Building Efficient Query Engines in a High-Level Language

Yannis Klonatos [1]*, Christoph Koch [1], Tiark Rompf [1,2], and Hassan Chafi [2]

[1] École Polytechnique Fédérale de Lausanne (EPFL) {firstname}.{lastname}@epfl.ch
[2] Oracle Labs {firstname}.{lastname}@oracle.com

ABSTRACT

In this paper we advocate that it is time for a radical rethinking of database systems design. Developers should be able to leverage high-level programming languages without having to pay a price in efficiency. To realize our vision of *abstraction without regret*, 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: the Scala code that constitutes the query engine, despite its high-level appearance, is actually a program generator that emits specialized, low-level C code. We show how the combination of high-level and generative programming allows to easily implement a wide spectrum of optimizations that are difficult to achieve with existing low-level query compilers, and how it can *continuously* optimize the query engine.

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 system as well as an existing query compiler, (b) these performance improvements require programming just a few hundred lines of high-level code instead of complicated low-level code that is required by existing query compilers and, finally, that (c) the compilation overhead is low compared to the overall execution time, thus making our approach usable in practice for efficiently compiling query engines.

1. INTRODUCTION

Software specialization is becoming increasingly important for overcoming performance issues in complex software systems [25]. In the context of database management systems, it has been noted that query engines do not, to date, match the performance of handwritten code [33]. Thus, compilation strategies [20, 15, 12] have been proposed in order to optimize away the overheads of traditional database abstractions like the Volcano operator model [4].

Despite the differences between the individual approaches, all compilation frameworks generate an *optimized* query evaluation engine *on-the-fly* for each incoming SQL query. We identify four main problems with existing solutions:

- *Template expansion misses optimization potential.* Virtually all previous query compilers are based on *code template expansion*, a technique that generates code directly, in one step, from the query plan by replacing each operator node by its code template. In its purest form, template expansion makes cross-operator code optimization inside the query compiler impossible.
- *Template expansion is brittle and hard to implement.* Providing low-level code templates – essentially in stringified form – makes it hard or impossible to automatically typecheck the code templates. Moreover, since the templates are to be directly emitted by the code generator, they are very low-level and difficult to implement and get right. The developer of the query compiler has to deal with low-level concerns of code generation, such as register allocation. (This is in a way even true when generating LLVM code.)
- *Limited scope of query compilation.* A compiler that only handles queries cannot optimize and inline their code *with* the remaining code of the database system, missing opportunities to further improve performance.
- *Limited adaptivity.* Systems such as LLVM provide support for runtime optimization and just-in-time compilation, but key runtime optimizations (such as some known from adaptive query processing) are only possible given query plans or high-level code that a system such as LLVM that receives code from a traditional query compiler never gets to see. For instance, LLVM will not be able to reverse-engineer the code it receives to make effective use of selectivity information.

In this paper, we argue that DBMSes and query compilers should, in general, allow for *both* productivity and high performance, *instead* of trading-off the former for the latter. Developers should be able to program DBMSes and their optimizations efficiently at a high-level of abstraction, without experiencing negative performance impact. This has been previously called *abstraction without regret* [21, 10, 11]. We draw inspiration from the recent use of high-level languages for complex system development (e.g. the Singularity Operating System [8]) to argue that it is now time for a radical rethinking of how database systems are designed.

This paper makes the following three contributions:

- We present LegoBase, a new in-memory query execution engine written in the high-level programming language Scala. This is in contrast to the traditional wisdom which calls for the use of low-level languages for DBMS development. To avoid the overheads of a high-level language (e.g. complicated memory management) while maintaining nicely defined abstractions, LegoBase compiles the Scala code to optimized, low-level C code for each SQL query on the fly. By programming databases in a high-level style and still being able to get good performance,

*Part of this work was performed while this author was an intern at Oracle Labs.

Proceedings of the VLDB Endowment, Vol. 7, No. 10
Copyright 2014 VLDB Endowment 2150-8097/14/06.

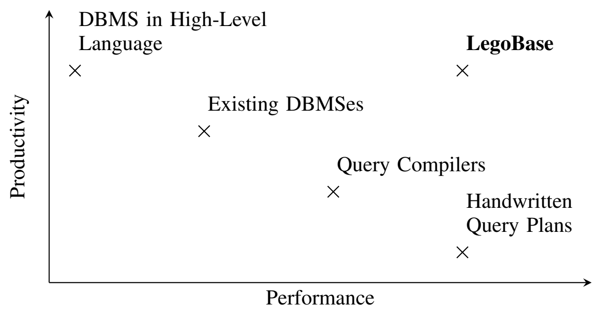


Figure 1: Comparison of performance/productivity tradeoff for all approaches presented in this paper.

```
select *
from R, (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
where R.Z=T.E and R.B=3
```

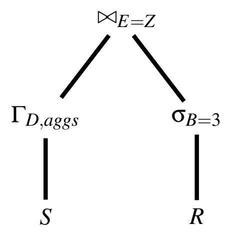


Figure 2: Motivating example showing missed optimizations opportunities by existing query compilers.

the time saved can be spent implementing more database features and optimizations. The LegoBase query engine is the first step towards providing a full DBMS system written in a high-level language.

In addition, high-level programming allows to quickly define system modules that are truly reusable (even in contexts very different from the one these were created for) and easily composable [16], thus putting an end to the monolithic nature of important DBMS components like the storage manager. This property makes the overall maintenance of the system significantly easier. More importantly, it grants great flexibility to developers so that they can easily choose and experiment with a number of choices when building query engines.

- We apply *generative* programming [27] to DBMS development. This approach provides two key benefits over traditional query compilers: (a) programmatic removal of abstraction overhead and (b) applying optimizations on multiple abstraction levels.

  First, the Scala code that constitutes the query engine, despite its high-level appearance, is actually a program generator that emits optimized, low-level C code. In contrast to traditional compilers, which need to perform complicated and sometimes brittle analyses before (maybe) optimizing programs, generative metaprogramming in Scala takes advantage of the type system of the language in order to provide programmers with strong *guarantees* about the shape and nature of the generated code. For example, it ensures that certain abstractions (e.g. generic data-structures and function calls) are definitely optimized away during code generation.

  Second, generative programming allows optimization and (re-)compilation of code at various execution stages. This is a very important property, as it allows us to view databases as living organisms. When the system is first developed, high-level and non-optimal abstractions can be used to simplify the development process. During deployment, as more information is gathered (e.g. runtime statistics, configuration and hardware specifications), we can *continuously* "evolve" the query engine by recompiling the necessary components in order to take advantage of up-to-date information. To our knowledge, LegoBase is the first to support such *continuous runtime optimization* of the whole query engine. This design choice differentiates our system from recent work on compiling only queries [15] or query optimization frameworks such as Starburst [6].

  In our work, we use the Lightweight Modular Staging (LMS) compiler [21] for Scala. In addition to the previous contributions, we leverage the high-level and *extensible* IR of LMS. This design property allows us to extend the scope of compilation and perform *whole-program* optimization, by specializing all data-structures and auxiliary algorithms of a query. We do so by specifying custom, database-specific optimizations. These are implemented as library components, providing a *clean* separation from the base code of LegoBase. Optimizations are (a) easily adjustable to the characteristics of workloads and architectures, (b) easily configurable, so that they can be turned on and off at demand and (c) easily composable, so that higher-level optimizations can be built from lower-level ones. These properties are very hard to provide using any existing template-based compiler. We present examples of optimizations for query plans (inter-operator optimizations), data-structures, and data layout.

- We provide an experimental evaluation with the TPC-H benchmark [28] which shows that our system, along with the aforementioned optimizations, can significantly outperform both a commercial in-memory database, called DBX, and the query compiler of the HyPer system [15]. This improvement requires programming just a few hundred lines of Scala code for the optimizations, thus demonstrating the great expressive power of our optimization framework. An important observation in this context is that developers cannot rely on low-level compilation frameworks, like LLVM, to automatically detect the high-level optimizations that we support in LegoBase. In addition, we show that our query compilation strategy incurs negligible overhead to query execution. These results aim to prove the promise of the *abstraction without regret* vision.

Motivating Example. To better understand the differences of our work with previous approaches, consider the simple 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 B. 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 all existing query compilers:

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

- Second, template-based approaches may result in unnecessary computation. 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 3.1.2.

- 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 pre-compiled 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 paper is organized as follows. Section 2 presents the overall design of LegoBase in more detail, while Section 3 gives examples of compiler optimizations in multiple domains. Section 4 presents our evaluation, where we experimentally show that our approach can lead to significant benefits compared to (i) an existing query compiler and (ii) a commercial database system. Section 5 presents related work in compilation and compares our approach with existing query compilers and engines. Finally, Section 6 concludes and highlights future work.

2. SYSTEM DESIGN

In this section we present the overall design of LegoBase, shown in Figure 3. First, we describe the Lightweight Modular Staging (LMS) compiler that is the core of our architecture. Then, we describe how LMS fits in the overall execution workflow of LegoBase (Subsection 2.2), and how we generate the final optimized C code (Subsection 2.3). While doing so, we give an example of how a physical query operator is implemented in our system.

2.1 Staged Compilation & LMS

LegoBase makes key use of the LMS framework [21], which provides runtime compilation and code generation facilities for the Scala programming language. LMS operates as follows. Given some program written in Scala, LMS first converts the code to a graph-like intermediate representation (IR). In contrast to low-level compilation frameworks like LLVM that offer an IR which operates on the level of registers and basic blocks, LMS provides high-level IR nodes which correspond to constructs and operations in Scala. This makes client code that uses LMS for runtime optimization similar to regular Scala code. In addition, LMS provides a high-level interface to add custom IR nodes, representing operations on programmer-defined types and abstractions. For example, IR nodes in LMS may represent the creation of a hash map, the update of an array element, or operations on primitive values such as the addition of two integers.

Programmers specify the result of a program transformation as a high-level Scala program, as opposed to a low-level, compiler-internal program representation. These transformations manipulate the structure of the IR graph and they add, remove or replace nodes, depending on the optimization goal. For example, our data-structure specialization (Section 3.2) replaces IR nodes representing operations on hash maps with IR nodes representing 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. We use this

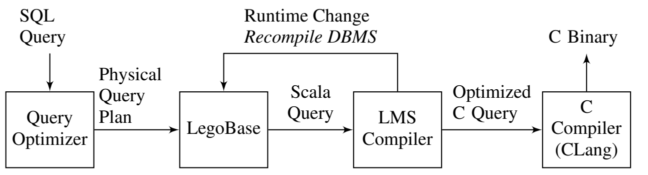


Figure 3: Overall system architecture. The domain-specific optimizations of LegoBase are applied during the LMS compiler phase.

optimization interface to provide database-specific optimizations as a library and to aggressively optimize our query engine.

LMS then performs consecutive transformation passes where it applies *all* possible user-defined optimizations and generates a new IR, which is closer to the optimized final code. This structured approach to optimizing code allows optimizations to be easily combined. As we show in Section 4, with a relatively small number of transformations, we can get significant performance improvement. After each optimization pass, the *whole* set of optimizations is re-examined, since more of them may now be applicable.

In addition to this high-level optimization framework, LMS provides a programming model for expressing what is placed in the IR. The key programming abstraction for this is to introduce a type distinction for program expressions that will be compiled at runtime. Depending on the type of an expression (`T` vs. `Rep[T]`, where `T` represents a type like Integer), we speak of *present-stage* vs. *future-stage* expressions and values. In particular, present-stage computation is executed right away, while future stage expressions (`Rep[T]`) are placed in the IR graph, so that they can be optimized in subsequent optimization passes, as described above. Operations that operate on future-stage objects, e.g. an addition, are also converted to IR nodes. This programming model can be leveraged in a number of design patterns: present-stage functions that operate on future-stage values are automatically inlined, loops with a present-stage trip count are unrolled, and future-stage data-structures can be specialized based on present-stage information. A key example in databases are records: when the schema is fixed, we can model schema information and records as present-stage objects that reference future-stage field values. Such records can be manipulated using object oriented or functional programming without paying a price at runtime: the type system guarantees that the future-stage code will contain *only* those field values that are actually used, and that these fields will be represented as local variables.

The LMS compiler by default generates Scala code as output from the optimized IR. In this work, we extend LMS so that it generates C code as well. To reach the abstraction level of C code, transformations also include multiple *lowering* steps that map Scala constructs to (a set) of C constructs. 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 automatic way of lowering does not require any modifications to the database code or effort from the database developer. After these lowering steps, we can also apply low-level, architecture-dependent optimizations. In the end, the final iteration over the IR nodes emits the C code.

LMS already provides many generic compiler optimizations like function inlining, common subexpression and dead code elimination, constant propagation, loop fusion, deforestation, 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 in more detail in Section 3.

```
1  case class AggOp[B](child: Operator, grp: Record=>B,
2    aggFuncs: (Record,Double)=>Double*) extends Operator {
3   val hm = HashMap[B,Array[Double]]()
4   def processAggs(aggs: Array[Double], rec: Record) {
5     aggFuncs.indices foreach { i =>
6       aggs(i) = aggFuncs(i)(rec, aggs(i))
7     }
8   }
9   def open() {
10    child foreach { t =>
11      val key = grp(t)
12      val aggs = hm.getOrElseUpdate(key,
13        new Array[Double](aggFuncs.size))
14      processAggs(aggs,t)
15    }
16  }
17  def next() : Record = {
18    if (hm.size != 0) {
19      val elem = hm.head
20      hm.remove(elem._1)
21      return elem
22    } else return null
23  }
24 }
```

Figure 4: Example of an operator in LegoBase.

2.2 General Execution Workflow

LMS interacts with LegoBase as 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 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 called DBX.

Then, we pass the generated physical plan to LegoBase. Our system in turn parses this plan and instantiates the corresponding Scala implementation of the mentioned operators. For example, for aggregations, LegoBase instantiates the Scala operator shown in Figure 4. Note that this operator implementation is exactly what one would write for a simple query engine that does not involve query compilation at all. However, without further optimizations, this engine cannot match the performance of existing databases: it consists of generic data-structures, mimics most of the overhead of Volcano-style engines and matches the code that is generated by a naive template-based query compiler.

However, in our system, LMS compiles all code on-the-fly (including 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 C constructs (Section 3.2). In addition, it inlines the operators and, in the code of Figure 4, it automatically unrolls the loop of lines 5-7, since the number of aggregations can be statically determined based on how many aggregations the SQL query has. Such fine-grained optimizations have a significant effect on performance, as they improve branch prediction. Finally, our system generates the optimized C code, which is compiled using any existing C compiler (e.g. we use CLang of LLVM [13] for our evaluation). We then return the query results to the user.

We also use LMS to recompile the components of our query engine that are affected by a runtime change. We motivate this decision based on the observation that, for database systems, many configuration variables are set *only* at start time but are unnecessarily checked multiple times at runtime. In LegoBase, the compiled final code does not include any **`if`** condition checking configuration parameters. Instead, at startup-time we generate only the proper branch according to the value of the parameter. Then, we re-compile components *only if* some parameter changes at runtime. For example, if the user disables logging, then we recompile to remove all logging statements. This *continuous* cycle of optimization and execution is only made possible by on-the-fly compilation and, to our knowledge, is not provided by any existing query compiler.

2.3 Generating Efficient C Code

As we already mentioned, the final step of compiling an incoming query using LegoBase is the C code generation. In general, the translation from Scala to C is straightforward. Most Scala abstractions (e.g. objects, classes, inheritance) are optimized away at program generation time, and for the remaining constructs (e.g. loops, variables, arrays) there exists a one-to-one correspondence between Scala and C. There are two exceptions to this rule, described next.

First, calls to library functions in Scala should be translated to corresponding calls in C. For example, all operations for data-structures like hash maps should be matched in C appropriately. In the final output code produced by LegoBase, with all optimizations enabled, we do not have many such calls as all our data-structures are lowered to primitive arrays (Section 3.2). Thus, this 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 support traditional collections as a library. We have found GLib to be efficient enough for this purpose. Thus, we match the Scala collections to the corresponding ones in GLib.

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 compilation from Scala to C, we must take 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 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 allocations and deallocations explicit in the Scala code. We also free the allocated memory after each query execution.

3. STAGING OPTIMIZATIONS

In this section we present examples of compiler optimizations in four domains: (a) inter-operator optimizations for query plans, (b) transparent data-structure modifications, (c) changing the data layout and, finally, (d) traditional compiler optimizations like dead code elimination. The purpose of this section is to demonstrate the expressive power 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. In addition, we explain the differences of coding these optimizations in our architecture compared to previous query compilers. The structure of this section closely follows the domains described above.

3.1 Inter-Operator Optimizations

3.1.1 From a Volcano (Pull) to a Push-based Engine

A recent proposal in the area of query compilers, which was presented in [15], states that we should change the flow of data processing in query engines. More specifically, it argues that operators should not *pull* data from other operators whenever needed (Volcano-style processing), but instead operators should *push* data to consumer operators. Data should then be continuously pushed until

```
case class HashJoin[B](leftChild: Operator,
  rightChild: Operator, hash: Record=>B,
  cond: (Record,Record)=>Boolean) extends Operator {
 val hm = HashMap[B,ArrayBuffer[Record]]()
 var it: Iterator[Record] = null
 def next() : Record = {
  var t: Record = null
  if (it == null || !it.hasNext) {
   t = rightChild.findFirst { e =>
    hm.get(hash(e)) match {
     case Some(hl) => it = hl.iterator; true
     case None => it = null; false
    }
   }
  }
  if (it == null || !it.hasNext) return null
  else return it.collectFirst {
   case e if cond(e,t) => conc(e, t)
  } get
 }
}
```

(a) The starting Volcano-style implementation.

```
case class HashJoin[B](leftChild: Operator,
  rightChild: Operator, hash: Record=>B,
  cond: (Record,Record)=>Boolean) extends Operator {
 val hm = HashMap[B,ArrayBuffer[Record]]()
 var it: Iterator[Record] = null
 def next(t: Record) {
  var res: Record = null
  while (res = {
   if (it == null || !it.hasNext) {
    hm.get(hash(t)) match {
     case Some(hl) => it = hl.iterator
     case None => it = null
    }
   }
   if (it == null || !it.hasNext) null
   else it.collectFirst {
    case e if cond(e,t) => conc(e, t)
   } get
  } != null) parent.next(res)
 }
}
```

(b) After the first two steps of the algorithm.

```
case class HashJoin[B](leftChild: Operator,
  rightChild: Operator, hash: Record=>B,
  cond: (Record,Record)=>Boolean) extends Operator {
 val hm = HashMap[B,ArrayBuffer[Record]]()
 var it: Iterator[Record] = null
 def next(t: Record) {
  if (it == null || !it.hasNext) {
   hm.get(hash(t)) match {
    case Some(hl) => it = hl.iterator
    case None => it = null
   }
  }
  while (it!=null && it.hasNext) it.collectFirst {
   case e if cond(e,t) => parent.next(conc(e,t))
  }
 }
}
```

(c) After the third step of the algorithm.

```
case class HashJoin[B](leftChild: Operator,
  rightChild: Operator, hash: Record=>B,
  cond: (Record,Record)=>Boolean) extends Operator {
 val hm = HashMap[B,ArrayBuffer[Record]]()
 def next(t: Record) {
  hm.get(hash(t)) match {
   case Some(hl) => hl.foreach { e =>
    if (cond(e,t)) parent.next(conc(e,t))
   }
   case None => {}
  }
 }
}
```

(d) The final result after additional optimizations.

Figure 5: Transforming a HashJoin from a Volcano engine to a Push Engine. The lines highlighted in red and blue are removed and added, respectively. All branches and intermediate iterators are automatically eliminated. The *open* function (not shown) is handled accordingly.

we reach a materialization point. This organization significantly improves cache locality and branch prediction [15].

However, this dataflow optimization comes at the cost of requiring an API change (from an iterator to a consumer/producer model). This in turn necessitates rewriting all operators: with traditional approaches, this is a challenging and error-prone task considering that the logic of each individual operator is likely spread over multiple code fragments of complicated low-level software [15].

Given the two types of engines, there exists a methodological way to obtain one from the other. Thus, LegoBase implements both the Volcano model and a push engine, which we mechanically derived from Volcano. We present the high-level ideas of this conversion next, using the HashJoin operator as an example (Figure 5).

A physical query plan consists of a set of operators in a tree structure. For each operator, we can extract its children as well as its (single) parent. Operators call the *next* function of other children operators in the Volcano model to make progress in processing a tuple. An operator can be the *caller*, the *callee* or even both depending on its position in the tree (e.g. an operator with no children is only the callee, but an operator in an intermediate position is both). Given a set of operators, we must take special care to (a) reverse the dataflow (turning callees to callers and vice versa) as well as (b) handle *stateful* operators in a proper way. The optimization handles these cases in the following three steps:

Turning callees to callers: When calling a *next* function in the Volcano model, a single tuple is returned by the callee[1]. In contrast, in a push model, operators call their parents whenever they have a tuple ready. The necessary transformation is straightforward: instead of letting callees return a single tuple, we remove this return statement. Then, we put the whole operator logic inside a while loop which continues until the value that would be returned in the *original* callee operator is null (operator has completed execution). For each tuple encountered in this loop, we call the *next* function of the original parent. For scan operators, who are only callees, this step is enough to port these operators to the push-style engine.

Turning callers to callees: The converse of the above modification should be performed: the original callers should be converted to callees. To do this, we remove the call to the *next* function of the child in the original caller, since in the push engine the callee calls the next function of the parent. However, we still need a tuple to process. Thus, this step changes all *next* functions to take a record as argument, which corresponds to the value that would be returned from a callee in the Volcano engine. Observe that the call to *next* may be explicit or implicit through functional abstractions like the

[1]This assumes no block-style processing, where multiple tuples are first materialized and then returned as a unit. In general, LegoBase avoids materialization whenever possible.

findFirst in line 9 of Figure 5(a). In addition, calls to the *next* function may happen in the *open* function of the Volcano model for purposes of *state-initialization*. We handle the *open* function similarly. This step ports the Sort, Map, Aggregate, Select, Window, View and Print operators of LegoBase to the push-engine[2].

Managing state: Finally, special care should be taken for *stateful* operators. The traditional example of such operators is the join variants (semi-join, hash-join, anti-join etc). For these operators, the tuples from the left child are organized in hash lists, matched on the join condition with tuples from the right child. Then, to avoid materialization, the join operator must keep state about how many elements have already been output from this list whenever there is a match. A nice abstraction for this is the *iterator* interface[3], where for each *next* call in the Volcano model the iterator is advanced by one (and one output tuple is produced). In this optimization we change this behaviour so that after the iterator is initialized, we exhaust it by calling the *next* function of the parent for each tuple in it.

It is important to note that the above methodology, which seems straightforward, reveals an important advantage of our staging compiler infrastructure: that by programming operators at a high-level, it then becomes straightforward to express optimizations for those operators. In this example, the optimization's code follows closely the human-readable description given above. Corner cases can then be handled on top of this baseline implementation, sacrificing neither the readability of the original operators nor the baseline optimization itself. In addition, after this optimization, the staging compiler can further optimize the generated code, as shown in Figures 5(c) and 5(d). There, the compiler detects that both the iterator abstraction and some while loops can be completely removed, and automatically removes them, thus improving branch prediction. This is an important advantage of staging compilers compared to the existing template-based and static query compilers.

3.1.2 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 the 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, 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. The code of the optimization is shown in Figure 6.

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 the chain is (as shown in line 2 of Figure 6) a hash-join operator connected to an aggregate operator. When this pattern is detected, a new HashJoin operator object is created, that is *not* connected to the aggregate operator,

```
1  def optimize(op: Operator): Operator = op match {
2    case hj@HashJoin(aggOp:AggOp,_,h,eq) =>
3      new HashJoin(aggOp.child,hj.rightChild,h,eq) {
4        override def open() {
5          // leftChild is now the child of aggOp
6          leftChild foreach { t =>
7            val key = hj.leftHash(aggOp.grp(t))
8            // Get aggregations from hash map of HJ
9            val aggs = hm.getOrElseUpdate(key,
10                       new Array[Double](aggOp.aggFuncs.size))
11           aggOp.processAggs(aggs,t)
12         }
13       }
14     }
15   case x: Operator =>
16     x.leftChild = optimize(x.leftChild)
17     x.rightChild = optimize(x.rightChild)
18   case null => null
19 }
```

Figure 6: Removing redundant materializations by high-level programming (here between a group by and a join).

but instead to the child of the latter (line 3 of Figure 6). 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 9), and we just call the corresponding function of the Aggregate operator (line 11), respectively. Observe that in this optimization there is almost no code duplication, showing the great merit of abstraction without regret. In addition, all low-level compiler optimizations can still be applied after the application of the optimization presented here.

Finally, 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 property raises the productivity provided by our compiler, and is another example where optimizations are developed in a way that is completely intuitive to programmers. This design also allows them to use all existing software development tools for optimizing the query engine.

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 database management system reveals that there are many distinct implementations of 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 by any existing query compilation engine. Since our LMS compiler can be used to optimize the *whole* Scala code, and not only the operator interfaces, it allows for various degrees of specialization in data-structures, as has been previous shown in [22]. In this paper, we demonstrate such possibilities by showing how hash maps, which are the most commonly used data-structures along with Trees in DBMSes, can be heavily specialized for significant performance improvements by using schema and query knowledge. Close examination of the generic hash maps in the baseline implementation of our operators (e.g. in the Aggregation of Figure 4) reveals the following three main abstraction overheads.

First, for every *insert* operation, a hash map must allocate a triplet holding the key, the corresponding value as well as a pointer to the next element in the hash bucket. This introduces a significant

[2]All operators initialize their state (if any) from one child in the open function, and call their other child (if any) in the next function. The only exception is the nested loop joins operator which calls both children in the next function. We handle this by introducing phases where each phase handles tuples only from one child.

[3]Observe that the *iterator* itself is an abstraction which introduces overheads during execution. Our compiler maps this high-level construct to efficient native C loops.

```
1  trait HashMapOpsGen extends HashMapOpsExp {
2   override def lowerNode[A](sym: Sym[A], rhs: Def[A]) =
3    rhs match {
4     case HashMapNew[K,V](size,h,eq)=> sym.atPhase(LOWERING){
5       // Create new IR node for array storing only values
6       val sym = new Array[V](size)
7       // Keep hash and equal functions in new IR node
8       sym.attributes += "hash" -> h
9       sym.attributes += "equals" -> eq
10      sym // The IR node now represents an array
11     }
12    case HashMapGetOrElseUpd(m,k,v)=> sym.atPhase(LOWERING){
13      // var m now represents an array instead of a hash map
14      // Extract functions
15      val hashF = m.attributes("hash")
16      val equalF = m.attributes("equals")
17      // Get bucket
18      var h = hashF(v) // Inlines hash function
19      var elem = m(h)
20      // Search for element & inline equals function
21      while (elem != null && !equalF(elem, k))
22        elem = elem.next
23      // Not found: create new elem / update pointers
24      if (elem == null) {
25        elem = v()
26        elem.next = m(h)
27        m(h) = elem
28      }
29      elem // The IR node now represents an array element
30     }
31   }
32   // Fill remaining operations accordingly
33 }
34
35 trait StructOpsExpOpt extends StructOpsExp {
36   override def struct[T](elems: Seq[(String,Any)]) = {
37     // Transparently append next field
38     val fields = ("next", manifest[T]) :: elems
39     val name = structName(fields)
40     super.struct(name, fields)
41   }
42 }
```

Figure 7: Specializing HashMaps by converting them to native arrays. The operations are mapped to a set of primitive C constructs.

number of expensive memory allocations on the critical path. Second, hashing and comparison functions are called for every *lookup*. 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 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. This property shows that our approach is practical, in contrast to the complicated, low-level approaches followed by query compilers so far. The optimization, which is shown in Figure 7, takes place during the *lowering* phase of the compiler (Section 2.1), where high-level Scala IR nodes are mapped to low-level C constructs. It makes use of the following three observations:

- For our workloads, the information stored on the key is *usually* a subset of the attributes of the value. Thus, the generic hash maps store redundant data. To avoid this, we convert the hash map to an array that stores only the values, and not the associated key (lines 4-11). Then, since we know that the inserted elements are 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*, lines 35-42). Thus, we no longer need the key-value-next container and we manage to reduce the amount of memory allocations significantly.

- Second, the hash and equal functions are themselves IR nodes in LMS. Thus, we can automatically inline the body of those functions wherever they are called (lines 18 and 21 of Figure 7), as described in Section 2.1. This significantly reduces the number of function calls (to almost zero), considerably improving branch prediction and cache locality.

- 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 4). Currently, we obtain this size by performing worst-case analysis on a given query, which means that we possibly allocate much more space that what is actually needed. However, we believe that database statistics can make this estimation very accurate.

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.

3.3 Changing Data Layout

A long-running debate in database literature is the one between row and column stores [1, 24, 7]. 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 the row to the column layout can be expressed as an optimization[4].

The optimization of Figure 8 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 takes place during the construction of IR nodes concerning arrays, and overrides the corresponding methods, thus providing the new behaviour. For example, when constructing a new array, array_new is called. 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 5,16,27). 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 staged representation of `arr` will be a record of arrays, and that `v` has the same attributes as `arr`. So for each of those attributes we extract the corresponding array from `arr` (line 18) and field from `v` (line 20); then we can perform the update operation on the extracted array (line 20).

This optimization also reveals another benefit of using a staging 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 28-29) and then building a new record to hold the result (line 31). This intermediate record can be *automatically* removed using dead code elimination (DCE), as shown in

[4]We must note that just 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.

```
trait ArrayOpsExpOpt extends ArrayOpsExp {
 // Override the IR node constructors
 override def array_new[T:Manifest](n:Int) =
  manifest[T] match {
   case Record(attrs) =>
    // Create a new array for each attribute
    val arrays = for (tp<-attrs) yield array_new(n)(tp)
    // Pack everything in a new record
    record(attrs, arrays)
   case _ => super.array_new(n)
  }

 override def array_update[T:Manifest](ar:Array[T],
                                     n:Int, v:T) =
  manifest[T] match {
   case Record(attrs) =>
    // Get columns and update each one
    val arrays = for (l <- attrs) yield field(ar, l)
    for ((a, l) <- arrays zip attrs)
     a(n) = field(v, l)
   case _ => super.array_update(ar, n, v)
  }

 override def array_apply[T:Manifest](ar:Array[T],
                                    n:Int) =
  manifest[T] match {
   case Record(attrs) =>
    val arrays = for (l <- attrs) yield field(ar, l)
    val elems = for (a <- arrays) yield a(n)
    // Perform record reconstruction
    record(attrs, elems)
   case _ => super.array_apply(ar, n)
  }

 // Fill remaining operations accordingly
}
```

Figure 8: Changing the data layout (from row to column) expressed as an optimization. ArrayOpsExp is the compiler trait for handling Arrays that we overwrite. Scala manifests carry type information.

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

Figure 9: Dead code elimination (DCE) can remove intermediate materializations, e.g. row reconstructions when using a column layout. Here *a* is an array of records and *i* is an integer. The records have two attributes *L1* and *L2*.

Figure 9. Similarly, if LMS can statically determine that some attribute is never used (e.g. by having all queries given in advance), then the row layout still has to skip this attribute during query processing. Instead, after applying this transformation, this attribute will just be an unused field in a record, which the staging compiler will be able to optimize away (e.g. attribute L2 in Figure 9).

Such optimization opportunities, which are provided *for free* by LMS, have to be manually encoded with existing query compilers. We argue that this is a benefit of actually *using* a compiler, instead of *mimicking* what a compiler *would do* inside the query engine.

3.4 Other Compiler Optimizations

There are several other optimizations that can be expressed with our compiler framework in order to further boost the performance of LegoBase. These include loop-fusion, automatic index introduction, automatic parallelization and vectorization. We leave these optimizations as future work. However, preliminary results show that these optimizations can be easily expressed in LegoBase and can significantly improve performance as expected. In general, we believe that exploration of even query-specific optimizations is certainly feasible, given the easy extensibility of our framework.

4. EVALUATION

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 6.5. For compiling the generated programs throughout our evaluation we use version 2.10.3 of the Scala compiler and version 2.9 of the CLang front-end for LLVM [13], with the default optimization flags for both compilers. For the Scala programs, we configure the Java Virtual Machine to run with 192GB of heap space. Finally, for C data-structures we use the GLib library (version 2.38.2).

For our evaluation we use the TPC-H benchmark [28]. TPC-H is a data-warehousing and decision support benchmark 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 execute each query five times and report the average performance of these runs. As a reference point for all 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. For all experiments, we have disabled huge pages in the kernel, since this provided better results for all tested systems and optimizations. As described in Section 2, LegoBase uses query plans from the DBX database.

Our evaluation is divided into three parts. First, we analyze the performance of LegoBase. More specifically, we show that, by using our compiler framework, we obtain a query engine that significantly outperforms both DBX and the HyPer query compiler. We also give insights about the performance improvement each of our optimizations provides. Second, we analyze the amount of effort required when programming 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. Finally, we evaluate the compilation overheads of our approach to show that it is practical for efficiently compiling query engines.

4.1 Optimizing Query Plans

First, we show that low-level compilation frameworks, such as LLVM, are not adequate for efficiently optimizing database systems. To do so, we generate a traditional Volcano-style engine, which we then compile to a final C binary using LLVM. As shown in Figure 10, the achieved performance is very poor: the LegoBase query engine system is significantly faster for all TPC-H queries. This is because frameworks like LLVM cannot automatically detect the data-structure, data flow or operator optimizations that we support in LegoBase: the scope of optimization is too coarse-grained to be detected by a low-level compiler.

In addition, as shown in the same figure, compiling with LLVM does not *always* yield better results compared to using a traditional compiler like GCC[5]. We see that LLVM outperforms GCC for *only* 11 out of 22 queries (by 14% on average) while, for the remaining ones, the binary generated by GCC is faster by 10% in average. In general, the performance difference between the two compilers can be significant (e.g. for Q15, there is a 26% difference). We also experimented with manually specifying optimizations flags to the

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

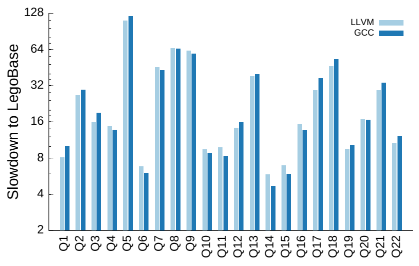


Figure 10: Performance of a Volcano-style engine compiled with LLVM and GCC.

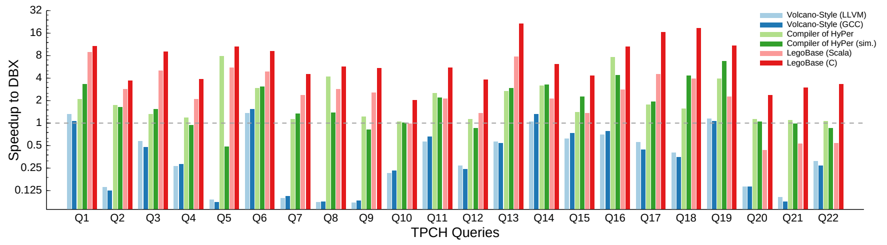


Figure 11: Performance comparison of LegoBase (C and Scala programs) with the code generated by the query compiler of [15].

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 database developers to invest their effort in developing high-level optimizations, like those presented so far in this paper.

Second, we show that the limited optimization scope of existing query compilers makes them miss significant optimization opportunities. To do so, we use the compiler of the HyPer database [15] which employs LLVM, a push engine and operator inlining[6]. We also simulate this system in LegoBase by enabling the corresponding optimizations in our architecture[7]. The results are presented in Figure 11. We see that, for both the simulated and actual HyPer compilers, performance is significantly improved by 2.15× and 2.44× on average, respectively. In addition, for 10 out of 22 TPC-H queries, our simulation actually generates code that performs better than that of HyPer. This is because we inline not only the operators' interfaces but *also* all data-structures and utilities leading to fewer function calls and better cache locality[8].

More importantly, this figure shows that by using the data layout and data structures optimizations of LegoBase (which are not performed by the query compiler of HyPer), we can get an additional 5.3× speedup, for a total average 7.7× performance improvement with all optimizations enabled. This is a result of the improved cache locality and branch prediction, as shown in Figure 13. More specifically, there is an improvement of 30% and 1.54× 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 2.98×, 1.54×, and 5% more, respectively compared to LegoBase. The data-structure and column layout optimizations cannot be provided by existing query compilers as they target pre-compiled DBMS components which exist outside their optimization scope. This shows that, by extending the optimization scope, LegoBase can outperform existing compilation techniques for all TPC-H queries.

Finally, we prove that the *abstraction without regret* vision necessitates our source-to-source compilation to C. To do so, we present performance results for the best Scala program; that is the program generated by applying all optimizations to the Scala output. We observe that the performance of Scala cannot compete with that of the optimized C code, and is on average 2.5× slower. Profiling information gathered with the *perf* tool of Linux reveals the following three reasons for the worse performance of Scala: (a) There are 30% to 1.4× more branch mispredictions, (b) The percentage of LLC misses is 10% to 1.8× higher, and more importantly, (c) Scala executes up to 5.5× more CPU instructions[9]. 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: for 18 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 45× when applying our data-structure specializations.

4.1.1 Impact of Compiler Optimizations

From the results presented so far, we observe that our optimizations do not equally benefit the performance of all queries, however they never result in negative performance impact. Here, we provide additional information about the performance improvement expected when applying one of our optimizations. These results are presented in Figure 12.

In general, the impact of an optimization depends on the characteristics of a query. For the data-structure specialization (Figure 12a), the improvement is proportional to the amount of data-structure operations performed. We observe that the hash map abstraction performs respectably for few operations. However, as we increase the amount of data that are inserted into these maps, their performance significantly drops and, thus, our specialization gives significant performance benefits. For the column layout optimization (Figure 12b), the improvement is proportional to the percentage of attributes in the input relations that are actually used. TPC-H queries reference 24% - 68% and, for this range, the optimization gives a 2.5× to 5% improvement, which degrades as more attributes are referenced. This is expected as the benefits of the column layout are evident when this layout can "skip" 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. This figure also shows that the performance improvement of both optimizations is not directly dependent on the number of operators, as queries with the same number of operators can exhibit completely different behaviour regarding data-structure and attributes references.

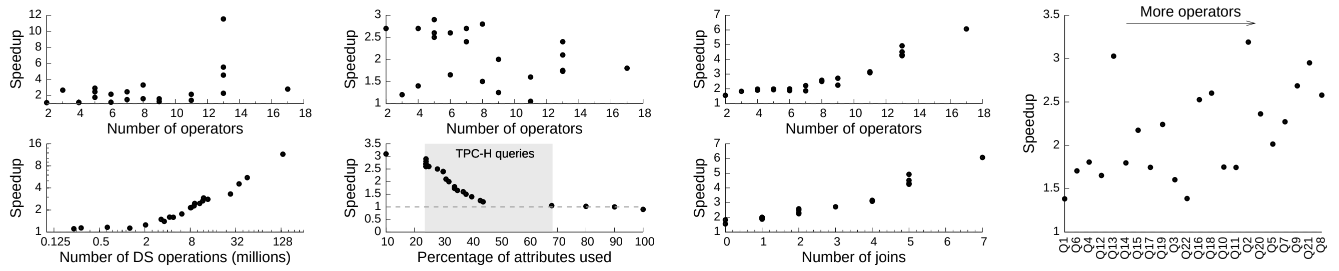
For the inlining optimization (Figure 12c) we observe that, when all operators are considered, inlining does not improve performance as we move from three to seven operators. This is because the improvement obtained from inlining depends on which operators are

[6]We also experimented with *another* in-memory DBMS that compiles SQL queries to native C++ code on-the-fly. However, we were unable to configure the system so that it performs well compared to the other systems. Thus, we omit its results from this section.

[7]In its full generality, the transformation between a Volcano and a push engine is still under development. For the results presented here, we have implemented the push version directly since, in our case, the code of the push engine turns out to be significantly simpler and easier to understand than the Volcano code.

[8]We note that the simulated and actual HyPer systems may use different physical query plans and data-structures implementation. These are the main reasons for the different performance observed in Figure 11 between the two systems in some queries.

[9]These results were confirmed with Intel's VTune profiler.



(a) Data Structure Opt. (b) Change Data Layout (c) Operator Inlining (d) Push Engine Opt.

Figure 12: Impact of different optimizations on query execution time. The baseline is a Volcano-style engine.

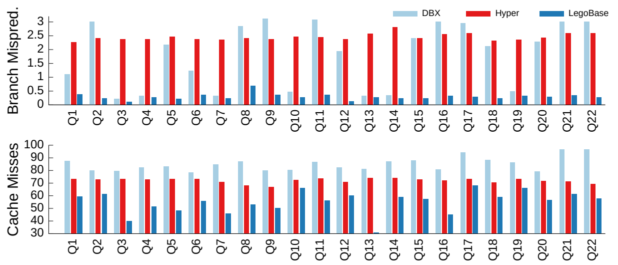


Figure 13: Percentage of cache misses and branch mispredictions for DBX, HyPer and LegoBase for all 22 TPC-H queries.

	Coding Effort	Scala LOC	Average Speedup
Operator Inlining	–	0	2.07×
Push Engine Opt.	1 Week	∼400 [6]	2.26×
Data Structure Opt.	4 Days	259	2.16×
Change Data Layout	3 Days	102	1.81×
Other Misc. Opt.	3 Days	124	–[10]
LegoBase Operators	1 Month	428	–
LMS Modifications	2 Months	3953	–
Various Utilities	1 Week	538	–
Total	∼4 Months	5831	7.7×

Table 1: Programming effort required for each LegoBase component along with the average speedup obtained from using it.

being inlined. We observe that if we consider inlining only join operators then the performance improves almost linearly as the number of join operators in a query plan increases. This is an important observation, as for very large queries, our system may have to choose which operators to inline (e.g. to avoid the code not fitting in the instruction cache). If that is the case, this experiment shows that the compiler framework should merit inlining joins instead of simpler operators (e.g. scans or aggregations).

Finally, the performance improvement gained by the pull to push optimization (Figure 12d) 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 to Q21 in the figure have the same number of operators, but the performance improvement gained varies significantly. At the same time Q13 has half the number of operators, but this optimization helps more: the push engine significantly simplifies the complex execution paths of the Left Outer Join operator used by this query. A similar observation about the complexity of execution paths holds for Q2 as well.

4.2 Productivity Evaluation

An important point of this paper 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 1.

We observe two things. First, by programming at a high-level we can provide a fully functional system within a small amount of time and lines of code required. For LegoBase, the majority of this effort was invested in extending the LMS compiler so that it generates C code (LMS by default outputs Scala). As a result of the reduced code size, we spent less time on debugging the system, thus focusing on developing new useful optimizations. Development of LegoBase required, including debugging time, four months for only one programmer. Second, each optimization requires only a few hundred lines of high-level code to provide significant performance improvements. More specifically, for ∼900 LOC LegoBase is improved by 7.7×, as we described in the previous section. Source-to-source compilation is critical to achieving this behaviour, as the combined size of the operators and optimizations of LegoBase is 40 times less than the code size for all 22 TPC-H queries written in C. Finally, in contrast to low-level query compilers which must themselves provide operator inlining, LMS provides this optimization for free. We believe these properties prove the productivity merit of the *abstraction without regret* vision.

4.3 Compilation Overheads

Finally, we analyze the compilation time for the C programs of all 22 TPC-H queries. Our results are presented in Figure 14, where the y-axis corresponds to the time to (a) optimize an incoming query in our system and generate the C code, and, (b) the time CLang requires before producing the final C executable.

We see that, in general, all TPC-H queries require less than 2.5 seconds to compile. 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 can be used in practice for quickly compiling query engines. In addition, the optimization time is, as expected, proportional to the number of joins in its physical query plan. This is because our compiler must optimize more data-structures and operators as the number of joins increases[11].

Finally, we note that if we generate Scala code instead of C, then compiling the final optimized Scala programs requires 7.2× more time on average. To some extent this is expected as calling the Scala

[10]The improvement of these optimizations is counted among the other optimizations.

[11]One exception to this rule is Q11. This query uses the Window Operator which is expensive to optimize in our implementation.

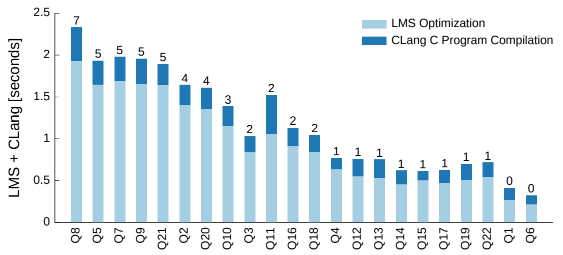


Figure 14: Compilation time for all C programs of TPC-H. Queries are sorted according to the number of join operators in them.

compiler is a heavyweight process: for every query compiled there is significant startup overhead for loading the necessary Scala and Java libraries. In addition, Scala has to perform additional transformations in order to convert a Scala program to Java bytecode. By just optimizing a Scala program in the form of an AST, our two-level architecture allows us to avoid these overheads, providing a much more lightweight compilation process.

5. RELATED WORK

We outline related work in three areas: (a) Previous query compilers, (b) Frameworks for applying intra-operator optimizations and, finally, (c) Orthogonal techniques to speed-up query processing. We briefly discuss these areas below.

Previous Compilation Frameworks. Historically, System R [2] 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 [5] system 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[12], 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.

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) [20]. Krikellas et al. take a step further and completely eliminate the Volcano iterator model in the generated code [12]. They do so by translating the algebraic representation to C++ code using templates in the HIQUE system. In addition, Zane et al. have shown how compilation can also be used to additionally improve operator internals [29].

The HyPer database system also uses query compilation, as described in [15]. 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 *pre-compiled* 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. Furthermore, their static nature makes them miss significant optimization opportunities that can only be detected by taking into account runtime information. 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. In addition, our *staging* compiler is used to *continuously* optimize our system at runtime. Finally, in contrast to previous work, we separate the optimization and code generation phases. Even though [15] 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.

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 [31, 30, 32]. 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 pre-compiled templates in an extensible way. All these optimizations are performed by default by the LMS 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 [30]. 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 the aforementioned two limitations.

Techniques to speed up query processing. Finally, there are many works that aim to speed-up query processing in general, by focusing mostly on improving the way data are processed, rather than individual operators. Examples of such work include block-wise processing [18], vectorized execution [23], compression techniques to provide constant-time query processing [19] or combination of the above along with a column-oriented data layout [14]. We believe all these approaches are orthogonal to this work, since our framework aims to provide 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).

6. CONCLUSIONS

LegoBase is a new analytical database system currently under development at EPFL. In this paper, we presented the current prototype of the query execution subsystem of LegoBase. Our system allows programmers to develop high-level abstractions without having to pay an abstraction penalty. To achieve this vision of *abstraction without regret*, LegoBase performs source-to-source compilation of the high-level Scala code to very efficient low-level C code. In addition, it uses state-of-the-art compiler technology in the form of an extensible *staging* compiler implemented as a library in which optimizations can be expressed naturally at a high level. Our approach admits a productivity/efficiency combination

[12]Examples of systems in the area since mid-2000s include SAP HANA [3], VoltDB [9, 26] and Oracle's TimesTen [17].

that is not feasible with existing low-level query compilers: Programmers need to develop just a few hundred lines of *high-level* code to implement techniques and optimizations that result in significant performance improvements. Our experiments show that LegoBase significantly outperforms both a commercial in-memory database and an existing query compiler.

Acknowledgments

We thank Andres Nötzli and Amir Shaikhha for insightful discussions that helped to improve this paper. Lewis Brown contributed to the column layout optimization. This work was supported by ERC grant 279804.

7. REFERENCES

[1] D. J. Abadi, S. Madden, and N. Hachem. Column stores vs. Row stores: How Different Are They Really? In *ACM SIGMOD*, pages 967–980, 2008.

[2] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of System R. *Comm. ACM*, 24(10):632–646, 1981.

[3] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2012.

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

[5] R. Greer. Daytona and the fourth-generation language Cymbal. In *ACM SIGMOD*, pages 525–526, 1999.

[6] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *ACM SIGMOD*, pages 377–388, 1989.

[7] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.

[8] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[9] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[10] C. Koch. Abstraction without regret in data management systems. In *CIDR*, 2013.

[11] C. Koch. Abstraction without regret in database systems building: a manifesto. *IEEE Data Eng. Bull.*, 37(1):70–79, 2014.

[12] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[13] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. `http://llvm.org/`.

[14] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

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

[16] M. Odersky and M. Zenger. Scalable Component Abstractions. In *OOPSLA*, pages 41–57, 2005.

[17] Oracle Corporation. TimesTen Database Architecture. `http://download.oracle.com/otn_hosted_doc/timesten/603/TimesTen-Documentation/arch.pdf`.

[18] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, pages 567–574, 2001.

[19] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, pages 60–69, 2008.

[20] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, pages 23–, 2006.

[21] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming and Component Engineering*, pages 127–136, 2010. `http://scala-lms.github.io/`.

[22] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*, pages 497–510, 2013.

[23] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, pages 33–40, 2011.

[24] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[25] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, pages 2–11, 2005.

[26] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[27] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[28] Transaction Processing Performance Council. TPC-H, a decision support benchmark. `http://www.tpc.org/tpch`.

[29] B. M. Zane, J. P. Ballard, F. D. Hinshaw, D. A. Kirkpatrick, and L. Premanand Yerabothu. Optimized SQL code generation, 2008. US Patent 7430549 B2.

[30] R. Zhang, S. Debray, and R. T. Snodgrass. Micro-specialization: dynamic code specialization of database management systems. In *Code Generation and Optimization*, pages 63–73, 2012.

[31] R. Zhang, R. Snodgrass, and S. Debray. Application of Micro-specialization to Query Evaluation Operators. In *ICDE Workshops*, pages 315–321, 2012.

[32] R. Zhang, R. Snodgrass, and S. Debray. Micro-Specialization in DBMSes. In *ICDE*, pages 690–701, 2012.

[33] M. Zukowski, P. A. Boncz, N. Nes, and S. HÃl'man. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, (2):17–22, 2005.