-level* compiler interfaces that database developers need to interact with when coding optimizations.

  We show that the design and interfaces of our optimizing compiler provide a number of nice properties for the LegoBase optimizations. These are expressed as library components, providing a *clean* separation from the base code of LegoBase (e.g. that of query operators), but also from each other. This is achieved, (as explained later in more detail in Section 2) by applying them in multiple, *distinct* optimization phases. Optimizations are (a) adjustable to the characteristics of workloads and architectures, (b) configurable, so that they can be turned on and off on demand and (c) composable, so that they can be easily chained but also so that higher-level optimizations can be built from lower-level ones.

  For each such optimization, we present: (a) the *domain-specific* conditions that need to be satisfied in order to apply it (if any) and (b) possible trade-offs (e.g. improved execution time versus increased memory consumption). Finally, we examine which categories of database systems can benefit from applying each of our optimizations by providing a classification of the LegoBase optimizations.

— We perform an experimental evaluation in the domain of analytical query processing using the TPC-H benchmark [Transaction Processing Performance Council 1999]. We show how our optimizations can lead to a system that has performance competitive to that of a standard, commercial in-memory database called DBX (that does not employ compilation) and the code generated by the query compiler of the HyPer database [Neumann 2011]. In addition, we illustrate that these performance improvements do not require significant programming effort as even complicated optimizations can be coded in LegoBase with only a few hundred lines of code. We also provide insights on the performance characteristics and trade-offs of individual optimizations. We do so by comparing major architectural decisions as fairly as possible, using a shared codebase that only differs by the effect of a single optimization. Finally, we conclude our analysis by demonstrating that our *whole-system* compilation approach incurs negligible overhead to query execution.

Motivating Example. To better understand the differences of our work with previous approaches, consider the SQL query shown in Figure 2. This query first calculates some aggregations from relation S in the group by operator Γ. Then, it joins these aggregations with relation R, the tuples of which are filtered by the value of column Q. The results are then returned to the user. Careful examination of the execution plan of this query, shown in the same figure, reveals the following three basic optimization opportunities missed by existing query compilers that use template expansion:

— First, the limited scope of existing approaches usually results in performing the evaluation of aggregations in *precompiled* DBMS code. Thus, each aggregation is evaluated *consecutively* and, as a result, common sub-expression elimination cannot be performed in this case (e.g. in the calculation of expressions `1-S.B` and `S.A*(1-S.B)`). This shows that, if we include the evaluation of all aggregations in the *compiled* final code, we can get an additional performance improvement. This motivates us to extend the scope of compilation in this work.

— Second, template-based approaches may result in unnecessary computation. This is because operators are not aware of each other. In this example, the generated code includes two materialization points: (a) at the group by and (b) when materializing the left side of the join. However, there is no need to materialize the tuples of the aggregation in two different data structures as the aggregations can be immediately materialized in the data structure of the join. Such *inter-operator* optimizations are hard to express using *template-based* compilers. By high-level programming, we can instead easily pattern match on the operators, as we show in Section 3.1.

— Finally, the data structures have to be *generic* enough for all queries. As such, they incur significant abstraction overhead, especially when these structures are accessed millions of times during query evaluation. Current query compilers cannot optimize the data structures since these belong to the precompiled part of the DBMS. Our approach eliminates these overheads as it performs *whole-program* optimization and compiles, along with the operators, the data structures employed by a query. This significantly contrasts our approach with previous work.

The rest of this article is organized as follows. Section 2 presents the overall design of LegoBase, along with a detailed description of the APIs provided by the new SC optimizing compiler. Section 3 gives an in-depth presentation of all supported compiler optimizations of our system in multiple domains. Section 4 presents our evaluation, where we experimentally show that our approach using the SC optimizing compiler can lead to significant benefits compared to (i) a commercial DBMS that does not employ compilation and (ii) a database system that uses low-level, code-generation templates during query compilation. We also give insights about the memory footprint, data loading time and programming effort required when working with the LegoBase system. Section 5 presents related work in the area of compilation and compares our approach with existing query compilers and engines. Finally, Section 6 concludes.

2. SYSTEM DESIGN

In this section, we present the design of the LegoBase system. First, we describe the overall system architecture of our approach (Subsection 2.1). Then, we describe in detail the SC compiler that is the core of our proposal (Subsection 2.2) as well as how we efficiently convert the *entire* high-level Scala code of the query engine (not just that of individual operators) to optimized C code for each incoming query (Subsection 2.3). While doing so, we give concrete code examples of how (a) physical query operators, (b) physical query plans, and, (c) compiler interfaces look like in our system.

2.1. Overall System Architecture

LegoBase implements the typical query plan operators found in traditional database systems, including equi, semi, anti, and outer joins, all on a high level. In addition,

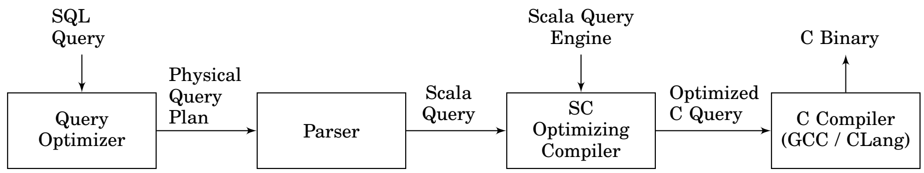

Fig. 3: Overall system architecture. The domain-specific optimizations of LegoBase are applied during the SC compiler optimization phase.

LegoBase supports both a classical Volcano-style [Graefe 1994] query engine as well as a push-style query interface [Neumann 2011][4].

The overall system architecture of LegoBase is shown in Figure 3. First, for each incoming SQL query, we must get a query plan which describes the physical query operators needed to process this query. For this work, we consider traditional query optimization (e.g. determining join ordering) as an orthogonal problem and we instead focus more on experimenting with the different optimizations that can be applied *after* traditional query optimization. Thus, to obtain a physical query plan, we pass the incoming query through *any existing* query optimizer. For example, for our evaluation, we choose the query optimizer of a commercial, in-memory database system.

Then, we pass the generated physical plan to LegoBase. Our system, in turn, parses this plan and instantiates the corresponding Scala implementation of the operators. Figure 4 presents an example of how query plans and operators are written in LegoBase, respectively. That is, the Scala code example shown in Figure 4a loads the data, builds a functional tree from operator objects and then starts executing the query by calling `next` for the root operator.

It is important to note that operator implementations like the one presented in Figure 4b are exactly what one would write for a simple query engine that does not involve compilation at all. However, without further optimizations, this engine cannot match the performance of existing databases: it consists of generic data structures (e.g. the one declared in line 4 of Figure 4b) and involves expensive memory allocations on the critical path[5], both properties that can significantly affect performance.

However, in our system, the SC optimizing compiler specializes the code of the *entire* query engine on the fly (including the code of individual operators, all data structures used as well as any required auxiliary functions), and progressively optimizes the code using our domain-specific optimizations (described in detail in Section 3). For example, it optimizes away the `HashMap` abstraction and transforms it to efficient low-level constructs (Section 3.2). In addition, SC utilizes the available *query-specific* information during compilation. For instance, it will inline the code of all individual operators and, for the example of Figure 4b, it automatically unrolls the loop of lines 8-11, since the number of aggregations can be statically determined based on how many aggregations the input SQL query has. Such fine-grained optimizations have a significant effect on performance, as they improve branch prediction. Finally, our system generates the

[4]In a push engine, the meaning of child and parent operators is reversed compared to the usual query plan terminology: Data flows from the leaves (the ancestors, usually being scan operators) to the root (the final descendant, which computes the final query results that are returned to the user).

[5]Note that such memory allocations are not always explicit (i.e. at object definition time through the **`new`** keyword in object-oriented languages like Java and Scala). For instance, in line 15 of Figure 4b, the HashMap data structure may have to expand (in terms of allocated memory footprint) and be reorganized by the Scala runtime in order to more efficiently store data for future lookup operations. We talk more about this issue and its consequences to performance later in this article.

```
def Q6() {
  val lineitemTable = loadLineitem()
  val scanOp = new ScanOp(lineitemTable)
  val startDate = parseDate("1996-01-01")
  val endDate = parseDate("1997-01-01")
  val selectOp = new SelectOp(scanOp)
    (x =>
      x.L_SHIPDATE >= startDate &&
      x.L_SHIPDATE < endDate &&
      x.L_DISCOUNT >= 0.08 &&
      x.L_DISCOUNT <= 0.1 &&
      x.L_QUANTITY < 24
    )
  val aggOp = new AggOp(selectOp)
    (x => "Total")
    ((t, agg) => { agg +
      (t.L_EXTENDEDPRICE * t.L_DISCOUNT)
    })
  val printOp = new PrintOp(aggOp)(
    kv => printf("%.4f\n", kv.aggs(0))
  )
  printOp.open
  printOp.next
}
```

(a)

```
class AggOp[B](child:Operator, grp:Record=>B,
              aggFuncs:(Record,Double)=>Double*)
  extends Operator {
    val hm = HashMap[B, Array[Double]]()
    def open() { parent.open }
    def process(aggs:Array[Double], t:Record){
      var i = 0
      aggFuncs.foreach { aggFun =>
        aggs(i) = aggFun(tuple, aggs(i))
        i += 1
      }
    }
    def consume(tuple:Record) {
      val key = grp(tuple)
      val aggs = hm.getOrElseUpdate(key,
        new Array[Double](aggFuncs.size))
      process(aggs, tuple)
    }
    def next() : Record = {
      hm.foreach { pair => child.consume(
        new AGGRecord(pair._1, pair._2)
      ) }
    }
}
```

(b)

Fig. 4: Example of a query plan and an operator implementation in LegoBase. The SQL query used as an input here is actually Query 6 of the TPC-H workload. The operator implementation presented here uses the Push-style interface [Neumann 2011].

optimized C code[6], which is compiled using any existing C compiler (e.g. we use the CLang[7] frontend of LLVM [Lattner and Adve 2004] for compiling the generated C code in our evaluation). We then return the query results to the user.

2.2. The SC Compiler Framework

LegoBase makes key use of the SC framework, which provides runtime compilation and code generation facilities for the Scala programming language, as follows.

To begin with, in contrast to low-level compilation frameworks like LLVM – which express optimizations using a low-level, compiler-internal intermediate representation (IR) that operates on the level of registers and basic blocks – programmers in SC specify the result of a program transformation as a high-level, *compiler-agnostic* Scala program. SC offers two *high-level* programming primitives named `analyze` and `rewrite` for this purpose, which are illustrated in Figure 5a and which analyze and manipulate statements and expressions of the input program, respectively. For example, our data-structure specialization (Section 3.2.2) replaces operations on hash maps with operations on native arrays. By expressing optimizations at a high level, our approach enables a user-friendly way to describe these domain-specific optimizations that humans can easily identify, without imposing the need to interact with compiler

[6]In this work, we choose C as our code-generation language as this is the language traditionally used for building high-performance database systems. However, SC is not particularly aware of C and can be used to generate programs in other languages as well (e.g. optimized Scala).
[7]http://clang.llvm.org/

```
analysis += statement {
  case sym -> code"new MultiMap[_, $v]"
    if isRecord(v) => allMaps += sym
}
analysis += rule {
  case loop @ code"while($cond) $body" =>
    currentWhileLoop = loop
}

rewrite += statement {
  case sym -> (code"new MultiMap[_, _]")
    if allMaps.contains(sym) =>
      createPartitionedArray(sym)
}
rewrite += remove {
  case code"($map: MultiMap[Any, Any])
    .addBinding($elem, $value)"
      if allMaps.contains(map) =>
}
rewrite += rule {
  case code"($map: MultiMap[Any, Any])
    .addBinding($elem, $value)"
      if allMaps.contains(map) =>
      /* Code for processing add Binding */
}
```

(a)

```
pipeline += OperatorInlining
pipeline += SingletonHashMapToValue
pipeline += ConstantSizeArrayToValue
pipeline += ParamPromDCEAndPartiallyEvaluate
if (settings.partitioning) {
  pipeline += PartitioningAndDateIndices
  pipeline += ParamPromDCEAndPartiallyEvaluate
}
if (settings.hashMapLowering)
  pipeline += HashMapLowering
if (settings.stringDictionary)
  pipeline += StringDictionary
if (settings.columnStore) {
  pipeline += ColumnStore
  pipeline += ParamPromDCEAndPartiallyEvaluate
}
if (settings.dsCodeMotion) {
  pipeline += HashMapHoisting
  pipeline += MallocHoisting
  pipeline += ParamPromDCEAndPartiallyEvaluate
}
if (settings.targetIsC)
  pipeline += ScalaToCLowering
// else: handle other languages, e.g. Scala
pipeline += ParamPromDCEAndPartiallyEvaluate
```

(b)

Fig. 5: (a) The analysis and transformation APIs provided by SC. (b) The SC transformation pipeline used by LegoBase. Details for the optimizations listed in this pipeline are presented in Section 3.

internals[8]. We use this optimization interface to provide database-specific optimizations as a library and to aggressively optimize our query engine.

Then, to allow for maximum efficiency when specializing all components of the query engine, developers must be able to easily experiment with different optimizations and optimization orderings (depending on the characteristics of the input query or the properties of the underlying architecture). In SC, developers do so by explicitly specifying a *transformation pipeline*. This is a straightforward task as SC transformers act as black boxes, which can be plugged in at any stage in the pipeline. For instance, for the transformation pipeline of LegoBase, shown in Figure 5b, Parameter Promotion, Dead Code Elimination and Partial Evaluation are all applied at the end of each of the custom, domain-specific optimizations. Through this transformation pipeline, developers can easily turn optimizations on and off at demand (e.g. by making their application dependant on simple runtime or configuration conditions) as well as specifying which optimizations should be applied *only* for specific hardware platforms.

Even though it has been advocated in previous work [Rompf et al. 2013] that having multiple transformers can cause phase-ordering problems, our experience is that system developers are empowered by the control they have when coding optimizations with SC and rise to the challenge of specifying a suitable order of transformations as

[8]Of course, every compiler needs to represent code through an intermediate representation. The difference between SC and other optimizing compilers is that the IR of our compiler is completely hidden from developers: both the input source code and all of its optimizations are written in plain Scala code, which is then translated to an internal IR through Yin-Yang [Jovanović et al. 2014].

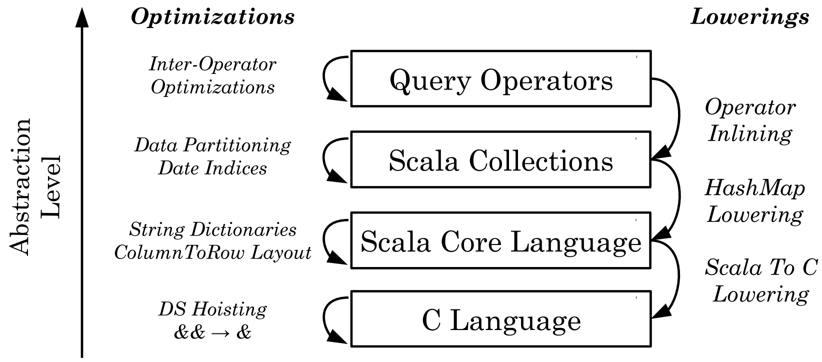

Fig. 6: Source-to-source compilation expressed through the progressive lowering approach – there different optimizations are applied in different optimization stages, thus guaranteeing the notion of separation of concerns.

they design their system and its compiler optimizations. As we show in Section 4, with a relatively small number of transformations we can get a significant performance improvement in LegoBase.

SC already provides many generic compiler optimizations like function inlining, common subexpression and dead code elimination, constant propagation, scalar replacement, partial evaluation, and code motion. In this work, we extend this set to include DBMS-specific optimizations (e.g. using the popular columnar layout for data processing). We describe these optimizations in more detail in Section 3.

2.3. Efficiently Compiling High-Level Query Engines

Database systems comprise many components of significantly different nature and functionality, thus typically resulting in very big code bases. To efficiently optimize those, developers must be able to express new optimizations without the having to modify neither (i) the base code of the system nor (ii) previously developed optimizations. As discussed previously, compilation techniques based on template expansion do not scale to the task, as their single-pass approach forces developers to deal with a number of low-level concerns, making their debugging and development costly.

To this end, the SC compiler framework is built around the principle that, instead of using template expansion to directly generate low-level code from a high-level program in a *single* macro expansion step, an optimizing compiler should instead **progressively lower the level of abstraction** until we reach the lowest possible level of representation, and only then generating the final, low-level code. This design is illustrated in Figure 6.

Each level of abstraction and all associated optimizations operating in it can be seen as independent modules, enforcing the principle of *separation of concerns*. Higher levels are generally more declarative, thus allowing for increased productivity, while lower levels are closer to the underlying architecture, thus making it possible to more easily perform low-level performance tuning. For example, optimizations such as join reordering are only feasible in higher abstraction levels (where the operator objects are still present in the code), while register allocation decisions can only be expressed in very low abstraction levels. This design provides the nice property that generation of the final code basically becomes a trivial and naive stringification of the lowest level representation. Table I provides a brief summary of the benefits of imperative and declarative languages in general.

Paradigm	Advantages
Declarative	✓ Concise programs ✓ Simple to analyze and verify ✓ Simple to parallelize
Imperative	✓ Efficient data structures ✓ Precise control of execution flow ✓ More predictable performance

Table I: Comparison of declarative and imperative language characteristics. We use both paradigms for different steps of our progressive lowering compilation approach.

More precisely, in order to reach the abstraction level of C code in LegoBase (the lowest level representation for the purposes of this article), transformations in SC also include multiple *lowering* steps that *progressively* map Scala constructs to (a set) of C constructs. Most Scala abstractions (e.g. objects, classes, inheritance) are optimized away in one of these intermediate stages (for example, hash maps are converted to arrays through the domain-specific optimizations described in more detail in Section 3), and for the remaining constructs (e.g. loops, variables, arrays) there exists a one-to-one correspondence between Scala and C. SC already offers such lowering transformers for an important subset of the Scala programming language. For example, classes are converted to structs, strings to arrays of bytes, etc. In general, composite types are handled in a recursive way, by first lowering their fields and then wrapping the result in a C struct. The final result is a struct of only primitive C constructs.

This way of lowering does not require any modifications to the database code or effort from database developers other than just specifying in SC how and after which abstraction level custom data types and abstractions should be lowered. More importantly, such a design allows developers to create new abstractions in one of their optimizations, which can be in turn optimized away in subsequent optimization passes. After all lowering steps have been performed, developers can now apply low-level, architecture-dependent optimizations, as the code is now close to the semantics offered by low-level programming languages (e.g. includes pointers for explicitly referencing memory). Then, a final iteration emits the actual C code.

Finally, there are two additional implementation details of our source-to-source compilation from Scala to C that require special mentioning.

First, the final code produced by LegoBase, with all optimizations enabled, does not require library function calls. For example, all collection data structures like hash maps are converted in LegoBase to primitive arrays (Section 3.2). Thus, lowering such library calls to C is not a big issue. However, we view LegoBase as a platform for easy experimentation of database optimizations. As a result, our architecture must also be able to support traditional collections as a library and convert, whenever necessary, Scala collections to corresponding ones in C. We have found GLib [The GNOME Project 2013] to be efficient enough for this purpose.

Second, and more importantly, the two languages handle memory management in a totally different way: Scala is garbage collected, while C has explicit memory management. Thus, when performing source-to-source compilation from Scala to C, we must take special care to free the memory that would normally be garbage collected in Scala in order to avoid memory overflow. This is a hard problem to solve automatically, as garbage collection may have to occur for objects allocated outside the DBMS code, e.g. for objects allocated inside the Scala libraries. For the scope of this work, we follow a conservative approach and make, whenever needed, allocations and deallocations explicit in the Scala code. We also free the allocated memory after each query execution.

```
SELECT COUNT(*)
FROM R, S
WHERE R.name == "R1"
 AND R.id == S.id
```

(a) The example query in SQL.

```
AggOp(HashJoinOp(
 SelectOp(ScanOp(R), r => r.name == "R1"),
 ScanOp(S), (r,s) => r.id == s.id
), (rec, count) => count + 1)
```

(b) The physical plan of the example query.

```
val hm = new MultiMap[Int,R]

for(r <- R) {
 if(r.name == "R1") {


  hm.addBinding(r.id, r)




 }
}
var count = 0
for(s <- S) {
  hm.get(s.id) match {
    case Some(rList) =>
      for(r <- rList) {
        if(r.id == s.id)
          count += 1
      }
    case None => ()
  }
}
return count
```

(c)

```
val MR: Array[Seq[R]] =
 new Array[Seq[R]](BUCKETSZ)
for(r <- R) {
 if(r.name == "R1") {


  MR(r.id) += r




 }
}
var count = 0
for(s <- S) {

 val rList = MR(s.id)
 for(r <- rList) {
  if(r.id == s.id)
   count += 1

 }
}

return count
```

(d)

```
val MR: Array[R] =
 new Array[R](BUCKETSZ)
for(r <- R) {
 if(r.name == "R1") {
  if(MR(r.id) == null) {
   MR(r.id) = r
  } else {
   r.next = MR(r.id)
   MR(r.id) = r
  }
 }
}
var count = 0
for(s <- S) {

 var r: R = MR(s.id)
 while(r != null) {
  if(r.id == s.id)
   count += 1
  r = r.next
 }
}

return count
```

(e)

```
val MR: Array[Pointer[R]] =
 malloc[Pointer[R]](BUCKETSZ)
for(r <- R) {

 if(r->name == "R1") {
  if(MR(r->id) == null) MR(r->id) = r
  else {
   r->next = MR(r->id)
   MR(r->id) = r
  }
 }
}
var count = 0
for(s <- S) {

 var r: Pointer[R] = MR(s->id)
 while(r != null) {
  if(r->id == s->id)
   count += 1
  r = r->next
 }
}
return count
```

(f)

```
R** MR = (R**)
 malloc(BUCKETSZ * sizeof(R*))
for(int i=0; i < R_REL_SIZE; i++) {
 R* r = R[i];
 if(strcmp(r->name, "R1") == 0) {
  if(MR[r->id] == NULL) MR[r->id] = r;
  else {
   r->next = MR[r->id];
   MR[r->id] = r;
  }
 }
}
int count = 0
for(int i=0; i < S_REL_SIZE; i++) {
 S* s = S[i];
 R* r = MR[s->id]
 while(r != NULL) {
  if(r->id == s->id)
   count += 1;
  r = r->next;
 }
}
return count;
```

(g)

Fig. 7: Progressively lowering an example query to C code with SC.

```
def Q12() {
  val ordersScan = new ScanOp(loadOrders())
  val lineitemScan = new ScanOp(loadLineitem())
  val lineitemSelect = new SelectOp(lineitemScan)(record =>
    record.L_RECEIPTDATE >= parseDate("1994-01-01") &&
    record.L_RECEIPTDATE < parseDate("1995-01-01") &&
   (record.L_SHIPMODE == "MAIL" || record.L_SHIPMODE == "SHIP") &&
    record.L_SHIPDATE < record.L_COMMITDATE && record.L_COMMITDATE < record.L_RECEIPTDATE
  )
  val jo = new HashJoinOp(ordersScan, lineitemSelect) // Join Predicate and Hash Functions
    ((ordersRec,lineitemRec) => ordersRec.O_ORDERKEY == lineitemRec.L_ORDERKEY)
    (ordersRec => ordersRec.O_ORDERKEY)(lineitemRec => lineitemRec.L_ORDERKEY)
  val aggOp = new AggOp(jo)(t => t.L_SHIPMODE) // L-SHIPMODE is the Aggregation Key
    ((t, agg) => {
      if (t.O_ORDERPRIORITY == "1-URGENT" || t.O_ORDERPRIORITY == "2-HIGH") agg + 1 else agg
    },
    (t, agg) => {
      if (t.O_ORDERPRIORITY != "1-URGENT" && t.O_ORDERPRIORITY != "2-HIGH") agg + 1 else agg
    })
  val sortOp = new SortOp(aggOp)((x, y) => x.key - y.key)
  val po = new PrintOp(sortOp)(kv => {
    printf("%s|%.0f|%.0f\n", kv.key, kv.aggs(0), kv.aggs(1))
  })
  po.open
  po.next
}
```

Fig. 8: Example of an input query plan (TPC-H Q12). We use this query to explain various characteristics of our domain-specific optimizations in Section 3.

2.4. Putting it all together – A compilation example

To better illustrate the various steps of our progressive lowering, we analyze how LegoBase converts the example SQL query shown in Figure 7a to efficient C code.

To begin with, the query plan, shown in Figure 7b, is parsed and converted to the program shown in Figure 7c. This step inlines the code of all relational operators present in the query plan and implements the equijoin using a hash table. This is the natural way database developers would typically implement a join operator using high-level collections programming. Then, this hash-table data structure is lowered to an array of linked lists (Figure 7d). However, these lists are not really required, as we can chain the records together using their *next* pointer. This optimization, which is presented in more detail in Section 3.2, takes place in the next step (Figure 7e). Finally, the code is converted to an embedded [Hudak 1996] version of the C language in Scala (Figure 7f) and, only then, SC generates the final C program out of this embedding (Figure 7g).

This example clearly illustrates that our optimizing compiler applies different optimizations in distinct transformation phases, thus guaranteeing the separation of concerns among different optimizations. For example, operator inlining is applied in the very first, high-level representation, which only describes operator objects. Performance concerns for data structures are then handled in subsequent optimization steps. Finally, low-level, code generation concerns are addressed only in the last, low-level representation. Next, we give more details about our individual optimizations.

3. COMPILER OPTIMIZATIONS

In this section, we present examples of compiler optimizations in six domains: (a) inter-operator optimizations for query plans, (b) transparent data-structure modifications,

```
1 def optimize(op: Operator): Operator = op match {
2   case joinOperator@HashJoinOp(aggOp:AggOp, rightChild, joinPred, leftHash, rightHash) =>
3     new HashJoinOp(aggOp.leftChild, rightChild, joinPred, leftHash, rightHash) {
4       override def open() {
5         // leftChild is now the child of aggOp (relation S)
6         leftChild foreach { t =>
7           // leftHash hashes according to the attributes referenced in the join condition
8           val key = leftHash(aggOp.grp(t))
9           // Get aggregations from the hash map of HashJoin
10          val aggs = hm.getOrElseUpdate(key, new Array[Double](aggOp.aggFuncs.size))
11          // Process all aggregations using the original code of Aggregate Operator
12          aggOp.process(aggs,t)
13        }
14      }
15    }
16  case op: Operator =>
17    op.leftChild = optimize(op.leftChild)
18    op.rightChild = optimize(op.rightChild)
19  // Operators with only one child have leftChild set, but rightChild null.
20  case null => null
21 }
```

Fig. 9: Removing redundant materializations by high-level programming (here between a group by and a join). The semantics (child-parent relationships) of this code segment are adapted to a Volcano-style engine. However, the same optimization logic can be similarly applied to a push engine. The code of the Aggregate Operator is given in Figure 4b. The `next` function of the HashJoinOp operator remains the same.

(c) changing the data layout, (d) using string dictionaries for efficient processing of string operations, (e) domain-specific code motion, and, finally, (f) traditional compiler optimizations like dead code elimination. The purpose of this section is to demonstrate the ease-of-use of our methodology: that by programming at the high-level, such optimizations are easily expressible without requiring changes to the base code of the query engine or interaction with compiler internals. Throughout this section we use, unless otherwise stated, Q12 of TPC-H, shown in Figure 8, as a guiding example in order to better illustrate various important characteristics of our optimizations.

3.1. Inter-Operator Optimizations – Eliminating Redundant Materializations

Consider again the motivating example of our introduction. We observed that existing query compilers use template-based generation and, thus, in such schemes operators are not aware of each other. This can cause redundant computation: in this example there are two materialization points (in the group by and in the left side of the hash join) where there could be only a single one.

By expressing optimizations at a higher level, LegoBase can optimize code across operator interfaces. For this example, we can treat operators as objects in Scala, and then match specific optimizations to certain chains of operators. Here, we can completely remove the aggregate operator and merge it with the join, thus eliminating the need of maintaining two distinct data structures. The code of this optimization is shown in Figure 9.

This optimization operates as follows. First, we call the optimize function, passing it the top-level operator as an argument. The function then traverses the tree of Scala operator objects, until it encounters a proper chain of operators to which the optimization can be applied to. In the case of the example this chain is (as shown in line 2 of

Figure 9) a hash-join operator connected to an aggregate operator. When this pattern is detected, a new HashJoinOp operator object is created, that is *not* connected to the aggregate operator, but instead to the child of the latter (first function argument in line 3 of Figure 9). As a result, the materialization point of the aggregate operator is completely removed. However, we must still find a place to (a) store the aggregate values and (b) perform the aggregation. For this purpose we use the hash map of the hash join operator (line 10), and we just call the corresponding function of the Aggregate operator (line 12), respectively. The processing of the tuples of the right-side relation (relation `R` in Figure 2), alongside with checking the join condition and the rest of join-related processing, still takes place during the call of `next` function of the HashJoinOp operator, similarly to the original query operator code.

We observe that this optimization is programmed in the same level of abstraction as the rest of the query engine: as normal Scala code. This allows to completely avoid code duplication during development, but more importantly it demonstrates that when coding optimizations at a high level of abstraction (e.g. to optimize the operators' interfaces), developers no longer have to worry about low-level concerns such as code generation (as is the case with existing approaches) – these concerns are simply addressed by later stages in the transformation pipeline. Both these properties raise the productivity provided by our system, showing the merit of developing database systems using high-level programming languages.

3.2. Data-Structure Specialization

Data-structure optimizations contribute significantly to the complexity of database systems today, as they tend to be heavily specialized to be workload, architecture and (even) query-specific. Our experience with the PostgreSQL[9] database management system reveals that there are many distinct implementations of the memory page abstraction and B-trees. These versions are *slightly* divergent from each other, suggesting that the optimization scope is limited. However, this situation significantly contributes to a *maintenance nightmare* as in order to apply any code update, many different pieces of code have to be modified.

In addition, even though data-structure specialization is important when targeting high-performance systems, it is not provided, to the best of our knowledge, by any existing query compilation engine. Since our approach can be used to optimize the *entire* Scala code, and not only the operator interfaces, it allows for various degrees of specialization in data structures, as has been previous shown in [Rompf et al. 2013].

In this article, we demonstrate such possibilities by explaining how our compiler can be used to: (1) Optimize the data structures used to hold in memory the data of the input relations, (2) Optimize Hash Maps which are typically used in intermediate computations like aggregations, and, finally, (3) Automatically infer and construct indices for SQL attributes of `date` type. We do so in the next three sections.

3.2.1. Data Partitioning. Optimizing the structures that hold the data of the input relations is an important form of data-structure specialization, as such optimizations generally enable more efficient join processing throughout query execution. We have observed that this is true even for multi-way, join-intensive queries. In LegoBase, we perform data partitioning when loading the input data. We analyze this optimization, the code of which can be found in Appendix B, next.

To begin with, in LegoBase developers can annotate the primary and foreign keys of their input relations, at schema definition time. Using this information, our system then creates optimized data structures for those relations, as follows.

[9]http://www.postgresql.org

First, for each input relation, LegoBase creates a data structure which is accessed through the primary key specified for that relation. There are two possibilities:

— For single-attribute primary keys, the value of this attribute in each tuple is used to place the tuple in a continuous 1D-array. For example, for the relations of the TPC-H workload this is a straightforward task as the primary keys are typically integer values in the range of $[1...\#num\_tuples]$. However, even when the primary key is not in a continuous value range, LegoBase currently aggressively trades-off system memory for performance, and stores the input data into a sparse array.
— For composite primary keys (e.g. those of the LINEITEM table of TPC-H), creating an 1D array does not suffice, as there may be multiple tuples with the same value for *any one* of the attributes of the primary key (thus causing conflicts when accessing the array). One possible solution would be to hash *all* attributes of the primary key and guarantee that we get a unique index value to access the 1D-array. However, deriving such a function in full generality requires knowledge of the whole dataset in advance (in order to know all possible combinations of the primary key). More importantly, it introduces additional computation on the critical path in order to perform the hash, a fact that, according to our observations, can lead to significant, negative impact on performance. For this reason, LegoBase does not create an 1D array and, instead, handles such primary keys similarly to the handling of foreign key, as we discuss shortly.

For the example given in Figure 8, LegoBase creates a 1D array for the ORDERS table, indexed through the O_ORDERKEY attribute, but does not create a data structure accessed through the primary key for LINEITEM (as this relation has a composite primary key of the L_ORDERKEY, L_LINENUMBER attributes).

Second, LegoBase replicates and repartitions the data of the input relations based on each specified foreign key. This basically leads to the creation of a two-dimensional array, indexed by the foreign key, where each bucket holds all tuples having a particular value for that foreign key. We also apply the same partitioning technique for relations that have composite primary keys, as we mentioned above. We resolve the case where the foreign key is not in a contiguous value range by trading-off system memory, in a similar way to how we handled primary keys.

For the example of Q12, LegoBase creates four partitioned tables: one for the foreign key of the ORDERS table (O_CUSTKEY), one for the composite primary key of the LINEITEM table (as described above), and, finally, two more for the foreign keys of the LINEITEM table (on L_ORDERKEY and L_PARTKEY/L_SUPPKEY respectively).

Observe that for relations that have multiple foreign keys, not all corresponding partitioned input relations need to be kept in memory at the same time, as an incoming SQL query may not need to use all of them. To decide which partitioned tables to load, LegoBase depends mainly on the derived physical query execution plan (attributes referenced as well as select and join conditions of the input query), but also on simple to estimate statistics, like cardinality estimation of the input relations. For example, for Q12, out of the two partitioned, foreign-key data structures of LINEITEM, our optimized generated code for Q12 uses only the partitioned table on L_ORDERKEY, as there is no reference to attributes L_PARTKEY or L_SUPPKEY in the query.

These data structures can be used to significantly improve join processing, as they allow to quickly extract matching tuples on a join between two relations on attributes that have a primary-foreign key relationship. This is best illustrated through our running example of Q12 and the join between the LINEITEM and ORDERS tables. For this query, LegoBase (a) infers that the ORDERKEY attribute actually represents a primary-foreign key relationship and (b) uses statistics to derive that ORDERS is the smaller of the two tables. By utilizing this information, LegoBase can generate the

```
// Sequential accessing for the ORDERS table (since it has smaller size)
for (int idx = 0 ; idx < ORDERS_TABLE_SIZE ; idx += 1) {
  int O_ORDERKEY = orders_table[idx].O_ORDERKEY;
  struct LINEITEMTuple* bucket = lineitem_table[O_ORDERKEY];
  for (int i = 0; i < counts[bucket]; i+=1) {
    // process bucket[i] -- a tuple of the LINEITEM table
  }
}
```

Fig. 10: Using primary and foreign keys in order to generate code for high-performance join processing. The underlying storage layout is that of a row-store for simplicity. The `counts` array holds the number of elements that exist in each bucket.

code shown in Figure 10 in order to directly get the corresponding bucket of the array of LINEITEM (by using the value of the ORDERKEY attribute), thus avoiding the processing of a possibly significant number of LINEITEM tuples.

LegoBase uses this approach for multi-way joins as well, to completely eliminate the overhead of intermediate data structures for most TPC-H queries. This results in significant performance improvement as the corresponding tuple copying between these intermediate data structures is completely avoided, thus reducing memory pressure and improving cache locality. In addition, a number of expensive system calls responsible for the tuple copying is also avoided by applying this optimization.

After the aforementioned optimization has been performed, LegoBase has removed the overhead of using generic data structures for join processing, but there are still some hash maps remaining in the generated code. These are primarily hash maps which correspond to aggregations, as in this case there is no primary/foreign key information that can be used to optimize these data structures away, but also hash maps which process joins on attributes that are not represented by a primary/foreign key relationship. In these cases, LegoBase lowers these maps to two-dimensional arrays as we discuss in our hash map lowering optimization in the next section.

3.2.2. Optimizing Hash Maps. Next, we show how hash maps, which are the most commonly used data-structures along with Trees in DBMSes, can be specialized for significant performance improvement by using schema and query knowledge.

By default, LegoBase uses GLib [The GNOME Project 2013] hash tables for generating C code out of the HashMap constructs of the Scala language. Close examination of these generic hash maps in the baseline implementation of our operators (e.g. in the Aggregation of Figure 4b) reveals the following three main abstraction overheads.

First, for every *insert* operation, a generic hash map must allocate a container holding the key, the corresponding value as well as a pointer to the next element in the hash bucket. This introduces a significant number of expensive memory allocations on the critical path. Second, hashing and comparison functions are called for every *lookup* in order to acquire the correct bucket and element in the hash list. These function calls are usually virtual, causing significant overhead on the critical path. Finally, the data structures may have to be resized *during runtime* in order to efficiently accommodate more data. These operations typically correspond to (a) allocating a bigger memory space, (b) copying the old data over to the new memory space and, finally, (c) freeing the old space. These resizing operations are a significant bottleneck, especially for long-running, computationally expensive queries.

Next, we resolve all these issues with our compiler, without changing a single line of the base code of the operators that use these data structures, or the code of other optimizations. This property shows that our approach, which is based on using a high-

```
1 class HashMapToArray extends RuleBasedTransformer {
2  rewrite += rule {
3   case code"new HashMap[K, V]($size, $hashFunc, $equalFunc)" => {
4    // Create new array for storing only the values
5    val arr = code"new Array[V]($size)"
6    // Keep hash and equal functions in the metadata of the new object
7    arr.attributes += "hash" -> hashFunc
8    arr.attributes += "equals" -> equalFunc
9    arr // Return new object for future reference
10   }
11  }
12  rewrite += rule {
13   case code"($hm: HashMap[K, V]).getOrElseUpdate($key, $value)" => {
14    val arr = transformed(hm) // Get the array representing the original hash map
15    // Extract functions
16    val hashFunc = arr.attributes("hash")
17    val equalFunc = arr.attributes("equals")
18    code"""
19     // Get bucket
20     val h = $hashFunc($value) // Inlines hash function
21     var elem = $arr(h)
22     // Search for element & inline equals function
23     while (elem != null && !$equalFunc(elem, $key))
24      elem = elem.next
25     // Not found: create new elem / update pointers
26     if (elem == null) {
27      elem = $value
28      elem.next = $arr(h)
29      $arr(h) = elem
30     }
31     elem
32    """
33   }
34  }
35  // Lowering of remaining operations is done similarly
36 }
```

Fig. 11: Specializing HashMaps by converting them to native arrays. The corresponding operations are mapped to a set of primitive C constructs.

level compiler API, is practical for specializing DBMS components. The transformation, shown in Figure 11, is applied during the *lowering* phase of the compiler (Section 2.3), where high-level Scala constructs are mapped to low-level C constructs. The optimization lowers Scala HashMaps to native C arrays and inlines the corresponding operations, by making use of the following three observations:

1. For our workloads, the information stored on the key is *usually* a subset of the attributes of the value. Thus, generic hash maps store redundant data. To avoid this, whenever a functional dependency between key and value is detected, we convert the hash map to a native array that stores only the values, and not the associated key (lines 2-11). Then, since the inserted elements are anyway chained together in a hash list, we provision for the next pointer when these are first allocated (e.g. at data loading, *outside the critical path*[10]). Thus, we no longer need the key-value-next container and we manage to reduce the amount of memory allocations significantly.

[10]The transformer shown in Figure 11 is applied only for the code segment that handles basic query processing. There is another transformer which handles the provision of the next pointer during data loading.

2. Second, the SC optimizing compiler offers function inlining for any Scala function out-of-the-box. Thus, our system can automatically inline the body of the hash and equal functions wherever they are called (lines 20 and 23 of Figure 11). This significantly reduces the number of function calls (to almost zero), considerably improving query execution performance.
3. Finally, to avoid costly maintenance operations on the critical path, we preallocate in advance all the necessary memory space that *may* be required for the hash map during execution. This is done by specifying a size parameter when allocating the data structure (line 3). Currently, we obtain this size by performing worst-case analysis on a given query, which means that we possibly allocate much more memory space that what is actually needed. However, database statistics can make this estimation very accurate, as we show in our experiments section where we evaluate the overall memory consumption of our system in more detail.

For our running example, the aggregation array, created in step 1 above, is accessed using the integer value obtained from hashing the `L_SHIPMODE` string. Then, the values located into the corresponding bucket of the array are checked one by one, in order to see if the value of `L_SHIPMODE` exists and if a match is found, the aggregation entries are updated accordingly, or a new entry is initialized otherwise.

In addition to the above optimizations, the SC optimizing compiler also detects hash table data structures that receive only a single, *statically-known* key and converts each such structure to a single value, thus completely eliminating the unnecessary abstraction overhead of these tables. In this case, this optimization maps all related HashMap operations to operations in the single value. For example, we convert a `foreach` to a single value lookup. An example of such a lowering is in aggregations which calculate one single global aggregate (in this case `key = ''TOTAL''`). This happens for example in Q6 of the TPC-H workload.

Finally, we note that data-structure specialization is an example of intra-operator optimization and, thus, each operator can specialize its own data-structures by using similar optimizations as the one shown in Figure 11.

3.2.3. Automatically Inferring Indices on Date Attributes. Assume that an SQL query needs to *fully* scan an input relation in order to extract tuples belonging to a particular year. A naive implementation would simply execute an **`if`** condition for each tuple of the relation and propagate that tuple down the query plan if the check was satisfied. However, it is our observation that such conditions, as simple as they may be, can have a pronounced negative impact on performance, as they can significantly increase the total number of CPU instructions executed in a query.

Thus, for such cases, LegoBase uses the aforementioned partitioning mechanism in order to automatically create indices, at data loading time, for all attributes of `date` type. It does so by grouping the tuples of a date attribute based on the year, thus forming a two-dimensional array where each bucket holds all tuples of a particular year. This design allows to immediately skip, at query execution time, all years for which this predicate is incorrect. That is, as shown in Figure 12, the **`if`** condition now just checks whether the first tuple of a bucket is of a particular year and if not the whole bucket is skipped, as *all* of its tuples have the same year and, thus, they *all* fail to satisfy the predicate condition.

These indices are particularly important for queries that process large input relations, whose date values are uniformly distributed across years. This is the case, for example, for the LINEITEM and ORDERS tables of TPC-H, whose date attributes are always populated with values ranging from 1992-01-01 to 1998-12-31 [Transaction Processing Performance Council 1999].

```
// Sequential scan through table
for (int idx=0 ; idx<TABLE_SIZE ; idx+=1) {

  if (table[idx].date >= "01-01-1994" &&
      table[idx].date <= "31-12-1994")
     // Propagate tuple down the query plan
}
```

(a) Original, naive code

```
// Sequential scan through table
for (int idx=0 ; idx<NUM_BUCKETS ; idx+=1) {
  // Check only the first entry
  if (table[idx][0].date >= "01-01-1994" &&
      table[idx][0].date <= "31-12-1994")
      // Propage all tuples of table[idx]
}
```

(b) Optimized code

Fig. 12: Using `date` indices to speed up selection predicates on large relations.

3.3. Changing Data Layout

A long-running debate in database literature is the one between row and column stores [Stonebraker et al. 2005; Harizopoulos et al. 2006; Abadi et al. 2008]. Even though there are many significant differences between the two approaches in all levels of the database stack, the central contrasting point is the *data-layout*, i.e. the way data is organized and grouped together. By default LegoBase uses the row layout, since this intuitive data organization facilitated fast development of the relational operators. However, we quickly noted the benefits of using a column layout for efficient data processing. One solution would be to go back and redesign the whole query engine; however this misses the point of our compiler framework. In this section, we show how the transition from row to column layout can be expressed as an optimization[11].

The optimization of Figure 13 performs a conversion from an array of records (row layout) to a record of arrays (column layout), where each array in the column layout stores the values for *one* attribute. The optimization basically overwrites the default lowering for arrays, thus providing the new behavior. As with all our optimizations, *type information* determines the applicability of an optimization: here it is performed only if the array elements are of record type (lines 3,13,26). Otherwise, this transformation is a NOOP and the original code is generated (e.g. an array of Integers remains unchanged).

Each optimized operation is basically a straightforward rewriting to a set of operations on the underlying record of arrays. Consider, for example, an update to an array of records (`arr(n) = v`), where **v** is a record. We know that the *new* representation of `arr` will be a record of arrays (column layout), and that `v` has the same attributes as the elements of `arr`. So, for each of those attributes we extract the corresponding array from `arr` (line 18) and field from `v` (line 19); then we can perform the update operation on the extracted array (line 19) using the same index.

This optimization also reveals another benefit of using an optimizing compiler: developers can create *new* abstractions in their optimizations, which will be in turn optimized away in *subsequent* optimization passes. For example, array_apply results in *record reconstruction* by extracting the individual record fields from the record of arrays (lines 29-34) and then building a new record to hold the result (line 35). This intermediate record can be *automatically* removed using dead code elimination (DCE), as shown in Figure 14. Similarly, if SC can statically determine that some attribute is never used (e.g. by having all queries given in advance), then this attribute will just be an unused field in a record, which the optimizing compiler will be able to optimize away (e.g. attribute L2 in Figure 14).

[11] We must note that changing the data layout does not mean that LegoBase becomes a column store. There are other important aspects which we do not yet handle, and which we plan to investigate in future work.

```
class ColumnarLayoutTransformer extends RuleBasedTransformer {
 rewrite += rule {
  case code"new Array[T]($size)" if typeRep[T].isRecord => typeRep[T] match {
   case RecordType(recordName, fields) => {
    val arrays =
     for((name, tp: TypeRep[Tp]) <- fields) yield
      name -> code"new Array[Tp]($size)"
    record(recordName, arrays)
   }
  }
 }
 rewrite += rule {
  case code"(arr:Array[T]).update($idx,$v)" if typeRep[T].isRecord => typeRep[T] match {
   case RecordType(recordName, fields) => {
    val columnarArr = transformed(arr) // Get the record of arrays
     for((name, tp: TypeRep[Tp]) <- fields) {
      code """
       val fieldArr: Array[Tp] = record_field($columnarArr, $name)
       fieldArr($idx) = record_field($v, $name)
      """
     }
   }
  }
 }
 rewrite += rule {
  case code"(arr:Array[T]).apply($index)" if typeRep[T].isRecord => typeRep[T] match {
   case RecordType(recordName, fields) => {
    val columnarArr = transformed(arr) // Get the record of arrays
    val elems = for((name, tp: TypeRep[Tp]) <- fields) yield {
     name -> code """
      val fieldArr: Array[Tp] = record_field($columnarArr, $name)
      fieldArr($index)
     """
    }
    record(recordName, elems)
   }
  }
 }
 // Fill remaining operations accordingly
}
```

Fig. 13: Changing the data layout (from row to column) expressed as an optimization. Scala's `typeRep` carries type information, which is used to differentiate between Array[Rec] and other non-record arrays (e.g. an array of integers).

We notice that, as was the case with previously presented optimizations, the transformation described in this section does not have any dependency on other optimizations or the code of the query engine. This is because it is applied in the distinct optimization phase that handles *only* the optimization of arrays. This separation of concerns leads, as discussed previously, to a significant increase in productivity as, for example, developers that tackle the optimization of individual query operators do not have to worry about optimizations handling the data layout (described in this section).

3.4. String Dictionaries

Operations on non-primitive data types, such as strings, incur a very high performance overhead. This is true for two reasons. First, there is typically a function call required.

```
val a1 = a.L1
val a2 = a.L2
val e1 = a1(i)
val e2 = a2(i)
val r =
  record(L1->e1,
         L2->e2)
r.L1
```

$\mapsto$

```
val a1 = a.L1
val a2 = a.L2
val e1 = a1(i)
val e2 = a2(i)
val r =
  record(L1->e1,
         L2->e2)
e1
```

$\mapsto$

```
val a1 = a.L1
val a2 = a.L2
val e1 = a1(i)
val e2 = a2(i)
e1
```

$\mapsto$

```
val a1 = a.L1
val e1 = a1(i)
e1
```

Fig. 14: Dead code elimination (DCE) can remove intermediate materializations, e.g. row reconstructions when using a column layout. Here a is a record of arrays (column-layout) and i is an integer. The records have only two attributes *L1* and *L2*. The notation `L1->v` associates the label (attribute name) L1 with value v.

String Operation	C code	Integer Operation	Dictionary Type
equals	strcmp(x, y) == 0	x == y	Normal
notEquals	strcmp(x, y) != 0	x != y	Normal
startsWith	strncmp(x, y, strlen(y)) == 0	x>=start && x<=end	Ordered
indexOfSlice	strstr(x, y) != NULL	N/A	Word-Token

Table II: Mapping of string operations to integer operations through the corresponding type of string dictionaries. Variables x and y are strings arguments which are mapped to integers. The rest of string operations are mapped in a similar way.

Second, most of these operations typically need to execute loops to process the encapsulated data. For example, *strcmp* needs to iterate over the underlying array of characters, comparing one character from each of the two input strings on each iteration. Thus, such operations significantly affect branch prediction and cache locality.

LegoBase uses String Dictionaries to remove the abstraction overhead of Strings. Our system maintains one dictionary for every attribute of String type, which generally operates as follows. First, at data loading time, each string value of an attribute is mapped to an integer value. This value corresponds to the index of that string in a single linked-list holding the *distinct* string values of that attribute. The list basically constitutes the dictionary itself. In other words, each time a string appears for the first time during data loading, a unique integer is assigned to it; if the same string value reappears in a later tuple, the dictionary maps this string to the previously assigned integer. Then, at query execution time, string operations are mapped to their integer counterparts, as shown in Table II. This mapping allows to significantly improve the query execution performance, as it typically eliminates underlying loops and, thus, significantly reduces the number of CPU instructions executed. For our running example, LegoBase compresses the attributes L_SHIPMODE and O_ORDERPRIORITY by converting the six string equality checks into corresponding integer comparisons.

Special care is needed for string operations that require ordering. For example, Q2 and Q14 of TPC-H need to perform the *endsWith* and *startsWith* string operations with a constant string, respectively. This requires that we utilize a dictionary that maintains the data in order; that is, if $string_x < string_y$ lexicographically, then $Int_x < Int_y$ as well. To do so, we take advantage of the fact that in LegoBase all input data is already materialized, and thus we can first compute the list of distinct values, as mentioned above, then sort this list lexicographically, and afterwords make a second pass over the data to assign integer values to the string attribute. By doing so, the constant string is then converted to a $[start, end]$ range, by iterating over the list of distinct values and finding the first and last strings which start or end with that particular string. This range is then used when lowering the operation, as shown in Table II. This *two-phase* string dictionary allows to map all operations that require some notion of ordering in string operations.

In addition, there is one additional special case where the string attributes need to be tokenized on a word granularity. This happens for example in Q13 of TPC-H. This is because queries like that one need to perform the *indexOfSlice* string operation, where the slice represents a word. LegoBase provides a *word-tokenizing* string dictionary that contains all words in the strings instead of the string attributes themselves to handle such cases. Then, searching for a word slice is equal to looking through all the integer-typed words in that string for a match during query execution. This is the only case where the integer counterparts of strings operations contain a loop. It is however our experience, that even with this loop through the integer vales, the obtained performance significantly outperforms that of the `strstr` function call of the C library. This may be because such loops can be more easily vectorized by an underlying C compiler like CLang, compared to the corresponding loops using the string types.

Finally, it is important to note that string dictionaries, even though they significantly improve query execution performance, they have an even more pronounced negative impact on the performance of data loading. This is particularly true for the word-tokenizing string dictionaries, as the impact of tokenizing a string is significant. In addition, string dictionaries can actually degrade performance when they are used for primary keys or for attributes that contain many distinct values (as in this case the string dictionary significantly increases memory consumption). In such cases, LegoBase can be configured so that it does not use string dictionaries for those attributes, through proper usage of the optimization pipeline described in Section 2.

3.5. Domain-Specific Code Motion

Domain-Specific code motion includes optimizations that remove code segments that have a negative impact on query execution performance from the critical path and instead executes the logic of those code segments during data loading. Thus, the optimizations in this category trade-off increased loading time for improved query execution performance. There are two main optimizations in this category, described next.

3.5.1. Hoisting Memory Allocations. Memory allocations can cause significant performance degradation in query execution. Our experience shows that, by taking advantage of type information available in each SQL query, we can completely eliminate such allocations from the critical path. The LegoBase system provides the following optimization for this purpose.

At query compilation time, information is gathered regarding the data types used throughout an incoming SQL query. This is done through an analysis phase, where the compiler collects all `malloc` nodes in the program, once the latter has been lowered to the abstraction level of C code. This is necessary to be done at this level, as high-level programming languages like Scala provide implicit memory management, which the SC optimizing compiler cannot currently optimize. The obtained types correspond either to the initial database relations (e.g. the LINEITEM table of TPC-H) or to types required for intermediate results, such as aggregations. Based on this information, SC initializes memory pools during data loading, one for each type.

Then, at query execution time, the corresponding `malloc` statements are replaced with references to those memory pools. We have observed that this optimization significantly reduces the number of CPU instructions executed during the query evaluation, and significantly contributes to improving cache locality. This is because the memory space allocated for each pool is contiguous and, thus, each cache miss brings useful records to the cache (this is not the case for the fragmented memory space returned by the `malloc` system calls). We also note that it is necessary to resolve dependencies between data types. This is particularly true for composite types, which need to reference the pools of the native types (e.g. the pool for Strings). We resolve such dependencies by first applying topological sorting on the obtained type information and only then generating the pools in the proper order.

Finally, we must mention that the size of the memory pools is estimated by performing worst-case analysis on a given query. This means that LegoBase may allocate much more space than needed. However, we have confirmed that our estimated statistics are accurate enough so that the pools do not unnecessarily create memory pressure, thus negatively affecting query performance.

3.5.2. Hoisting Data-Structure Initialization. The proper initialization and maintenance of any data structure needed during query execution generally require specific code to be executed in the critical path. This is typically true for data structures representing some form of *key-value* stores, as we describe next.

Consider the case of TPC-H Q12, for which a data structure is needed to store the results of the aggregate operator. Then, when evaluating the aggregation during query execution, we must check whether the corresponding key of the aggregation has been previously inserted in the aggregation data structure. In this case, the code must check whether a specific value of O_ORDERPRIORITY is already present in the data structure. If so, it would return the existing aggregation. Otherwise, it would insert a new aggregation into the data structure. This means that *at least one* **`if`** condition must be evaluated for *every* tuple that is processed by the aggregate operator. We have observed that such **`if`** conditions, which exist purely for the purpose of data-structure initialization, significantly affect branch prediction and overall query execution performance.

LegoBase provides an optimization to remove such data-structure initialization from the critical path by utilizing domain-specific knowledge. For example, LegoBase takes advantage of the fact that aggregations can usually be statically initialized with the value zero, for each corresponding key. To infer all these possible key values (i.e. infer the *domain* of that attribute), LegoBase utilizes the statistics collected during data loading for the input relations. Then, at query execution time, the corresponding **`if`** condition mentioned above no longer needs to be evaluated, as the aggregation already exists and can be accessed directly. We have observed that by removing code segments that perform *only* data-structure initialization, branch prediction is improved and the total number of CPU instructions executed is significantly reduced as well.

Observe that this optimization is not possible in its full generality, as it directly depends on the ability to predict the possible key values in advance, during data loading. However, we note three things. First, once our partitioning optimization (Section 3.2.1) has been applied, LegoBase requires intermediate data structures mostly for aggregate operators, whose initialization code segment we remove, as described above. Second, particularly for TPC-H, there is no key that is the result of an intermediate join operator deeply nested in the query plan. Instead, TPC-H uses attributes of the original relations to access most data structures, attributes whose value range can be accurately estimated during data loading through statistics, as we discussed previously. Finally, for TPC-H queries the key value range is very small, typically ranging up to a couple of thousand sequential key values[12]. These three properties allow to completely

[12] A notable exception is TPC-H Q18 which uses O_ORDERKEY as a key, which has a sparse distribution of key values. LegoBase generates a specialized data structure for this case.

remove initialization overheads and the associated unnecessary computation for all TPC-H queries.

3.6. Traditional Compiler Optimizations

In this section, we present a number of traditional compiler optimizations that originate mostly from work in the PL community. Most of them are generic in nature, and, thus, they are offered out-of-the-box by the SC optimizing compiler.

3.6.1. Removal of Unused Relational Attributes. In Section 3.3 we mentioned that LegoBase provides an optimization for removing relational attributes that are not accessed by a particular SQL query, assuming that this query is known *in advance*. For example, the Q12 running example references eight relational attributes. However, the relations LINEITEM and ORDERS contain 25 attributes in total. LegoBase avoids loading these unnecessary attributes into memory at data loading time. It does so by analyzing the input SQL query and removing the set of unused fields from the record definitions. This reduces memory pressure and improves cache locality.

3.6.2. Removing Unnecessary Let-Bindings. The SC compiler uses the Administrative Normal Form (ANF) when generating code. This simplifies code generation for the compiler. However, it has the negative effect of introducing many unnecessary intermediate variables. We have observed that this form of code generation not only affects code compactness but also significantly increases register pressure. To improve upon this situation, SC uses a technique first introduced by the programming language community [Sumii and Kobayashi 2001], which removes any intermediate variable that satisfies the following three conditions: the variable (a) is set only once, (b) has no side effects, and, finally, (c) is initialized with a single value (and thus its initialization does not correspond to executing possibly expensive computation). SC then replaces any appearance of this variable later in the code with its initialization value. We have noticed that this optimization makes the generated code much more compact and reduces register pressure, resulting in improved performance. Moreover, we have observed that since the variable initialization time may take place significantly earlier in the code of the program than its actual use, this does not allow for this optimization opportunity to be detected by low-level compilers like LLVM.

Finally, our compiler applies a technique known as *parameter promotion*[13]. This optimization removes *structs* whose fields can be flattened to local variables. This optimization has the effect of removing a memory access from the critical path as the field of a struct can be referenced immediately without referencing the variable holding the struct itself. We have observed that this optimization significantly reduces the number of memory accesses occurring during query execution.

3.6.3. Fine-grained Compiler Optimizations. Finally, there is a category of fine-grained compiler optimizations that are applied last in the compilation pipeline. These optimizations target optimizing very small code segments (even individual statements) under particular conditions. We briefly present three such optimizations next.

First, SC can transform arrays to a set of local variables. This lowering is possible only when (a) the array size is statically known at compile time, (b) the array is relatively small (to avoid increasing register pressure) and, finally, (c) the index of every array access can be inferred at compile time (otherwise, the compiler is not able to know to which local variable an array access should be mapped to).

Second, the compiler provides an optimization to change the boolean condition `x && y` to `x & y` where `x` and `y` both evaluate to boolean and the second operand does not

[13]This technique is also known as Scalar Replacement in the PL community.

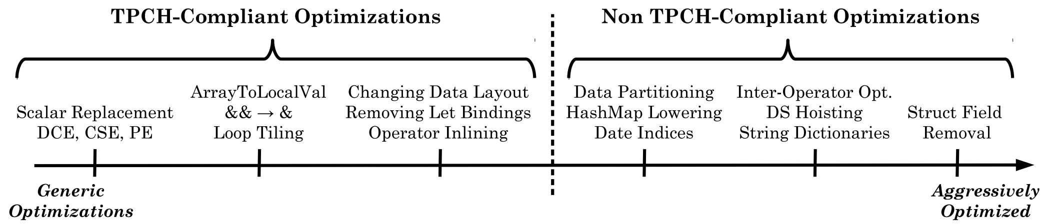

Fig. 15: Classification of LegoBase optimizations.

have any side effect. According to our observations, this optimization can significantly improve branch prediction, when the aforementioned conditions are satisfied.

Finally, the compiler can be instructed to apply tiling to `for` loops whose range are known at compile time (or can be accurately estimated).

It is our observation that all these fine-grained optimizations (as described above), which can be typically written in less than a hundred lines of code, can help to improve the performance of certain queries. More importantly, since they have very fine-grained granularity, their application does not introduce additional performance overheads.

3.7. Discussion

In this section, we classify the LegoBase optimizations according to (a) their generality and (b) whether they follow the rules of the TPC-H benchmark, which we use in our evaluation. These two metrics are important for a more thorough understanding of which categories of database systems can benefit from these optimizations. We detect six groups of optimizations, illustrated in Figure 15, described next in the order they appear from left to right in the figure.

Generic Compiler Optimizations: In this category, we include optimizations which are also applied by traditional compilers, such as LLVM. These include Dead Code Elimination (DCE), Common Subexpression Elimination (CSE), Partial Evaluation (PE) and the Scalar Replacement optimization presented in Section 3.6.2. These optimizations are TPC-H compliant and do not require any particular domain-specific knowledge; thus they can be applied for optimizing any input query as well as the code of the query engine.

Fine-grained Optimizations: In this TPC-H compliant category we include, as described in Section 3.6.3, fine-grained optimizations that aim to transform and improve the performance of individual statements (or a small number of contiguous statements). We do not list this category alongside the generic compiler optimizations, as whether they improve the performance or not depends on the characteristics of the input query. Thus, SC needs to analyze the program before detecting whether the application of one of the optimizations in this group is beneficial.

Optimizing Data Accesses: The two optimizations presented in Sections 3.3 and 3.6.2, alongside the generic *operator inlining* optimization, aim to improve performance by minimizing the number of function calls and optimizing data accesses and code compactness. Even though they are coarse-grained in nature, affecting large code segments, they are still TPC-H compliant, as they are neither query specific nor depend on type information.

Partitioning and Indexing Optimizations: This class of optimizations, presented in detail in Section 3.2, aims to improve query execute performance. However, even though they provide significant performance improvement (as we show in our evaluation), they are not TPC-H compliant, as this workload does not allow logical replication of data. Similarly, our HashMap lowering optimization requires knowledge of the domain of the aggregation keys in advance. Still, there is a class of database systems that can greatly benefit from such indexing and partitioning transformations. These include systems that have all their data known in advance (e.g. OLAP style processing) or systems where we can introduce pre-computed indexing views, as in the case of Incremental View Maintenance (IVM).

Inter-Operator, String Dictionaries, and Domain-Specific Hoisting Optimizations: The three optimizations in this category, presented in Sections 3.1, 3.4 and 3.5 respectively, aim to remove unnecessary materialization points and computation from the critical path. However, they are *query specific*, as they can only be applied if a query is known in advance. This is the primary characteristic that differentiates this category of optimizations from the previous one. They also depend on type information and introduce auxiliary data structures. Thus, they are not TPC-H compliant.

Struct Field Removal Optimization: The most aggressive optimization that LegoBase applies removes unnecessary relational attributes from C structs. This optimization is query specific and is highly dependant on type information. It also requires specializing the data structures during data loading (to remove the unnecessary fields). Thus, it is not TPC-H compliant.

4. EVALUATION

In this section, we evaluate the realization of the abstraction without regret vision in the domain of analytical query processing. After briefly presenting our experimental platform, we address the following topics and open questions related to LegoBase:

(1) How well can general-purpose compilers, such as LLVM or GCC, optimize query engines? We show that these compilers fail to detect many high-level optimization opportunities and, thus, they perform poorly compared to our system (Section 4.2).
(2) Is the code generated by LegoBase competitive, performance-wise, to (a) traditional database systems and (b) query compilers based on template expansion? We show that by utilizing query-specific knowledge and by extending the scope of compilation to optimize the *entire* query engine, we can obtain a system that significantly outperforms both alternative approaches (Section 4.3).
(3) We experimentally validate that the source-to-source compilation from Scala to efficient, low-level C binaries is necessary as even highly optimized Scala programs exhibit a considerably worse performance than C (Section 4.4).
(4) What insights can we gain by analyzing the performance improvement of individual optimizations? Our analysis reveals that important optimization opportunities have been so far neglected by compilation approaches that optimize *only* queries. To demonstrate this, we compare architectural decisions as fairly as possible, using a shared codebase that only differs by the effect of a single optimization (Section 4.5).
(5) How much are the overall memory consumption and data loading speed of our system? These two metrics are of importance to main-memory databases, as a query engine must perform well in both directions to be usable in practice (Section 4.6).
(6) We analyze the amount of effort required when coding query engines in LegoBase and show that, by programming in the abstract, we can derive a fully functional system in a relatively short amount of time and coding effort (Section 4.7).

System	Description	Compiler optimizations	TPC-H compliant	Uses query-specific info
DBX	Commercial, in-memory DBMS	No compilation	Yes	No
Compiler of HyPer	Query compiler of the HyPer DBMS	Operator inlining, push engine	Yes	No
LegoBase (Naive)	A naive engine with the minimal number of optimizations	Operator inlining, push engine	Yes	No
LegoBase (TPC-H/C)	TPC-H compliant engine	Operator inlining, push engine, data partitioning	Yes[14]	No
LegoBase (StrDict/C)	Non TPC-H compliant engine with some optimizations applied	Like above, plus String Dictionaries	No	No
LegoBase (Opt/C)	Optimized push-style engine	All optimizations of this article	No	Yes
LegoBase (Opt/Scala)	Optimized push-style engine	All optimizations of this article	No	Yes

Table III: Description of all systems evaluated in this section. Unless otherwise stated, all generated C programs of LegoBase are compiled to a final C binary using CLang. All listed LegoBase engines and optimizations are written with *only* high-level Scala code, which is then optimized and compiled to C or Scala code with SC.

(7) We evaluate the compilation overheads of our approach. We show that SC can efficiently compile query engines even for the complicated, multi-way join queries typically found in analytical query processing (Section 4.8).

4.1. Experimental Setup

Our experimental platform consists of a server-type x86 machine equipped with two Intel Xeon E5-2620 v2 CPUs running at 2GHz each, 256GB of DDR3 RAM at 1600Mhz and two commodity hard disks of 2TB storing the experimental datasets. The operating system is Red Hat Enterprise 6.7. For all experiments, we have disabled huge pages in the kernel, since this provided better results for all tested systems (described in more detail in Table III). For compiling the generated programs throughout the evaluation section, we use version 2.11.4 of the Scala compiler and version 3.4.2 of the CLang front-end for LLVM [Lattner and Adve 2004], with the default optimization flags set for both compilers. For the Scala programs, we configure the Java Virtual Machine (JVM) to run with 192GB of heap space, while we use the GLib library (version 2.38.2) [The GNOME Project 2013] whenever we need to generate generic data structures in C.

For our evaluation, we use the TPC-H benchmark [Transaction Processing Performance Council 1999]. TPC-H is a data warehousing and decision support bench-

[14]We note that according to the TPC-H specification rules, a database system can employ data partitioning (as described in Section 3.2.1) and still be TPC-H compliant. This is the case when all input relations are partitioned on *one and only one* primary or foreign key attribute across all queries. The LegoBase(TPC-H/C) configuration of our system follows exactly this partitioning approach, which is also used by the HyPer system (but in contrast to SC, partitioning in HyPer is not expressed as a compiler optimization).

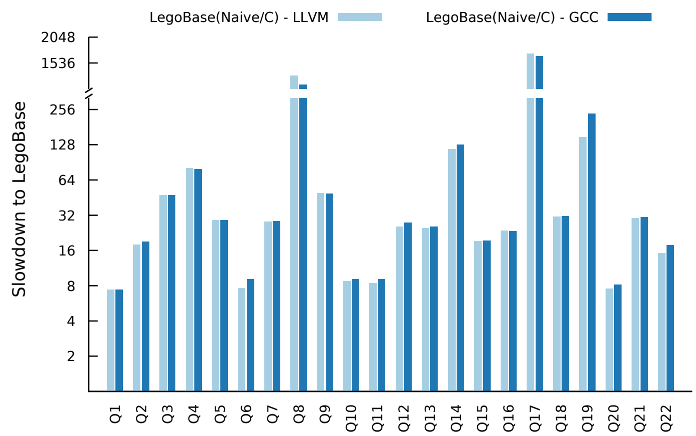

Fig. 16: Performance of a push-style engine compiled with LLVM and GCC. These engines are generated using *only* operator inlining. The baseline is the performance of the optimal generated code, LegoBase(Opt/C), with all optimizations enabled.

mark that issues business analytics queries to a database with sales information. This benchmark suite includes 22 queries with a high degree of complexity that express most SQL features. We use all 22 queries to evaluate various design choices of our methodology. We execute each query five times and report the average performance of these runs. Unless otherwise stated, the scaling factor of TPC-H is set to 8 for all experiments. It is important to note that the final generated optimized code of LegoBase (configurations LegoBase(Opt/C) and LegoBase(Opt/Scala) in Table III) employs materialization (e.g. for the date indices) and, thus, this version of the code does *comply* with the TPC-H implementation rules. However, we also present a TPC-H compliant configuration, LegoBase(TPC-H/C), for comparison purposes.

As a reference point for most results presented in this section, we use a commercial, in-memory, row-store database system called DBX, which does not employ compilation. We assign 192GB of DRAM as memory space in DBX, and we use the DBX-specific data types instead of generic SQL types. As described in Section 2, LegoBase uses query plans from the DBX database. We also use the query compiler of the HyPer system [Neumann 2011] (v0.4-452) as a point of comparison with existing query compilation approaches. Since parallel execution is still under development at the time of writing for LegoBase, all systems have been forced to single-threaded execution, either by using the execution parameters some of them provide or by manually disabling the usage of CPU cores in the kernel configuration.

4.2. Optimizing Query Engines Using General-Purpose Compilers

First, we show that low-level, general-purpose compilation frameworks, such as LLVM, are not adequate for efficiently optimizing query engines. To do so, we use LegoBase to generate an *unoptimized* push-style engine with only operator inlining applied, which is then compiled to a final C binary using LLVM. We choose this engine as a starting point since it allows the underlying C compiler to be more effective when optimizing the whole C program (as the presence of procedures may otherwise force the compiler to make conservative decisions or miss optimization potential during compilation).

As shown in Figure 16, the achieved performance is very poor: the unoptimized query engine, LegoBase(Naive/C)–LLVM, is significantly slower for all TPC-H queries, requiring more than 16× the execution time of the optimal LegoBase configuration in most cases. This is because frameworks like LLVM cannot automatically detect all optimization opportunities that we support in LegoBase (as described thus far in this article). This is because either (a) the scope of an optimization is too coarse-grained to be detected by a low-level compiler or (b) the optimization relies on domain-specific knowledge that general-purpose optimizing compilers such as LLVM are not aware of.

In addition, as shown in the same figure, compiling with LLVM does not *always* yield better results compared to using another traditional compiler like GCC[15]. We see that LLVM outperforms GCC for *only* 15 out of 22 queries (by 1.09× on average) while, for the remaining ones, the binary generated by GCC performs better (by 1.03× on average). In general, the performance difference between the two compilers can be significant (e.g. for Q19, there is a 1.58× difference). We also experimented with manually specifying optimizations flags to the two compilers, but this turns out to be a very delicate and complicated task as developers can specify flags which actually make performance worse. We argue that it is instead more beneficial for developers to invest their effort in developing high-level optimizations, like those presented in this article.

4.3. Optimizing Query Engines Using Template Expansion

Next, we compare our approach – which compiles the *entire* query engine and utilizes *query-specific* information – with the compiler of the HyPer database [Neumann 2011]. HyPer performs template expansion through LLVM in order to inline the relational operators of a query executed on a push engine. The results are presented in Figure 17.

We perform this analysis in two steps. First, we generate a query engine that (a) does not utilize any query-specific information and (b) adheres to the implementation rules of the TPC-H workload. Such an engine represents a system where data are loaded *only once*, and all optimizations are applied before any query arrives (as happens with HyPer and any other traditional DBMS). We show that this LegoBase configuration, titled LegoBase(TPC-H/C), has performance competitive to that of the HyPer database system, and that efficient handling of string operations is essential in order to have the performance of our system match that of HyPer. Second, we show that by utilizing query-specific knowledge and performing aggressive materialization and repartition of input relations based on multiple attributes, we can generate a query engine, titled LegoBase(Opt/C), which significantly outperforms existing approaches. Such an engine corresponds to systems that, as discussed previously in Section 3.7, have all queries or data known in advance.

Figure 17 shows that by using the query compiler of HyPer, performance is improved by 6.4× on average compared to DBX. To achieve this performance improvement, HyPer uses a push engine, operator inlining, and data partitioning. In contrast, the TPC-H-compliant configuration of our system, LegoBase(TPC-H/C), which uses the same optimizations, has an average execution time of only 4.4x the one of the DBX system, across all TPC-H queries. The main reason behind this significantly slower performance is, as we mentioned above, the inefficient handling of string operations in LegoBase(TPC-H/C). In this version, LegoBase uses the `strcmp` function (and its variants). In contrast, HyPer uses the SIMD instructions found in modern instructions sets (such as SSE4.2) for efficient string handling [Boncz et al. 2014], a design choice that can lead to significant performance improvement compared to LegoBase(TPC-H/C). To validate this analysis, we use a configuration of our system, called LegoBase(StrDict/C), which additionally applies the string dictionary optimization. This configuration is no longer TPC-H-compliant (as it introduces an auxiliary data structure), but is still does not require query-specific information. We notice that

[15]For this experiment, we use version 4.4.7 of the GCC compiler.

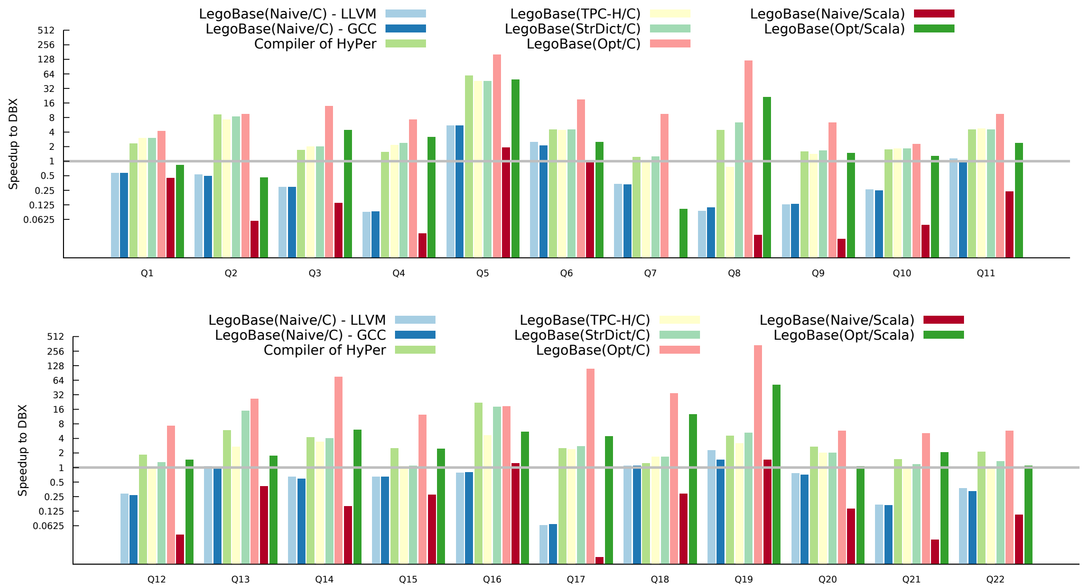

Fig. 17: Performance comparison of various LegoBase configurations (C and Scala programs) with the code generated by the query compiler of [Neumann 2011]. The baseline for all systems is the performance of the DBX commercial database system. The absolute execution times for this figure can be found in Appendix A. This graph also includes the performance of the naive push-engines of Figure 16 for reference.

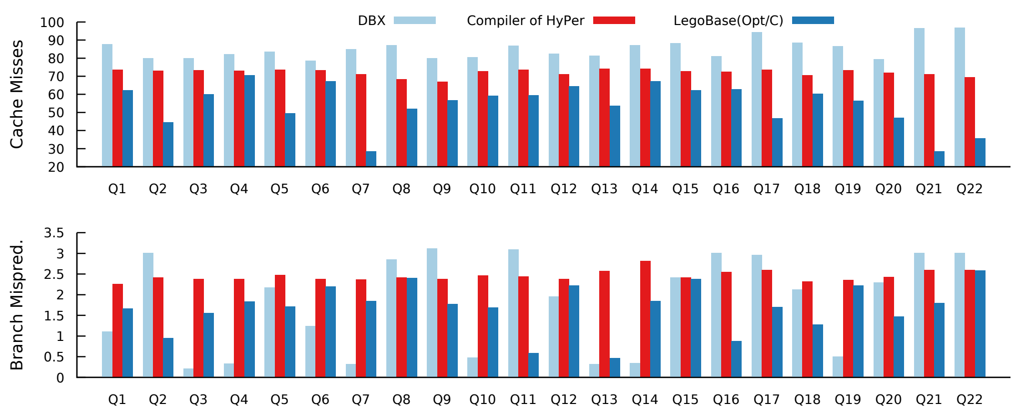

Fig. 18: Percentage of cache misses and branch mispredictions for DBX, HyPer and LegoBase(Opt/C) for all 22 TPC-H queries.

the introduction of this optimization is enough to make LegoBase(StrDict/C) match the performance of HyPer: the two systems have *only* a 1.06× difference in performance.

Second, Figure 17 also shows that by using all other optimizations of LegoBase (as they were presented in Section 3), which are not performed by the query compiler of HyPer, we can get a total 45.4× performance improvement compared to DBX with all optimizations enabled. This is because, for example, LegoBase(Opt/C) uses query-specific information to remove unused relational attributes or hoist out expensive computation (thus reducing memory pressure and decreasing the number of CPU instructions executed) and aggressively repartitions input data on multiple attributes (thus allowing for more efficient join processing). Such optimizations result in improved cache locality and branch prediction, as shown in Figure 18. More specifically, there is an improvement of $1.68\times$ and $1.31\times$ on average for the two metrics, respectively, between DBX and LegoBase. In addition, the maximum, average and minimum difference in the number of CPU instructions executed in HyPer is $3.76\times$, $1.61\times$, and $1.08\times$ that executed in LegoBase. These results prove that the optimized code of LegoBase(Opt/C) is competitive, performance wise, to both traditional database systems and query compilers based on template expansion.

Finally, we note that we plan to investigate even more aggressive and query-specific data-structure optimizations in future work. Such optimizations are definitely feasible, given the easy extensibility of the SC compiler.

4.4. Source-to-Source Compilation from Scala to C

Next, we show that source-to-source compilation from Scala to C is necessary in order to achieve optimal performance in LegoBase. To do so, Figure 17 also presents performance results for both a naive and an optimized Scala query engine, named LegoBase(Naive/Scala) and LegoBase(Opt/Scala), respectively. First, we notice that the optimized generated Scala code is significantly faster than the naive counterpart, by $40.3\times$ (excluding Q8 whose performance is prohibitively slow in the unoptimized Scala version). This shows that extensive optimization of the Scala code is essential in order to achieve high performance. However, we also observe that the performance of the optimized Scala program cannot compete with that of the optimized C code, and is on average $10\times$ slower. Profiling information gathered with the *perf*[16] profiling tool of Linux reveals the following three reasons for this behavior: (a) Scala causes an increase to branch mispredictions, by $1.8\times$ compared to the branch mispredictions in C, (b) The percentage of LLC misses is $1.3\times$ to $2.4\times$ those in Scala, and more importantly, (c) The number of CPU instructions executed in Scala is $6.2\times$ the one executed by the C binary. Of course, these inefficiencies are to a great part due to the Java Virtual Machine and not specific to Scala. Note that the optimized Scala program is competitive to DBX (especially for non-join-intensive queries, e.g. queries that have less than two joins): for 19 out of 22 queries, Scala outperforms the commercial DBX system. This is because we remove all abstractions that incur significant overhead for Scala. For example, the performance of Q18, which builds a large hash map, is improved by $40\times$ when applying the data-structure specialization provided by SC.

4.5. Impact of Individual Compiler Optimizations

In this section, we provide additional information about the performance improvement expected when applying one of the compiler optimizations of LegoBase. These results, illustrated in Figure 19, aim to demonstrate that significant optimization opportunities have been ignored by existing compilation techniques that handle only queries.

To begin with, we can see in this figure that the most important transformation in LegoBase is the data-structure specialization (presented in Sections 3.2.1 and 3.2.2). This form of optimization is not provided by existing approaches, as data structures are typically precompiled in existing database systems. We see that, in general, when data-structure specialization is applied, the generated code has an average performance improvement of $30\times$ (excluding queries Q8 and Q17 where the partitioning

[16]https://perf.wiki.kernel.org/index.php/Main_Page.

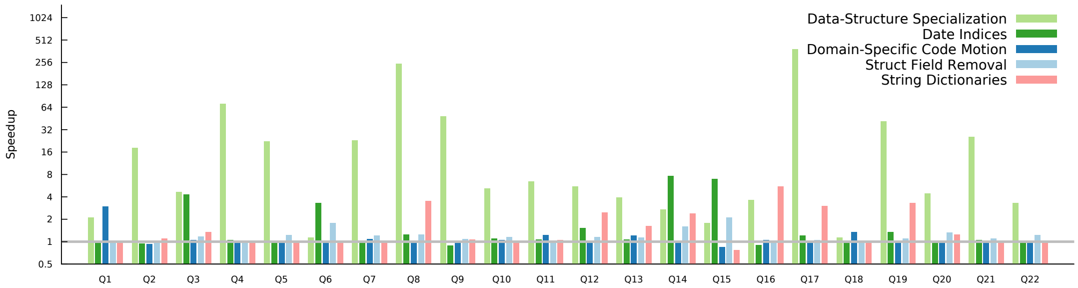

Fig. 19: Impact of different optimizations on query execution time. The baseline is an engine that does not apply this optimization.

optimization facilitates skipping the processing of the majority of the tuples of the input relations). Moreover, we note that the performance improvement is not directly dependent on the number of join operators or input relations in the query plan. For example, join-intensive queries such as Q5, Q7, Q8, Q9, Q21 obtain a speedup of at least 22× when applying this optimization. However, the single-join queries Q4 and Q19 also receive similar performance benefit to that of multi-way join queries. This is because query plans may filter input data early on, thus reducing the need for efficient join data structures. Thus, selectivity information and analysis of the whole query plan are essential for analyzing the potential performance benefit of this optimization. Note that, for similar reasons, date indices (Section 3.2.3) allow to avoid unnecessary tuple processing and thus lead to increased performance for a number of queries.

For the domain-specific code motion and the removal of unused relational attributes optimizations, we observe that they both improve performance, by $1.12\times$ and $1.21\times$, respectively on average for all TPC-H queries. This improvement is not be as pronounced as that of other optimizations of LegoBase (like the one presented above). However, it is important to note that they both significantly reduce memory pressure, thus allowing the freed memory space to be used for other optimizations, such as the partitioning specialization, which in turn provide significant performance improvement. Nevertheless, these two optimizations – which are not provided by previous approaches (since they depend on query-specific knowledge) – can provide considerable performance improvement by themselves for some queries. For example, for TPC-H Q1, performing domain-specific code motion leads to a speedup of $2.96\times$, while the removal of unused attributes gives a speedup of $2.11\times$ for Q15.

Moreover, the same figure evaluates the speedup we gain when using string dictionaries. We observe that for the TPC-H queries that perform a number of expensive string operations, using string dictionaries always leads to improved query execution performance: this speedup ranges from 1.06× to 5.5×, with an average speedup of 2.41×[17]. We also note that the speedup this optimization provides depends on the characteristics of the query. More specifically, if the query contains string operations on scan operators, as is the case with Q8, Q12, Q13, Q16, Q17, and Q19, then this optimization provides a greater performance improvement than when string operations occur in operators appearing later in the query plan. This is because, TPC-H queries typically filter out more tuples as more operators are applied in the query plan. Stated otherwise, operators being executed in the last stages of the query plan do not pro-

[17]The rest of the TPC-H queries (Q1, Q4, Q5, Q6, Q7, Q10, Q11, Q15, Q18, Q21, Q22) either did not have any string operation or the number of these operations on those queries was negligible.

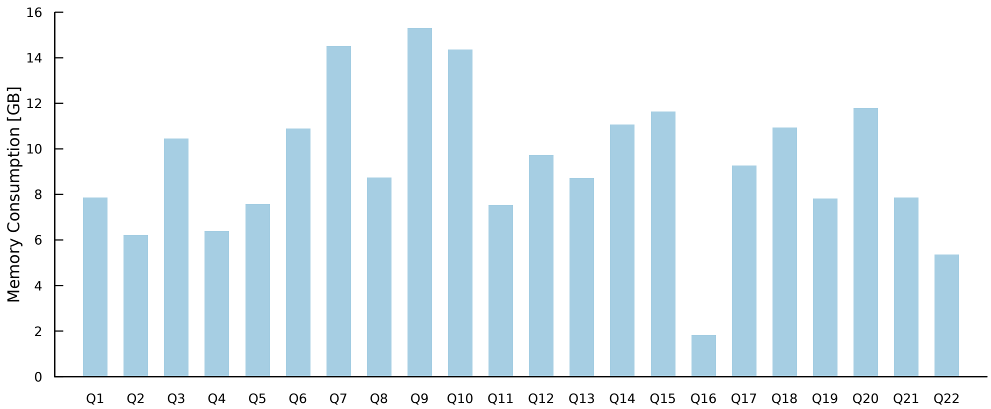

Fig. 20: Memory consumption of LegoBase(Opt/C) for the TPC-H queries.

cess as many tuples as scan operators. Thus, the impact of string operations is more pronounced when such operations take place in scan operators.

It is important to note that using string dictionaries comes at a price. First, this optimization increases the loading time of the query. Second, this optimization requires keeping a dictionary between strings and integer values, a design choice which requires additional memory. This may, in turn, increase memory pressure, possibly causing a drop in performance. However, it is our observation that, based on the individual use case and data characteristics (e.g. number of distinct values of a string attribute), developers can easily detect whether it makes sense performance-wise to use this optimization or not. We also present a more detailed analysis of the memory consumption required by the overall LegoBase system later in this section.

Then, the benefit of applying operator inlining (not shown) varies significantly between different TPC-H queries and ranges from a speedup of $1.07\times$ up to $19.5\times$, with an average performance improvement of $3.96\times$. The speedup gained from applying this optimization depends on the complexity of the execution path of a query. This is a hard metric to visualize, as the improvement depends not only on *how many* operators are used but also on their type, their position in the overall query plan and how much each of them affects branch prediction and cache locality. For instance, queries Q5, Q7 and Q9 have the same number of operators, but the performance improvement gained varies significantly, by 10.4×, 1.4× and 7.5×, respectively. In addition, it is our observation that the benefit of inlining depends on which operators are being inlined. This is an important observation, as for very large queries, the compiler may have to choose which operators to inline (e.g. to avoid the code not fitting in the instruction cache). In general, when such cases appear, we believe that the compiler framework should merit inlining joins instead of simpler operators (e.g. scans or aggregations).

Finally, for the column layout optimization (not shown), the performance improvement is generally proportional to the percentage of attributes in the input relations that are actually used. This is expected as the benefits of the column layout are evident when this layout can "skip" loading into memory a number of unused attributes, thus significantly reducing cache misses. Synthetic queries on TPC-H data referencing 100% of the attributes show that, in this case, the column layout actually yields no benefit, and it is slightly worse than the row layout. Actual TPC-H queries reference 24% - 68% of the attributes and, for this range, the optimization gives a 2.5× to 1.05× improvement, which degrades as more attributes are referenced.

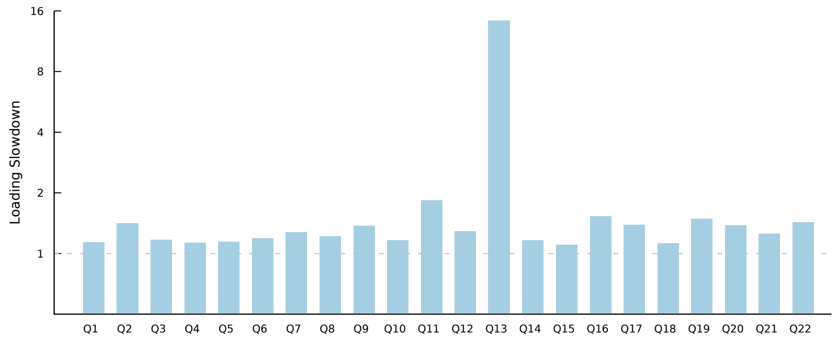

Fig. 21: Slowdown of input data loading occurring from applying all LegoBase optimizations to the C programs of the TPC-H workload (scaling factor 8).

4.6. Memory Consumption and Overhead on Input Data Loading

Figure 20 shows the memory consumption for all TPC-H queries. We use Valgrind for memory profiling as well as a custom memory profiler, while the JVM is always first warmed up. We make the following observations. First, the allocated memory is at most twice the size of the input data for all TPC-H queries (16GB of memory for 8GB of input data for all relations), while the *average* memory consumption is only $1.16\times$ the total size of the input relations. These results suggest that our approach is usable in practice, as even for complicated, multi-way join queries the memory used remains relatively small. The additional memory requirements come as a result of the fact that LegoBase aggressively repartitions input data in many different ways (as was described in Section 3.2) in order to achieve optimal performance. Second, when all optimizations are enabled, LegoBase consumes less memory than the total size of the input data, for a number of queries. For instance, Q16 consumes merely 2GB for all necessary data structures. This behavior is a result of removing unused attributes from relational tables when executing a query. In general, it is our observation that memory consumption grows linearly with the scaling factor of the TPC-H workload.

In addition, we have mentioned before that applying our compiler optimizations can lead to an increase in the loading time of the input data. Figure 21 presents the total slowdown on input data loading when applying all LegoBase optimizations to the generated C programs (LegoBase(Opt/C)) compared to the loading time of the unoptimized C programs (LegoBase(Naive/C)). We observe that the total time spent on data loading, across all queries and with all optimizations applied, is *not* (excluding Q13 which applies the word-tokenizing string dictionary) more than $1.5\times$ that of the unoptimized, push-style generated C code. This means that while our optimizations lead to significant performance improvement, they do not affect the loading time of input data significantly (there is an average slowdown of $1.88\times$ including Q13). Based on these observations, as well as the absolute loading times presented in Appendix A, we can see that the additional overhead of our optimizations is not prohibitive: it takes in average less than a minute for LegoBase to load the 8GB TPC-H dataset, repartition the data, and build all necessary auxiliary data structures for efficient query processing.

4.7. Productivity Evaluation

An important point of this article is that the performance of query engines can be improved without much programming effort. Next, we present the productivity/performance evaluation of our system, which is summarized in Table IV.

Data-Structure Partitioning	505
Automatic Inference of Date Indices	318
Memory Allocation Hoisting	186
Column Store Transformer	184
Constant-Size Array to Local Vars	125
Flattening Nested Structs	118
Horizontal Fusion	152
Scala Constructs to C Transformer	793
Scala Collections to GLib Transformer	411
Scala Scanner Class to mmap Transformer	90
Other miscellaneous optimizations	≈ 200
Total	3082

Table IV: Lines of code of several transformations in LegoBase with the SC compiler.

We observe three things. First, by programming at the high level, we can provide a fully functional system with a small amount of effort. Less development time was spent on debugging the system, thus allowing us to focus on developing new useful optimizations. Development of the LegoBase query engine alongside the domain-specific optimizations required, including debugging time, eight months for only two programmers. However, the majority of this effort was invested in building the new optimizing compiler SC (27K LOC) rather than developing the basic, unoptimized, query engine itself (1K LOC).

Second, each optimization requires only a few hundred lines of high-level code to provide significant performance improvement. More specifically, for $\approx$3000 LOC in total, LegoBase is improved by 45.4$\times$ compared to the performance of DBX, as we described previously. Source-to-source compilation is critical to achieving this behavior, as the combined size of the operators and optimizations of LegoBase is around 40 times less than the generated code size for all 22 TPC-H queries written in C.

Finally, from Table IV it becomes clear that new transformations can be introduced in SC with relative small programming effort. This becomes evident when one considers complicated transformations like those of Automatic Index Inference and Horizontal Fusion[18] which can both be coded for merely $\approx$500 lines of code. We also observe that around half of the code-base required to be introduced in SC concerns converting Scala code to C. However, this is a naïve task to be performed by SC developers, as it usually results in a one-to-one translation between Scala and C constructs. More importantly, this is a task that is required to be performed only *once* for each language construct, and it needs to be extended *only* as new constructs are introduced in SC (e.g. those required for custom data types and operations on those types).

4.8. Compilation Overheads

Finally, we analyze the compilation time for the optimized C programs of LegoBase(Opt/C) for all TPC-H queries. Our results are presented in Figure 22, where the y-axis corresponds to the time to (a) optimize a query and generate the C code with SC, and, (b) the time CLang requires before producing the final C executable.

[18]To perform a decent loop fusion, the short-cut deforestation is not sufficient. Such techniques only provide *vertical* loop fusion, in which one loop uses the result produced by another loop. However, in order to perform further optimizations one requires to perform *horizontal* loop fusion, in which different loops iterating over the same range are fused into one loop [Beeri and Kornatzky 1990; Goldberg and Paige 1984]. A decent loop fusion is still an open topic in the PL community [Svenningsson 2002; Coutts et al. 2007; Gill et al. 1993].

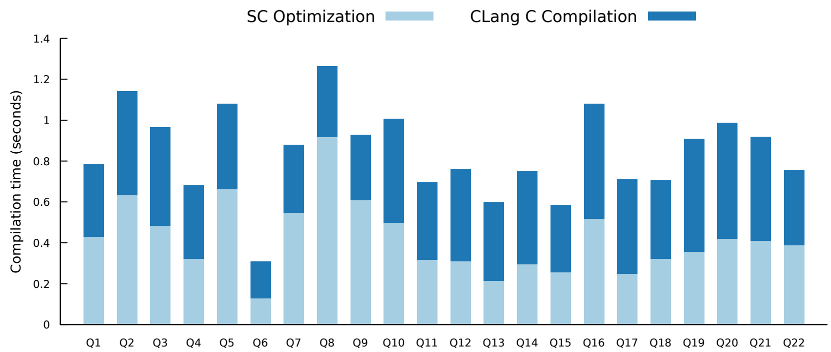

Fig. 22: Compilation time (in seconds) of all LegoBase(Opt/C) programs.

We see that, in general, all TPC-H queries require less than 1.2 seconds of compilation time. We argue that this is an acceptable compilation overhead, especially for analytical queries like those in TPC-H that are typically known in advance and which process huge amounts of data. In this case, a compilation overhead of some seconds is negligible compared to the total execution time. This result proves that our approach is usable in practice for quickly compiling *entire* query engines written using high-level programming languages. To achieve these results, special effort was made so that the SC compiler can quickly optimize input programs. More specifically, our progressive lowering approach allows for quick application of optimizations, as most of our optimizations operate on a relatively small number of language constructs, thus making it easy to quickly detect which parts of the input program should be modified at each transformation step, while the rest of them can be quickly skipped. In addition, we observe that the CLang C compilation time can be significant. This is because, by applying all the domain-specific optimizations of LegoBase to an input query, we get an increase in the total program size that CLang receives from SC.

Finally, we note that if we generate Scala code instead of C, then the time required for compiling the final optimized Scala programs is 7.2× that of compiling the C programs with LLVM. To some extent this is expected as calling the Scala compiler is a heavyweight process: for every query compiled there is significant startup overhead for loading the necessary Scala and Java libraries. By just optimizing a Scala program using optimizations written in the same level of abstraction, our architecture allows us to avoid these overheads, providing a much more lightweight compilation process.

5. RELATED WORK

We outline related work in five areas: (a) Previous compilation approaches, (b) Frameworks for applying intra-operator optimizations, (c) Orthogonal techniques to speed up query processing, (d) a brief summary of work on Domain Specific Compilation in the Programming, and, finally, (e) a comparison with a previous realization of the abstraction without regret vision Languages (PL) community, a field of study that closely relates to ours. We briefly discuss these areas below.

Previous Compilation Approaches. Historically, System R [Chamberlin et al. 1981] first proposed code generation for query optimization. However, the Volcano iterator model eventually dominated over compilation, since code generation was very expensive to maintain. The Daytona system [Greer 1999] revisited compilation in the late nineties; however, it heavily relied on the operating system for functionality that is traditionally provided by the DBMS itself, like buffering.

The shift towards pure *in-memory* computation in databases, evident in the space of data analytics and transaction processing has lead developers to revisit compilation. The reason is that, as more and more data is put in memory, query performance is increasingly determined by the effective throughput of the CPU. In this context, compilation strategies aim to remove unnecessary CPU overhead. Examples of industrial systems in the area since the mid-2000s include SAP HANA [Färber et al. 2012], VoltDB [Stonebraker et al. 2007; Kallman et al. 2008] and Oracle's TimesTen [Oracle Corporation 2006]. In the academic context, interest in query compilation has also been renewed since 2009 and continues to grow [Rao et al. 2006; Zane et al. 2008; Ahmad and Koch 2009; Grust et al. 2009; Krikellas et al. 2010; Neumann 2011; Koch 2013; Crotty et al. 2014; Nagel et al. 2014; Viglas et al. 2014; Armbrust et al. 2015; Crotty et al. 2015; Goel et al. 2015]. We briefly discuss some of these systems next.

Rao et al. propose to remove the overhead of virtual functions in the Volcano iterator model by using a compiled execution engine built on top of the Java Virtual Machine (JVM) [Rao et al. 2006]. The HIQUE system takes a step further and completely eliminates the Volcano iterator model in the generated code [Krikellas et al. 2010]. It does so by translating the algebraic representation to C++ code using templates. In addition, Zane et al. have shown how compilation can also be used to additionally improve operator internals [Zane et al. 2008].

The query compiler of the HyPer database system also uses query compilation, as described in [Neumann 2011]. This work targets minimizing the CPU overhead of the Volcano operator model while maintaining low compilation times. The authors use a mixed LLVM/C++ execution engine where the algebraic representation of the operators is first translated to low-level LLVM code, while the complex part of the database (e.g. management of data structures and memory allocation) is still *precompiled* C++ code called periodically from the LLVM code whenever needed. Two basic optimizations are presented: operator inlining and reversing the data flow (to a push engine).

All these works aim to improve database systems by removing unnecessary abstraction overheads. However, these *template-based* approaches require writing low-level code which is hard to maintain and extend. This fact significantly limits their applicability. In contrast, our approach advocates a new methodology for programming query engines where the query engine and its optimizations are written in a *high-level* language. This provides a programmer-friendly way to express optimizations and allows extending the scope of optimization to cover the whole query engine. Finally, in contrast to previous work, we separate the optimization and code generation phases. Even though [Neumann 2011] argues that optimizations should happen completely before code generation (e.g. in the algebraic representation), there exist many optimization opportunities that occur only *after* one considers the complete generated code, e.g. after operator inlining. Our compiler can detect such optimizations, thus providing additional performance improvement over existing techniques.

Frameworks for applying intra-operator optimizations. There has recently been extensive work on how to specialize the code of query operators in a systematic way by using an approach called Micro-Specialization [Zhang et al. 2012a; Zhang et al. 2012b; 2012c]. In this line of work, the authors propose a framework to encode DBMS-specific intra-operator optimizations, like unrolling loops and removing if conditions, as precompiled templates in an extensible way. All these optimizations are performed by default by the SC compiler in LegoBase. However, in contrast to our work, there are two main limitations in Micro-Specialization. First, the low-level nature of the approach makes the development process very time-consuming: it can take days to code a single intra-operator optimization [Zhang et al. 2012a]. Such optimizations are very fine-grained, and it should be possible to implement them quickly: for the same amount of time we are able to provide much more coarse-grained optimizations in LegoBase. Second, the optimizations are limited to those that can be statically determined by examining the DBMS code and cannot be changed at runtime. Our architecture maintains all the benefits of Micro-Specialization, while it is not affected by these two limitations.

Techniques to speed up query processing. There are many works that aim to speed up query processing in general, by focusing on improving the way data is processed rather than individual operators. Examples of such works include block-wise processing [Padmanabhan et al. 2001], vectorized execution [Sompolski et al. 2011], compression techniques to provide constant-time query processing [Raman et al. 2008] or a combination of the above along with a column-oriented data layout [Manegold et al. 2009]. These approaches are orthogonal to this work as our framework provides a high-level framework for encoding *all* such optimizations in a user-friendly way (e.g. we present the transition from row to column data layout in Section 3.3).

Domain-specific compilation, which admits domain-specific optimizations, is a topic of great current interest in multiple research communities. Once one limits the domain or language, program analysis can be more successful. More powerful and global transformations then become possible, yielding speedups that cannot be expected from classical compilers for general purpose languages. To this end, multiple frameworks and research prototypes [Hudak 1996; Faith et al. 1997; van Deursen et al. 2000; Kennedy et al. 2005; Rompf and Odersky 2010; Ackermann et al. 2012; Lee et al. 2011; Jovanović et al. 2014; Humer et al. 2014], have been proposed to easily introduce and perform domain-specific compilation and optimization for systems. Of interest is the observation that domain-specificity has already benefited query optimization tremendously: Relational algebra is a domain-specific language, and yields readily available associativity properties that are the foundation of query optimization. Optimizing compilers can combine the performance benefits of classical interpretation-based query engines with the benefits of abstraction and indirection elimination by compilers. Finally, OCAS [Klonatos et al. 2013] has been developed within the context of domain-specific synthesis and attempts to automatically generate optimized out-of-core algorithms for a particular target memory hierarchy.

Previous realization of the abstraction without regret vision. We have previously realized this vision for query engines in [Klonatos et al. 2014]. In this article, we provide a from scratch implementation of the vision using a *new* optimizing compiler, called SC, developed specifically to handle the optimization needs of large-scale software systems. We also present a detailed analysis of the compiler interfaces of SC as well as a significantly more thorough list of the optimizations supported by the LegoBase system in order to demonstrate the ease-of-use of our compiler framework for optimizing database components that differ significantly in granularity and scope of operations. Finally, we provide a more extensive evaluation where, along with a renewed analysis of the previous results, we also evaluate three additional query engine configurations. We do so in order to compare as fairly as possible the performance of our system with that of previous work.

6. CONCLUSIONS

LegoBase is a new analytical database system currently under development. In this article, we presented the current prototype of the query execution subsystem of

LegoBase. Our approach suggests using high-level programming languages for DBMS development without having to pay the associated abstraction penalty. This vision has been previously called *abstraction without regret*. The key technique to admit this productivity/efficiency combination is to apply generative programming and source-to-source compile the high-level Scala code to efficient low-level C code. We demonstrate how state-of-the-art compiler technology allows developers to express database-specific optimizations naturally at a high level as a library and use it to optimize the *entire* query engine. In LegoBase, programmers need to develop just a few hundred lines of *high-level* code to implement techniques and optimizations that result in significant performance improvement. All these properties are very hard to achieve with existing compilers that handle *only* queries and which are based on template expansion. Our experiments show that LegoBase significantly outperforms both a commercial in-memory database system as well as an existing query compiler.

APPENDIX

A. ABSOLUTE EXECUTION TIMES

For completeness, the following tables present the *absolute performance results* of all evaluated systems and metrics in this article.

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
DBX	1790	396	1528	960	19879	882	969	2172	3346	985	461
Compiler of HyPer	779	43	892	622	338	198	798	493	2139	565	102
LegoBase (Naive/C) – LLVM	3140	755	5232	10742	3627	357	2901	23161	26203	3836	409
LegoBase (Naive/C) – GCC	3140	801	5204	10624	3652	423	2949	19961	25884	3966	445
LegoBase (Naive/Scala)	3972	6910	11118	30103	10307	874	114677	72587	137369	20353	1958
LegoBase(TPC-H/C)	593	55	767	445	440	199	975	2871	2387	546	98
LegoBase(StrDict/C)	592	47	759	402	439	197	781	346	2027	544	103
LegoBase(Opt/C)	426	42	110	134	126	47	104	18	530	439	49
LegoBase(Opt/Scala)	2174	871	352	306	413	356	9496	104	2296	775	197

System	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
DBX	881	13593	823	578	12793	1224	4535	6432	744	1977	447
Compiler of HyPer	485	2333	197	229	590	490	3682	1421	277	1321	212
LegoBase (Naive/C) – LLVM	3037	12794	1289	889	16362	18893	4135	2810	974	11648	1187
LegoBase (Naive/C) – GCC	3286	13149	1398	899	16159	18410	4174	4460	1055	11848	1396
LegoBase (Naive/Scala)	21735	33403	5163	2093	10568	86953	15798	4470	5301	61712	4207
LegoBase(TPC-H/C)	891	5106	244	550	2774	513	2725	2020	370	1992	453
LegoBase(StrDict/C)	688	910	204	535	702	445	2735	1222	370	1706	333
LegoBase(Opt/C)	120	516	11	46	695	11	133	19	130	388	79
LegoBase(Opt/Scala)	604	7743	136	234	2341	274	355	125	700	955	406

Table V: Execution times (in milliseconds) of Figure 16 and Figure 17. The various configurations of LegoBase are explained in more detail in Table III of this article.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
LegoBase (Naive/C) – LLVM	3140	755	5232	10742	3627	357	2901	23161	26203	3836	409
+Struct Field Removal	3104	734	4480	10346	2983	202	2394	18707	24125	3323	403
+Domain-Specific Code Motion	1047	794	4283	10435	2902	196	2203	18507	23854	3177	332
+Data-Structure Specialization	497	44	918	148	130	172	96	75	498	610	52
+Date Indices	497	47	213	140	131	52	96	60	568	553	49
+String Dictionaries	497	43	158	140	130	51	94	17	533	552	47
LegoBase(Opt/C)	426	42	110	134	126	47	104	18	530	439	49

	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
LegoBase (Naive/C) – LLVM	3037	12794	1289	889	16362	18893	4135	2810	974	11648	1187
+Struct Field Removal	2631	11291	812	420	16068	17953	4070	2550	736	10647	970
+Domain-Specific Code Motion	2553	9415	786	495	15251	18063	3050	2568	742	10386	985
+Data-Structure Specialization	467	2389	291	277	4243	47	2709	62	168	410	300
+Date Indices	308	2233	38	40	4737	39	2718	46	168	392	291
+String Dictionaries	125	1379	16	52	860	13	2730	20	136	389	299
LegoBase(Opt/C)	120	516	11	46	695	11	133	19	130	388	79

Table VI: Execution times (in milliseconds) of TPC-H queries with individual optimizations applied (as shown in Figure 19 of this article). Each listed optimization is applied additionally to the set of optimizations applied in the system specified above it.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Memory Consumption	7.86	6.20	10.45	6.39	7.56	10.88	14.51	8.72	15.30	14.35	7.53
Loading Time (No opt.)	34	7	44	42	43	33	43	46	45	44	5
Loading Time (All opt.)	38	10	52	47	49	39	55	56	61	52	10
SC Optimization	429	633	482	323	663	128	547	918	608	498	317
CLang C Compilation	354	509	482	359	418	179	332	346	320	507	378

	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Memory Consumption	9.73	8.72	11.06	11.64	1.81	9.26	10.92	7.81	11.77	7.86	5.36
Loading Time (No opt.)	41	9	36	34	7	34	42	35	38	41	9
Loading Time (All opt.)	53	135	42	38	10	47	47	52	53	52	13
SC Optimization	310	215	295	255	518	248	321	357	420	411	389
CLang C Compilation	449	386	454	329	563	461	382	552	566	507	365

Table VII: Memory consumption in GB, input data loading time in seconds, and optimization/compilation time in milliseconds as shown in Figure 20, Figure 21, and, Figure 22 of this article, respectively.

B. CODE SNIPPET FOR THE PARTITIONING TRANSFORMER

Next, we present a portion of the data partitioning transformation, an explanation of which was given in Section 3.2.1. This code corresponds to the join processing for equi-joins (and not the actual partitioning of input data), but similar rules are employed for other join types as well. The aim of this snippet is to demonstrate the ease-of-use of the SC compiler.

```
/* A transformer for partitioning and indexing MultiMap data-structures. As a result, this
   transformation converts MultiMap operations to native Array operations. */
class HashTablePartitioning extends RuleBasedTransformer {
  val allMaps = mutable.Set[Any]()
  var currentWhileLoop: While = _

 /* ---- ANALYSIS PHASE ---- */
 /* Gathers all MultiMap symbols which are holding a record as their value */
 analysis += statement {
  case sym -> code"new MultiMap[_, $v]" if isRecord(v) => allMaps += sym
 }
 /* Keeps the closest while loop in scope (used in the next analysis rule)*/
 analysis += rule {
  case whileLoop @ code"while($cond) $body" => currentWhileLoop = whileLoop
 }
 /* Maintain necessary information for the left relation */
 analysis += rule {
  case code"($mm: MultiMap[_,_]).addBinding(struct_field($struct, $fieldName),$value)" =>
    mm.attributes("addBindingLoop") = currentWhileLoop
 }
 /* Maintain necessary information for the right relation */
 analysis += rule {
  case code"($mm : MultiMap[_, _]).get(struct_field($struct, $fieldName))" =>
    mm.attributes("partitioningStruct") = struct
    mm.attributes("partitioningFieldName") = fieldName
 }

 /* ---- REWRITING PHASE ---- */
 def shouldBePartitioned(mm: Multimap[Any, Any]) = allMaps.contains(mm)

 /* If the left relation should be partitioned, then remove the 'addBinding' and 'get'
    function calls for this multimap, as well as any related loops. Notice that there is
    no need to remove the multimap itself, as DCE will do so once all of its dependent
    operations have been removed.*/
 rewrite += remove {
  case code"($mm: MultiMap[Any, Any]).addBinding($elem, $value)" if
    shouldBePartitioned(mm) =>
 }
 rewrite += remove {
  case code"($mm: MultiMap[Any, Any]).get($elem)" if shouldBePartitioned(mm) =>
 }
 rewrite += remove {
  case node @ code"while($cond) $body" if allMaps.exists({
    case mm => shouldBePartitioned(mm) && mm.attributes("addBindingLoop") == node
  }) =>
 }
 /* If a MultiMap should be partitioned, instead of the construction of that MultiMap
    object, use the corresponding partitioned array constructed during data-loading.
    This can be an 1D or 2D array, depending on the properties and relationships of the
    primary and foreign keys of that table (described in Section 3.2.1 in more detail). */
 rewrite += statement {
  case sym -> (code"new MultiMap[_, _]") if shouldBePartitioned(sym) =>
```

```
    getPartitionedArray(sym)
  }

  /* Rewrites the logic for extracting matching elements of the left relation (initially
     using the HashMap), inside the loop iterating over the right relation. */
  rewrite += rule {
    case code"($mm:MultiMap[_,_]).get($elem).get.foreach($f)" if shouldBePartitioned(mm) =>{
      val leftArray = transformed(mm)
      val hashElem = struct_field(mm.attributes("partitioningStruct"),
                                  mm.attributes("partitioningField"))
      val leftBucket = leftArray(hashElem)
      /* In what follows, we iterate over the elements of the bucket, even though the
         partitioned array may be an 1D-array as discussed in Section 3.1.2. There is
         another optimization in the pipeline which flattens the for loop of this case. */
      for(e <- leftBucket) {
        /* Function f corresponds to checking the join condition and creating the join
           output. This functionality remains the same, thus, we can simply inline the
           related code here as follows */
        ${f(e)}
      }
    }
  /* For a partitioned relation, there is no need to check for emptiness, due to primary /
     foreign key relationship. The if (true) is later removed by another optimization. */
  rewrite += rule {
    case code"($mm: MultiMap[Any, Any]).get($elem).nonEmpty" if shouldBePartitioned(mm) =>
      true
  }
}
```

REFERENCES

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 967–980. DOI:http://dx.doi.org/10.1145/1376616.1376712

Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. 2012. Jet: An Embedded DSL for High Performance Big Data Processing. In *International Workshop on End-to-end Management of Big Data (BigData 2012)*. http://infoscience.epfl.ch/record/181673/files/paper.pdf

Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL Compiler for High-performance Delta Processing in Main-Memory Databases. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1566–1569. DOI:http://dx.doi.org/10.14778/1687553.1687592

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 169–180. http://research.cs.wisc.edu/multifacet/papers/vldb01_pax.pdf.

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark *(SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394. DOI:http://dx.doi.org/10.1145/2723372.2742797

Catriel Beeri and Yoram Kornatzky. 1990. Algebraic Optimization of Object-Oriented Query Languages. In *ICDT '90*, Serge Abiteboul and Paris C. Kanellakis (Eds.). Lecture Notes in Computer Science, Vol. 470. Springer Berlin Heidelberg, Berlin, Heidelberg, 72–88. DOI:http://dx.doi.org/10.1007/3-540-53507-1_71

Peter Boncz, Thomas Neumann, and Orri Erling. 2014. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*. Springer International Publishing, Cham, 61–76. DOI:http://dx.doi.org/10.1007/978-3-319-04936-6_5

Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1981. A History and Evaluation of System R. *Comm. ACM* 24, 10 (1981), 632–646. DOI:http://dx.doi.org/10.1145/358769.358784

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All.. In *ICFP* (2007-11-06), Ralf Hinze and Norman Ramsey (Eds.). ACM, New York, NY, USA, 315–326. DOI:http://dx.doi.org/10.1145/1291151.1291199

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1466–1477. DOI:http://dx.doi.org/10.14778/2824032.2824045

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Cetintemel, and Stan Zdonik. 2014. Tupleware: Redefining Modern Analytics. *CoRR* abs/1406.6667 (2014). http://arxiv.org/abs/1406.6667

Rickard E. Faith, Lars S. Nyland, and Jan F. Prins. 1997. KHEPERA: A System for Rapid Implementation of Domain Specific Languages. In *Proceedings of the 1997 Conference on Domain-Specific Languages (DSL' 97)*. USENIX Association, Berkeley, CA, USA, 19–19. https://www.usenix.org/legacy/publications/library/proceedings/dsl97/full_papers/faith/faith.pdf

Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database – Data Management for Modern Business Applications. *SIGMOD Record* 40, 4 (2012), 45–51. DOI:http://dx.doi.org/10.1145/2094114.2094126

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 223–232. DOI:http://dx.doi.org/10.1145/165180.165214

Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards Scalable Real-time Analytics: An Architecture for Scale-out of OLxP Workloads. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1716–1727. DOI:http://dx.doi.org/10.14778/2824032.2824069

Allen Goldberg and Robert Paige. 1984. Stream Processing. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 53–62. DOI:http://dx.doi.org/10.1145/800055.802021

Goetz Graefe. 1994. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (Feb 1994), 120–135. DOI:http://dx.doi.org/10.1109/69.273032

Rick Greer. 1999. Daytona And The Fourth-Generation Language Cymbal. In *the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. ACM, New York, NY, USA, 525–526. DOI:http://dx.doi.org/10.1145/304182.304242

Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. 2009. FERRY – Database-supported Program Execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 1063–1066. DOI:http://dx.doi.org/10.1145/1559845.1559982

Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. 2006. Performance Tradeoffs in Read-optimized Databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 487–498. http://dl.acm.org/citation.cfm?id=1182635.1164170

Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D. Matsakis. 2013. GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language. In *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW '13)*. IEEE Computer Society, Washington, DC, USA, 315–324. DOI:http://dx.doi.org/10.1109/IPDPSW.2013.173

Paul Hudak. 1996. Building Domain-specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (Dec. 1996). DOI:http://dx.doi.org/10.1145/242224.242477

Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A Domain-specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. ACM, New York, NY, USA, 123–132. DOI:http://dx.doi.org/10.1145/2658761.2658776

Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (2007), 37–49. DOI:http://dx.doi.org/10.1145/1243418.1243424

Vojin Jovanović, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-Yang: Concealing the Deep Embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. ACM, New York, NY, USA, 73–82. DOI:http://dx.doi.org/10.1145/2658761.2658771

Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB* 1, 2 (2008), 1496–1499. http://dl.acm.org/citation.cfm?id=1454159.1454211

Ken Kennedy, Bradley Broom, Arun Chauhan, Robert J. Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. 2005. Telescoping Languages: A System for Automatic Generation of Domain Languages. *Proc. IEEE* 93, 2 (2005), 387–408. DOI:http://dx.doi.org/10.1109/JPROC.2004.840447

Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-Level Language. *PVLDB* 7, 10 (2014), 853–864.

Yannis Klonatos, Andres Nötzli, Andrej Spielmann, Christoph Koch, and Victor Kuncak. 2013. Automatic Synthesis of Out-of-core Algorithms. In *the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, 133–144. DOI:http://dx.doi.org/10.1145/2463676.2465334

Christoph Koch. 2013. Abstraction without regret in data management systems.. In *CIDR*. www.cidrdb.org. http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper149.pdf

Christoph Koch. 2014. Abstraction Without Regret in Database Systems Building: a Manifesto. *IEEE Data Eng. Bull.* 37, 1 (2014), 70–79. http://sites.computer.org/debull/A14mar/p70.pdf

Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *Proceedings of the 26th International Conference on Data Engineering (ICDE '10)*. IEEE Computer Society, Washington, DC, USA, 613–624. DOI:http://dx.doi.org/10.1109/ICDE.2010.5447892

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–86. http://dl.acm.org/citation.cfm?id=977395.977673

HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro* 31, 5 (Sept. 2011), 42–53. DOI:http://dx.doi.org/10.1109/MM.2011.68

Stefan Manegold, Martin L. Kersten, and Peter Boncz. 2009. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB* 2, 2 (2009), 1648–1653. DOI:http://dx.doi.org/10.14778/1687553.1687618

Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. 2014. Code Generation for Efficient Query Processing in Managed Runtimes. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1095–1106. DOI:http://dx.doi.org/10.14778/2732977.2732984

Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550. http://www.vldb.org/pvldb/vol4/p539-neumann.pdf

Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. In *the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 41–57. DOI:http://dx.doi.org/10.1145/1094811.1094815

Oracle Corporation. 2006. TimesTen In-Memory Database Architectural Overview. (2006). http://download.oracle.com/otn_hosted_doc/timesten/603/TimesTen-Documentation/arch.pdf.

Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. 2001. Block oriented processing of Relational Database operations in modern Computer Architectures. In *Proceedings of the 17th International Conference on Data Engineering (ICDE '01)*. IEEE Computer Society, Washington, DC, USA, 567–574. DOI:http://dx.doi.org/10.1109/ICDE.2001.914871

Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. 2008. Constant-Time Query Processing. In *Proceedings of the 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, Washington, DC, USA, 60–69. DOI:http://dx.doi.org/10.1109/ICDE.2008.4497414

Jun Rao, Hamid Pirahesh, C. Mohan, and Guy Lohman. 2006. Compiled Query Execution Engine using JVM. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, Washington, DC, USA, 23–34. DOI:http://dx.doi.org/10.1109/ICDE.2006.40

Tiark Rompf. 2012. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. Ph.D. Dissertation. École Polytechnique Fédérale de Lausanne (EPFL). DOI:http://dx.doi.org/10.5075/epfl-thesis-5456

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *the ninth international conference on Generative programming and component engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136. DOI:http://dx.doi.org/10.1145/1868294.1868314

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-level Programs: New Directions for Extensible Compilers based on Staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 497–510. DOI:http://dx.doi.org/10.1145/2429069.2429128

Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. Compilation in Query Execution. In *the Seventh International Workshop on Data Management on New Hardware (DaMoN '11)*. ACM, New York, NY, USA, 33–40. DOI:http://dx.doi.org/10.1145/1995441.1995446

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-oriented DBMS. In *the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, 553–564. http://dl.acm.org/citation.cfm?id=1083592.1083658

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era: (it's time for a complete rewrite). In *the 33rd international conference on Very large data bases (VLDB '07)*. VLDB Endowment, 1150–1160. http://dl.acm.org/citation.cfm?id=1325851.1325981

Arvind K Sujeeth, Austin Gibbons, Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*. ACM, New York, NY, USA, 145–154. DOI:http://dx.doi.org/10.1145/2517208.2517220

Eijiro Sumii and Naoki Kobayashi. 2001. A Hybrid Approach to Online and Offline Partial Evaluation. *Higher Order Symbol. Comput.* 14, 2-3 (Sept. 2001), 101–142. DOI:http://dx.doi.org/10.1023/A:1012984529382

Josef Svenningsson. 2002. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 124–132. DOI:http://dx.doi.org/10.1145/581478.581491

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. DOI:http://dx.doi.org/10.1016/S0304-3975(00)00053-0

The GNOME Project. 2013. GLib: Library Package for low-level data structures in C – The Reference Manual. (2013). https://developer.gnome.org/glib/2.38/.

Transaction Processing Performance Council. 1999. TPC-H, an ad-hoc, decision support benchmark. (1999). http://www.tpc.org/tpch

Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.* 35, 6 (June 2000), 26–36. DOI:http://dx.doi.org/10.1145/352029.352035

Stratis Viglas, Gavin M. Bierman, and Fabian Nagel. 2014. Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes. *IEEE Data Eng. Bull.* 37, 1 (2014), 12–21. http://sites.computer.org/debull/A14mar/p12.pdf

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI' 08)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=1855741.1855742

Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P. Wright. 2006. On Incremental File System Development. *Transactions on Storage* 2, 2 (May 2006), 161–196. DOI:http://dx.doi.org/10.1145/1149976.1149979

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud' 10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

Barry M. Zane, James P. Ballard, Foster D. Hinshaw, Dana A. Kirkpatrick, and Less Premanand Yerabothu. 2008. Optimized SQL Code Generation (US Patent 7430549 B2). WO Patent App. US 10/886,011. (Sept. 2008). http://www.google.ch/patents/US7430549

Rui Zhang, Saumya Debray, and Richard T. Snodgrass. 2012a. Micro-Specialization: Dynamic Code Specialization of Database Management Systems. In *the Tenth ACM International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 63–73. DOI:http://dx.doi.org/10.1145/2259016.2259025

Rui Zhang, Richard T. Snodgrass, and Saumya Debray. 2012b. Application of Micro-specialization to Query Evaluation Operators. In *Proceedings of the 28th International Conference on Data Engineering Workshops (ICDEW '12)*. IEEE Computer Society, Washington, DC, USA, 315–321. DOI:http://dx.doi.org/10.1109/ICDEW.2012.43

Rui Zhang, Richard T. Snodgrass, and Saumya Debray. 2012c. Micro-Specialization in DBMSes. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 690–701. DOI:http://dx.doi.org/10.1109/ICDE.2012.110