How to Architect a Query Compiler

Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown,
Mohammad Dashti, and Christoph Koch
{firstname}.{lastname}@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

This paper studies architecting query compilers. The state of the art in query compiler construction is lagging behind that in the compilers field. We attempt to remedy this by exploring the key causes of technical challenges in need of well founded solutions, and by gathering the most relevant ideas and approaches from the PL and compilers communities for easy digestion by database researchers. All query compilers known to us are more or less monolithic template expanders that do the bulk of the compilation task in one large leap. Such systems are hard to build and maintain. We propose to use a stack of multiple DSLs on different levels of abstraction with lowering in multiple steps to make query compilers easier to build and extend, ultimately allowing us to create more convincing and sustainable compiler-based data management systems. We attempt to derive our advice for creating such DSL stacks from widely acceptable principles. We have also re-created a well-known query compiler following these ideas and report on this effort.

1. INTRODUCTION

Query compilation has been with us since the dawn of the relational database era: IBM's System R employed query compilation in its very first prototype, but this approach was quickly abandoned in favor of query *interpretation* [15]. Recently, query compilation has returned to the limelight, with commercial systems such as StreamBase, IBM Spade, Microsoft's Hekaton, Cloudera Impala, and MemSQL employing it. Academic research has also intensified [33, 2, 52, 56, 64, 53, 54, 55, 50, 84, 19, 62, 44, 6].

We can argue that, despite all this recent work, the state of the art in the design of query compilers lags behind the programming languages and compilers research field. To the best of our knowledge, all existing query compilers are *template expanders* at heart. A template expander is a procedure that, simply speaking, generates low-level code in one direct macro expansion step. While a query interpreter *calls* an operator implementation used inside a query plan, the template expander essentially *inlines* the operator code in the plan, for each operator, to obtain low-level code for the entire plan. We restrict our study to the actual compiler component of a possibly larger data management system. For instance, a system which first parses SQL into query plans and optimizes these Selinger-style, then feeds such plans to a compiler which generates LLVM bytecode, which is in turn lowered to machine code by LLVM, is still a template expander if that core compiler component – into which all of the DBMS engineering team's compiler efforts went – is sufficiently primitive, even if more than two languages and abstraction levels are present in the system as a whole.

Template expansion is a robust and intellectually accessible concept, but it has a number of drawbacks. The System R team reports that query compilation was abandoned in favor of interpretation since query compiler code was hard to maintain (cf. [15]). More fundamentally, template expanders make it impractical to support a range of sophisticated optimizations since multiple code transformers (with different optimization roles) have to be composed and inlined in all possible ways and orderings, causing a code size explosion in compiler code bases. Furthermore, template expanders make cross-operator code optimization impossible [50][1].

To illustrate the above-mentioned code explosion further, consider the example of a template expander that is to support two transformations: 1) pipelining (i.e. removing the need to materialize intermediate results between query operators) and 2) data-structure specialization (i.e. adapting the definition of a data structure to the particular context in which it is used). In order to perform these two optimizations together, one has to implement every combination of their respective cases. For example, if each optimization handles 3 different cases, one has to create a template expander with 9 cases to handle their combination. In general, the code complexity grows exponentially with the depth of the stack of desired transformations. Figure 1a illustrates this code explosion.

System R's initial query compiler as well as Hekaton are confirmed template expanders (cf., [15] and a private communication with the Hekaton team). While academic work makes numerous contributions to the practice of query compilers, it is fair to say that creating a query compiler that produces highly optimized code is a formidable challenge due to the needs to work with various Domain-Specific Languages (DSLs), understand their optimization potentials, and build on the wealth of algorithmic and systems results created by the research over multiple decades. The database community can profit from further tools and techniques from the compilers field.

In this paper, we provide, to the best of our knowledge, the first principled methodology for building query compilers.

We create tools and techniques to increase the *modularity* of query compiler components, to manage the complexity of these systems. Instead of using template expansion to directly gener-

[1]The key contribution of [64] is to show how a push operator interface can eliminate the need for cross-operator fusion transformers, making template expanders produce faster code.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2915244

ate low-level code from a high-level query plan, we propose **progressively lowering the level of abstraction** until we reach the lowest level, and only then generating low-level code. Each level of abstraction and each associated optimization can be seen as independent modules, enforcing the principle of *separation of concerns*. There are two kinds of code transformations, optimizations (where the source and target DSLs are the same) and lowerings. By supporting optimizations on every abstraction level, we obtain real power – moving to a lower-level DSL tends to expand the code size and there is a tradeoff between the search space for optimizations and the granularity of the DSL. Optimizations such as join reorderings are only feasible in high-level DSLs, while register allocation decisions can only be expressed in very low-level DSLs. We propose to use a stack of multiple internal DSLs, one for each such abstraction layer. We attempt to derive our advice for creating such DSL stacks from easily acceptable principles.

Returning to the above example, we can define a third intermediate abstraction level, a *data-structure aware DSL*, to place between the source and target languages. Pipelining transforms a high-level query plan to that intermediate language which has explicit constructs for the operations on the hash-table and list data structures. Then, data-structure specialization transforms the program from this language to low-level code by using an appropriate implementation of each data structure based on its context. Using this intermediate DSL and stepwise lowering, there is no longer any need to consider every combination of the two optimizations, as they no longer interfere by manipulating the same expression. As a result, the complexity of the query compiler code base is more manageable, as demonstrated in Figure 1b.

Creating a sophisticated query compiler is a challenging undertaking, and there are a number of results from the PL and compilers research communities that can help. We explore the key technical challenges in need of well founded solutions, and attempt to gather the most relevant ideas and solution approaches from the PL and compilers communities for easy digestion by database researchers. Specifically, we look at the choice of intermediate languages, how to maximize the separation of concerns among compiler optimizations and lowerings, implementation design choices, and several transformations expressible using this approach.

Throughout this paper, we use the Scala language for our examples and for embedding our DSLs, but nowhere does this paper specifically depend on this choice of programming language.

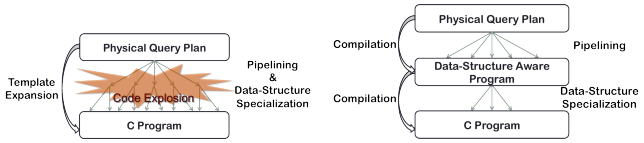
This is an atypical paper in that our main contributions are hard to experimentally validate. Our insights are based on building four distinct compiler-based data management systems over the past seven years, but creating query compilers with a maximally shared codebase for the multitude of alternatives discussed in this paper is beyond the resources of any research group.

Instead, we have re-created a well-known query compiler [50] following our ideas – our DSL stack with stepwise lowering – and report on this effort. We report on the productivity of creating this compiler using a suitable DSL compiler framework [1], and show that we are able to implement all the optimizations of [50] (and more), obtaining at least comparable and often better performance.

2. OVERALL DESIGN PRINCIPLES

2.1 Background

Most database management systems to date use high-level DSLs such as SQL to express queries. Those queries are transformed into optimized physical query plans, and then passed to an engine that interprets them. This approach pays the cost of interpretation overhead, and many low-level optimization opportunities are missed



(a) Pipelining and data-structure specialization are applied concurrently. Hence, we should apply brute-force to all combinations, resulting in code explosion.

(b) Pipelining and data-structure specialization are applied sequentially. Hence, there is no need to consider all combinations, resulting in managed code base.

Figure 1: Handling concurrent optimizations in template expansion and progressive compilation approaches.

(e.g. function inlining), because they are not expressible in the high-level DSL of query plans. These sources of overhead may become especially significant for queries working with in-memory data, where computation is increasingly CPU-bound (rather than I/O-bound). To circumvent these limitations, query compilation aims to generate low-level code from the high-level query plans, allowing specific optimizations to be applied at this low-level representation[2].

However, by directly generating low-level code, we also miss optimization opportunities that are both not available at the high level, and hard to express at the low level – mainly because the locality of code patterns to match is lost. For example, loop fusion (i.e. replacing multiple loops with a single one) is not expressible in query plans, as there is no notion of loop at this abstraction level. Also, it is proven to be NP-complete [21, 48] and impractical in imperative languages such as C [26], the usual target of query compilers.

We propose the introduction of intermediate abstraction levels (referred to as *intermediate DSL*s) for expressing such optimizations. This has already been done in other domains, such as signal transforms and linear algebra. In these domains, we can cite the Σ-SPL [26] and Σ-LL [73] intermediate languages, which have been developed in Spiral [67] for expressing loop fusion optimizations.

In this work, we show that by using several abstraction levels and by doing *step-wise lowering* across them, we can express powerful optimizations (such as pipelining and data-structure synthesis) that are hard or impossible to express in existing query compilation approaches. The resulting set of DSLs, ordered from higher level to lower level, is referred to as the *DSL stack*.

2.2 Choosing The DSLs

Although the design space for a DSL stack may seem overwhelming, there are several constraints that make some design choices impractical, or even infeasible. These directions can be discarded. Among these constraints, there is an important principle for designing intermediate abstraction levels related to the expressive power of programming languages, in the sense of Felleisen [23]:

[2]In this work, we consider compilation as a form of *partial evaluation* [38]. A compiler-based approach can always delay some of this partial evaluation to runtime, in which case the nature of the system moves, at least in some aspect, from compilation to interpretation. A strict choice of either one or the other at design time is not, however, necessary. More concretely, there are scenarios where compilation staging, just-in-time compilation, or even interpretation are desirable: if some essential piece of information that can substantially speed up evaluation is not available at compile time, it is better to delay this partial evaluation until that information is available. The classical example is probably statistics for query optimization. Even a compilation-based query engine cannot afford to go without a query optimizer.

Expressibility principle: *Any program written in a given DSL should be expressible in any of the lower-level DSLs as well. Therefore, by lowering the level of abstraction from the former to the latter, one should retain the same expressive power or gain more.*

Note that the converse does not need to hold: a program in a low-level DSL does *not* need to be expressible in higher-level DSLs. Based on the expressibility principle, we can start designing an appropriate DSL stack. For that, we need to answer the following questions: 1) What abstraction levels (DSLs) do we need? 2) On which abstraction level(s) should we put each transformation?

We propose a methodology that naturally answers both questions at the same time: to design a DSL stack, one should start from the simplest stack possible (a high-level DSL that maps directly to a low level one), then iteratively examine the desired transformations one by one and see if the existing DSL stack is sufficient for it or if a new abstraction level needs to be introduced.

Every transformation requires an input program in a *source language* and produces a program in a *target language*. If the source language and the target language are the same, the transformation is called an *optimization*, whereas if the target language is at lower level, it is called a *lowering* transformation. We will see in the next section why the source language is never going to be lower-level than the target language.

Optimizations are subject to the well-known *phase-ordering problem* [80]: most optimizations can produce opportunities for other optimizations that apply to the same language, but it is not clear how to order them optimally. This problem is still a topic of research in the Programming Language community, and no definitive answer has been formulated yet.[3] To mitigate this, we recursively apply optimizations inside the same abstraction level until we reach a fixed point,[4] where either no more optimizations can be applied or the application of an optimization does not yield structurally different code. On the other hand, lowering transformations are not subject to this issue, because they change the abstraction level of the program so previous optimizations are no longer applicable.

Note that lowering transformations should always be applicable, irrelevant of the input program. Otherwise, the transformation chains would be broken. In contrast, optimizations may or may not be applicable, depending on the information and patterns found in the program being optimized.

2.3 Constructing The Stack

For the DSL stack to be well-formed and to maximize the reuse of transformations, we need to ensure that transformations are not redundant. Each lowering transformation translates a DSL to the next lower-level DSL, and each high-level DSL program is, through a sequence of lowerings, eventually mapped to the target language. This requirement is summarized in the following principle:

Transformation cohesion principle: *Between any two DSLs of different abstraction levels, there should be a unique path of lowering transformations translating programs in the higher-level DSL into programs in the lower-level one.*

[3] *Online* (or *local*) transformations do help removing ordering problems for a certain class of optimizations [39]. We provide facilities (cf. [13]) to encode them in our framework, but this is out of the scope of the current paper.

[4] Special care needs to be taken in the design of optimizations to ensure that iteratively applying them leads to termination.

Notice that this does not prevent having several different high-level front-end DSLs, or different low-level target languages. On the other hand, the principle implies that there can be no transformations from a given DSL *a* to a higher-level DSL *b*, as there would also need to be a path down from *b* to *a*, creating a loop, and thus an infinity of lowering paths from *b* to *a*, violating the principle.

Let us consider the case where we need to introduce a new abstraction level, which mainly happens when we have more than one lowering transformation between two particular DSLs. In such a case, we have to split either the source language or the target language into two separate languages. The new intermediate language should follow the expressibility principle, interpolating between the level above and below. Furthermore, the affected transformations should update their source or target languages: optimizations on the source and target languages, and the lowering transformations from the source to the target language. This update in the existing lowering transformations is necessary as, otherwise, the transformation cohesion principle would be violated.

Going back to our working example, consider a query compiler with SQL as the query language, C as the target language, and pipelining and data-structure specialization as transformations. First, we start from a two-level DSL stack, as shown in Figure 1a.

Second, we consider where to place the pipelining transformation. The information required for checking the applicability of pipelining is available in SQL. However, it is very hard to check such opportunities in a low-level language like C. Expressing a pipelined program is not possible in SQL. This requires the expressibility of a lower language, like C in this case. Hence, pipelining is a lowering transformation from SQL to C.

Finally, we examine the existing DSL stack (which is still two levels till now) for adding data-structure specialization. Similar to pipelining, this transformation is a lowering from SQL to C. This is because we need the high-level information available in SQL, but also a way to explicitly represent data structures (not possible in SQL). We now have two lowering transformations from SQL to C. This means we should break one of these languages into two languages to modularize these two transformations. In this case, we break C into two languages: 1) Data-structure-aware C, which is an extension to the standard C language with specific constructs for the data structures of a query engine, and 2) The standard C language. Data-structure specialization can now use data-structure-aware C as its source language and low-level C as its target language. At last, all transformations which had C as their source or target language should be updated accordingly. In this case, pipelining should update its target language in order to follow the transformation cohesion principle. Pipelining can be expressed in data-structure-aware C as well. Hence, the pipelining transformation is now a lowering transformation between SQL and data-structure-aware C. The new DSL stack is shown in Figure 1b.

In the next section, we further discuss the design space as well as compilation concerns for different DSL stacks.

3. DESIGN SPACE

3.1 Imperative vs. Declarative

So far, we have been referring to *high-level* and *low-level* languages without providing a clear definition for these terms. A more precise terminology would have us use *declarative* and *imperative* instead, although these do not always coincide. This dichotomy is not the only criterion to take into account while designing abstraction levels, but it does play a central role, as we will see. In a declarative language, programs are close to specifications of the results we want to compute, while in an imperative language, de-

Paradigm	Advantages
Declarative	✓ Concise programs ✓ Small search space of equivalent programs ✓ Simple to analyze and verify ✓ Simple to parallelize
Imperative	✓ Efficient data structures ✓ Precise control of runtime constructs ✓ More predictable performance

Table 1: Comparison of declarative and imperative languages

tails about how the result is computed are made explicit. The latter usually involves the use of side-effects like mutation. Table 1 summarizes the important differences between these two approaches.

Different mixes of declarative and imperative features in a DSL are amenable to different optimizations, and require different compilation techniques. Programs in high-level languages are smaller, and the search space of semantically equivalent programs is therefore more manageable. This allows us to use search strategies similar to the ones query optimizers use. These result in global optimizations that have more impact on the resulting program than optimizations on low-level programs, since an expression in a high-level program corresponds to many low-level expressions.

Moreover, high-level languages usually target specific domains – in our case, query processing. As a result, they need not be as complete and powerful as general-purpose languages: like SQL, they do not even need to be Turing-complete [59]. This allows compilers to provide more guarantees, and to perform better reasoning, static analysis, and optimization [83]. For example, it is feasible to statically reason about the runtime cost of a SQL query with typically good or acceptable accuracy. This is *not* possible in low-level Turing-complete languages without actually running the program [51].

Since low-level programs are larger, there is a large number of equivalent low-level programs. This makes the use of search techniques [51] impractical, preventing the global optimization of these programs. Instead, optimizing compilers usually perform only local optimizations (typically, so-called "peephole" optimizations, that only consider a small part of the program at a time).

In the rest of this section, we see how to design our intermediate DSLs to integrate more or less imperative and declarative features, depending on the type of optimizations we want to perform on them. Then, we discuss different possible Intermediate Representations (IRs) to encode programs written in these DSLs.

3.2 DSL Design and Optimization

Functional languages are a particularly important class of declarative languages. We use functional features to represent the declarative aspects of our DSLs for several reasons: first, these languages are easy to understand and to reason about, possibly reusing frameworks developed in the PL community [14]; second, functional constructs integrate well with imperative ones [65], supporting our goal of progressively turning programs from a declarative to an imperative form; finally, the hybrid Scala programming language, which we use in our framework, naturally allows such a mix of functional and imperative code.

Although functional programs are executable, they introduce a relatively heavy performance penalty and do not allow enough control on runtime constructs. This prevents the fine tuning of program performance. For example, standard functional data structures are immutable, and do not allow in-place modification of their elements, requiring copies instead. Without aggressive optimizations like deforestation [86], this translates into many unnecessary allocations and copies. Therefore, we need to lower these high-level declarative constructs into specific, optimized imperative representations. Such representations are close to the underlying architecture, providing opportunities for fine-grained performance tuning.

On the other hand, imperative programs that exhibit poor performance are harder to optimize. For example, detecting loop fusion opportunities is much harder in imperative programs than in functional ones. If the loops in an imperative program are not manually fused by the programmer, there is little chance that the compiler will be able to fuse them automatically. The reason is that optimizing imperative programs with side effects is notoriously hard [7], chiefly because the compiler has to reason about aliased mutable memory locations, a problem that has been shown to be intractable in general [69]. This has implications on low-level optimizations.

For example, consider a `for` loop written in C that only manipulates local variables. Modern compilers know how to optimize such constructs in near-optimal, almost unbeatable ways. But as soon as one introduces non-trivial function calls inside the loop, the compiler's bets are off and many automatic rewritings become impossible. Consider the following code:

```
for (int i = 0; i < size(str); i++) { str[i] = 'X'; }
```

In general, a compiler must not assume that it is safe to extract the call to `size(str)` out of the loop, because the way it is computed could be influenced by assignments performed inside the loop body. In fact, for the particular case where `str` is a simple C string, the compiler cannot know that we are not going to override the string termination character while iterating over the string (which would change the result of a subsequent call to `size`).[5] This would prevent the compiler from implementing resetting the characters of a string as an efficient `memset` instruction.

In this context, human expertise becomes important again. Based on domain-specific knowledge, one can make assumptions that low-level C compilers cannot, even after expensive program analyses that try to recover high-level information from the low-level code. Since we progressively lower abstraction one step at a time, we can exploit as many optimization opportunities as possible along the way. Our framework allows the expression of effectful computations, but can still reason about code that is known to be pure, and our transformations can leverage invariants that are known to hold in the intermediate DSLs.

Next, we discuss various intermediate representation choices for each abstraction level.

3.3 Intermediate Representation

Compilers usually convert input programs, given as text strings, into an *Intermediate Representation (IR)* which contains all essential information available about the program after parsing[6]. Optimizing compilers use IRs to facilitate the definition and application of optimizations.

The simplest form of an IR is an *Abstract Syntax Tree (AST)*. In database management systems, an AST represents a query in relational algebra or its physical plan. This representation is sufficient for performing algebraic rewrite rules on such algebraic languages without variable bindings. Examples of transformations include pushing down selections or changing the join order in relational algebra. As an example, Stratego [85] uses ASTs for its IR.

[5] A special case could be added in the compiler to handle this particular example, but this approach does not scale, as the general problem is undecidable.

[6] Observe that *different* DSLs or abstractions levels may use the *same* IR as their underlying data structure; however, the information (DSL constructs) encoded using these IRs may vary significantly.

However, there are optimizations that require more sophisticated IRs. For example, *Common Subexpression Elimination* (CSE) involves sharing leaves among sub-trees, which calls for a DAG rather than a tree representation, or equivalently a language with variable bindings (cf. [3, 30]).

Furthermore, as the language becomes more complicated (e.g., through the introduction of mutability), programs start requiring *data-flow analysis* [43, 42, 47] to check for the applicability of optimizations. Performing data-flow analysis for every independent optimization that requires it results in more analysis passes than necessary and is difficult to implement, debug, and maintain [79]. Hence, the need to replace plain ASTs with a data structure that stores the result of data-flow analysis, as a better representation on which to apply optimizations.

Several IRs have been proposed in the PL community for simplifying data-flow analysis and optimization. These IRs encode data-flow information by converting a given program into a *canonical representation*. For example, in all of these IRs, every subexpression in a program is bound to a local variable, and reassignment to these variables is not allowed (i.e. they are immutable) [14].

All DSLs in our stack use administrative-normal form (ANF) [25]. This form is better illustrated with an example. Consider the following expression which represents a part of an aggregation:

```
agg1 += R_A * R_B
agg2 += R_A * R_B * (1 - R_C)
agg3 += R_D * (1 - R_C)
```

After converting to ANF, all operators should accept either a constant or a local variable. Hence, every arithmetic operation is converted to a version which uses the local variable bound to its arguments. The ANF representation of this expression is as follows:

```
val x1 = R_A * R_B
agg1 += x1
val x2 = 1 - R_C
val x3 = x1 * x2
agg2 += x3
val x4 = R_D * x2
agg3 += x4
```

While converting a subexpression to an immutable variable, one can look up the mappings between the existing variable bindings and their corresponding subexpressions. If there is already a subexpression with the same operator and the same arguments, then the existing bound variable can be reused. This provides CSE *for free*. In the previous example, the expression `R_A * R_B` is computed once and is used twice for both `agg1` and `agg2`. The same happens for `1 - R_C` in both `agg2` and `agg3`. Observe that this optimization is only one of the advantages of the ANF form. Appendix A presents more information on ANF and several other well-known IRs in the PL community.

Finally, we encode additional information (other than data-flow information) about the expressions in the IR. There are cases in which we need some high-level information about the expressions which is not available in the current abstraction level. Such information can be guided through *annotations* from a higher level of abstraction. Note that since ANF assigns a unique symbol to each subexpression, this process is simplified by keeping a hash-table from these unique symbols to their associated annotations.

4. DSL STACK

In this section, we present the construction of our DSL stack by progressively refining a naïve two-level stack consisting of query plans and C, respectively the source and target languages commonly used in existing query compilers. We progressively add intermediate abstraction levels to perform new optimizations in a modular way, as described in Section 2. Finally, we demonstrate the straightforward addition of a new front-end language which reuses the lower abstraction levels already defined in the DSL stack, thereby benefiting from all transformations that apply to them.

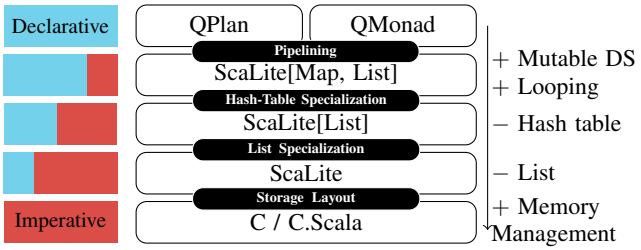


Figure 2: DSL Stack for Query Compilation

Scala DSLs. Figure 2 shows the final DSL stack. On the right, we show which constructs are added and removed by each intermediate DSL. Scala is the implementation language of the framework, and we use a subset of its features to encode intermediate DSLs as well. We refer to the main subset as ScaLite. We write ScaLite[X,Y,...] to denote ScaLite augmented with features X, Y, etc. QPlan and QMonad are Scala DSLs used as two possible front-ends for the DSL stack. ScaLite[Map, List] and ScaLite[List] are intermediate, data-structure-aware DSLs used for specializing the abstract *hash table* and *list* data structures. As we will see, we enforce more restrictions on higher-level DSLs, so that for example, restricted mutability makes ScaLite[Map, List] in fact *less* expressive than ScaLite[List], where mutable Maps can be implemented directly. In ScaLite, all data structure are completely implemented in the language itself, but the memory is assumed to be managed by the garbage collector of a runtime system (e.g. the JVM). Finally, C.Scala is another Scala DSL that expresses C constructs, and in particular memory manipulation constructs, so a program written in C.Scala is equivalent to a C program, modulo a straightforward syntactic transformation (called *stringification* or *unparsing*).

The advantage of using Scala to host these DSLs (we say they are *embedded* [36] in Scala) is that we benefit from its compilation tool-chain: parsing, type-checking, execution and debugging. Indeed, each DSL is executable as a Scala program, with low performance but improved debugging possibilities. Note that these DSLs could be designed in other programming languages (e.g. *quoted* DSLs [63] in Haskell) or as external DSLs if one is willing to build the compilation tool-chain from scratch.

Example Query. We use one query as a running example for demonstrating transformations. The query is shown in SQL and expressed in each intermediate DSL (with Scala syntax) in Figure 4.

4.1 Two-Level Stack (QPlan & C)

This is the two-level stack corresponding to existing query compilers, which are template-based.[7] The high-level DSL is an algebraic representation of query operators. A *query optimizer* typically finds the best query plan for a given SQL query, and produces a program in this declarative DSL. The low-level DSL is an architecture-dependent language that can express implementation details useful for tuning performance. Typical choices include C and LLVM IR.

QPlan. The QPlan DSL contains query plan operators typically encountered in various commercial database systems, including semi-, anti- and outer joins. These operators are sufficient for expressing a large class of SQL queries, including the 22 TPC-H [81] queries.

[7] A state-of-the-art database system will often manipulate SQL, relational algebra and basic query plans before handing the result to the query compiler, but these are not seen by the query compiler, and are thus not considered in this paper.

C.Scala. C.Scala is an extension of ScaLite with basic memory management constructs (e.g. `malloc` and `free`) and memory referencing constructs (pointers and pointer arithmetic). We use the GLib data structures to represent dynamic `Array`s and sorted lists (as binary search trees).

Transformations. At this stage, we perform pipelining while producing C.Scala code, either by pulling [31] or pushing [64] data. With this approach, we remove many materialization points, which results in improved data locality and I/O costs. This transformation is discussed further in Section 5.1.

4.2 Three-Level Stack (+ ScaLite)

We saw in the introduction that this two-level stack with pipelining is not appropriate for adding a transformation that affects constructs that are also affected by pipelining. This is because both transformations interfere by manipulating related constructs at the same time (single compilation stage). Here, we want to introduce memory-management and layout optimizations. To resolve the problem as suggested in Section 2, we add a new intermediate DSL that can express pipelining, but does not contain memory management constructs yet.

ScaLite. The core of ScaLite is the simply-typed λ-calculus, which does not have recursion and mainly consists of constructs for function abstraction (a.k.a. λ abstraction) and for function application (invoking a function with an input parameter, possibly another function). ScaLite additionally supports control-flow constructs such as `if` statements and bounded loops (loops for which we statically know the maximum number of iterations). This DSL is not a purely functional language, as it also supports variables that are either immutable (as in `val x = e; f(x)`) or mutable (as in `var x = e1; f(x); x = e2`). It supports user-defined records and three data structures: fixed-size arrays, dynamic arrays, and sorted lists.

These make ScaLite a powerful enough low-level language satisfying the expressibility principle. The restrictions (bounded loops, no recursion) simplify program analysis.

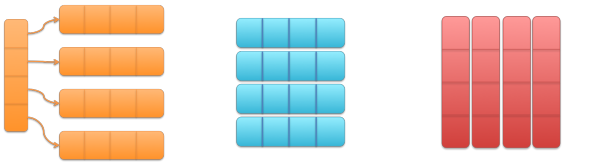
Transformations. While lowering ScaLite programs to C.Scala, we enhance memory management. For example, we use memory pools to preallocate intermediate records. By using statistical information about the input, a worst-case estimate of the cardinality of elements is used to preallocate a memory pool, which obviates the need to perform a system call when allocating new memory. Another memory-management enhancement is to specialize the memory layout of data structures. For example, depending on the context, we represent an array of records either as: 1) an array of pointers to structs – a *boxed* [88] *layout*; 2) an array of structs [84]; or 3) a struct containing one array for each record field – a *columnar layout* [37, 75], which often has a positive impact on cache locality. These data-layout representations are demonstrated in Figure 3.

Figure 4g shows our working example in C.Scala. Line 1 explicitly specifies the representation of an array of records as an array of pointers to records. In line 2, we use the `malloc` function to allocate the array. In the rest of the program, we use arrows (`->`) to access the fields of a referenced record, as in C.

4.3 Four-Level Stack (+ ScaLite[Map, List])

Now, consider adding a transformation for data-structure specialization, which requires a data-structure-aware DSL, as explained in Section 2. Such a DSL has specific constructs for representing the operations of a set of data structures.

Hash tables and lists are two essential data structures for query engines [68]. Hash tables are used for implementing the aggregation and hash join operators, while lists are used for storing intermediate collections of records. There are two kinds of hash tables we are interested in: `HashMap`s associate every key to a single value, and are used for expressing aggregation operators; `MultiMap`s associate every key to a set or list of values, and are used for expressing hash join operators.



(a) Boxed layout (b) Row layout (c) Columnar layout

Figure 3: Different data-layout representations.

These data structures are specialized to efficient implementations, depending on the context in which they are used, which requires a prior analysis phase. This is not possible to do with naïve expansion strategies, like with C++ templates or C macros. It could also be interesting to consider other specialized data structures such as indexed trees to extend the DSL further. This can be useful for other workloads, which we leave as future work.

ScaLite[Map, List]. This DSL is an extension of ScaLite with the `HashMap`, `MultiMap`, and `List` data structures, as well as operations defined on them. For simplifying program analysis, we ensure that for every program in this DSL, the following invariant holds: hash-table data structures are not allowed to contain mutable elements, which means that the records we put into these hash tables should be immutable. This is because if we were allowed to read *and* write fields of an element obtained from a hash table, inferring the access patterns for this hash table would become significantly more complicated. Another way of expressing this invariant is to state that the DSL does not allow *nested mutability*.

Transformations. At this level, we can perform optimizations that use hash tables, such as *string dictionaries* [10]. This optimization maps string operations to integer operations and is discussed further in Section 5.3. Also, hash-table specialization is performed while lowering ScaLite[Map, List] programs. Access patterns are analyzed before this transformation in order to make informed *materialization* decisions and push some computations to a pre-processing phase, which is explained in more details in Section 5.2.

Figure 4d shows our working example in ScaLite[Map, List]. Implementing a hash join is done in two phases: the first phase (lines 3-12) iterates over the elements of the first relation and *builds* a hash table which groups these elements based on their join key; in the second phase, the algorithm *probes* the elements of the second relation and iterates over the corresponding elements of the first relation using the constructed hash table.

4.4 Five-Level Stack (+ ScaLite[List])

Directly translating hash tables to arrays is not necessarily optimal. We would like to translate them to lists first, so we can reuse the fine-grained, context-dependent lowering already defined that converts lists to arrays. Sometimes, it is better to lower lists to linked lists, whereas in some other cases, it is better to lower them to arrays. In order to do that, we add an abstraction level similar to the one above, but without hash tables.

ScaLite[List]. ScaLite[List] is also built atop ScaLite, but only adds constructs related to Lists. However, to encode `MultiMap`s using arrays of lists, we need to relax the restrictions imposed on ScaLite[Map, List]: if nested mutability was forbidden, there would be no way to express `MultiMap`s in a useful way, because we

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

(a) The example query in SQL.

```
AggOp(
  HashJoinOp(
    SelectOp(R, "name", EQ,"R1"),
    S, "sid", "rid"), COUNT)
```

(b) The example query in QPlan.

```
R.filter(r =>
   r.name == "R1"
 ).hashJoin(S)(r => r.sid)(s => s.rid)
 .count
```

(c) The example query in QMonad.

```
1  val hm = new MultiMap[Int, R]
2
3  for(r <- R) {
4    if(r.name == "R1") {
5
6
7      hm.addBinding(r.sid, r)
8
9
10
11   }
12 }
13 var count = 0
14 for(s <- S) {
15   hm.get(s.rid) match {
16     case Some(rList) =>
17       for(r <- rList) {
18         if(r.sid == s.rid)
19           count += 1
20       }
21     case None => ()
22   }
23 }
24 return count
```

(d) The example query in ScaLite[Map, List].

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


    MR(r.sid) += r




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

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

  }
}

return count
```

(e) The example query in ScaLite[List]. Note that List is a mutable data structure.

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

  var r = MR(s.rid)
  while(r != null) {
    if(r.sid == s.rid)
      count += 1
    r = r.next
  }
}

return count
```

(f) The example query in ScaLite.

```
val MR: Array[Pointer[R]] =
  malloc[Pointer[R]](R_GROUPS)
for(r <- R) {
  if(r->name == "R1") {
    if(MR(r->sid) == null) {
      MR(r->sid) = r
    } else {
      r->next = MR(r->sid)
      MR(r->sid) = r
    }
  }
}
var count = 0
for(s <- S) {

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

return count
```

(g) The example query in C.Scala.

Figure 4: Representations of a query in different DSLs.

would not be able to update the set of elements associated with a particular key incrementally (a capability that was hidden behind the `MultiMap` interface of ScaLite[Map, List]).

Transformations. We perform list specialization while lowering from ScaLite[List] to ScaLite. Consider the case in which we lower lists to linked lists. In a typical scenario, lists are used for holding records. In that case, we use *intrusive linked lists*, which store the next pointer of each list node in the records themselves. This removes one level of indirection caused by the separate allocations of the container nodes and the records. On the other hand, since we are working with ScaLite, which only has bounded loops, it is sometimes possible to perform worst-case size analysis and obtain an estimate of the maximum cardinality of some lists. We consequently lower them to native static `Array`s, instead of linked lists. This way, we benefit from the existing array layout optimizations provided for ScaLite down the DSL stack (e.g. columnar layout).

Figure 4e shows the working example in ScaLite[List]. Lines 1-2 contain the lowered representation of the `MultiMap` data structure. Line 7 shows the implementation of the `addBinding` method using the `+=` method of List. Line 16 shows how we lower the `get` method of `MultiMap` by accessing a bucket in the lowered array.

4.5 Collection Programming Front-end

We refer to *collection programming* as the practice of preferring generic operations defined on collections like lists and associative maps (`filter`, `groupBy`, `sum`, etc.) rather than writing them out as loops. The user base of *collection programming* APIs is growing. These APIs improve the integration of applications with database back-ends by making them more seamless [61, 34, 33]. Thus, it makes sense to consider a DSL with a collection programming API as an alternative front-end for a query compiler.

QMonad. The QMonad DSL is a functional language inspired by Monad Calculus on lists [11, 12, 87], Query and Monoid Comprehensions [35, 82, 22] and other collection programming APIs like Spark *RDD*s [89]. In addition to standard collection operators (such as `map`, `filter`, `fold`, etc.), this DSL contains different join operators including semi-, anti-, and outer joins.[8]

Transformations. As we discussed in Section 3.2, declarative and functional languages are not appropriate for performance tuning. It is thus necessary to compile and optimize programs written in QMonad before executing them. Thankfully, by simply lowering those programs to ScaLite[Map, List], we can reuse the transformations provided by the lower level DSLs of our stack *for free*. To produce even faster code, we should perform pipelining to remove materialization points (pipeline breakers). This transformation has a similar effect to what we do for QPlan, by pushing or pulling data. Section 5.1 gives more detail about it.

Figure 4c shows our working example in QMonad. The `filter` method is a higher order function that corresponds to the selection operator in relational algebra. It takes the selection predicate as a parameter. The first parameter of the `hashJoin` method is the second relation, and the second and third parameters indicate the join keys of the first and second relations, respectively. The `count` method returns the number of elements in a list.

4.6 Extensibility

One of the main advantages of our DSL stack design is its extensibility in various dimensions. First, as we just saw, one can use another algebra as the front-end by replacing the front-end DSL (here, QMonad and QPlan). By providing a lowering transformation from the new algebra to one of our intermediate DSLs, the existing infrastructure generates optimized C code for that new front-end.

Second, user-defined functions (UDFs) can be added to input queries. There are two approaches for doing so: 1) expressing them in terms of our low-level DSLs – in this case, we miss high-

[8] Map and join expressions are expressively redundant with nested fold expressions, but represent an important performance choice, and are hard to reconstruct from folds.

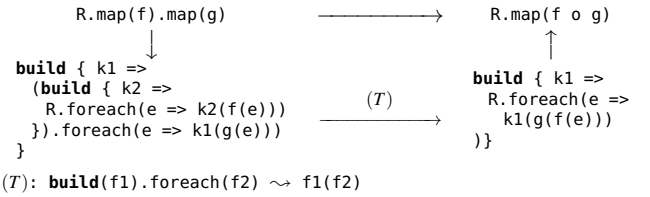


Figure 5: A simple example of loop fusion using short-cut fusion.

level optimization opportunities that could apply to them; 2) adding UDFs as constructs in a high-level DSL, and defining lowerings to immediately-lower DSLs in the appropriate phase. This approach works best if the user provides additional information (by using annotations, which was desribed in Section 3.3) about the additional language construct (e.g. their side-effects).

Third, we can change our target language without any need to change the higher-level DSLs, and still benefit from the optimizations provided in those higher-level DSLs. The only thing we need to do is to provide a lowering from ScaLite (or a higher-level DSL) to our desired target language, and then *unparse* the generated IR in a similar way as we unparse C.Scala to C. This approach works well as long as the underlying architecture is not changed. The extensibility of our DSL stack in the case of changing the target architecture (e.g. using a multi-core architecture instead of a single-core one) is discussed in Section 8.

5. TRANSFORMATIONS

In this section, we detail three transformations that were previously introduced, and which are expressed using the proposed DSL stack. First, we present the pipelining transformation used to improve data locality. Second, we present data-structure synthesis, which specializes and materializes data structures to improve query execution performances. Finally, we provide more details about the string dictionaries.

5.1 Pipelining – From Fusion to Push Query Engines

A query written using QMonad consists of chained invocations of higher-order functions such as `map`, `filter`, `flatMap`, etc. After naively generating low-level code for such queries (e.g. using template expansion), the generated code typically contains: 1) intermediate list constructions and destructions; and 2) loops corresponding to each higher-order function. Creating intermediate lists causes space and time overheads due to unnecessary allocations.

Loop fusion removes these overheads by removing the need to create intermediate lists. This is because in the fused version of the loops, the elements are pipelined from one operation to the next. Moreover, by merging several loops into a single loop, a single traversal is performed, reducing the iteration overhead.

The literature contains a very well-defined set of algebraic rewrite rules for Monad Calculus [11]. These rules are sufficient to express loop fusion for the Monad Calculus subset of QMonad. However, fusion is also needed outside of that subset. Since we add constructs to the language, we need to add the corresponding loop fusion rewrite rules. For this, it is important to be aware of relevant research that has been conducted in the PL community. In fact, it turns out that QMonad with n constructs needs $O(n^2)$ loop fusion rewrite rules, which is not scalable [27]. Ideally, one should need only $O(n)$ rewrite rules for a DSL with n constructs. Furthermore, there are cases in which the fusion of two operators in QMonad is not expressible in QMonad itself. Figure 4c shows a program that exhibits this property: the fusion of `filter` and `hashJoin` cannot

```
class QueryMonad[T] {
 /* These methods are both consumer and producer */
 def map[S](f: T => S): QueryMonad[S] = build { k =>
  for(e <- this) k(f(e))
 }
 def filter(p: T => Boolean): QueryMonad[T] = build { k =>
  for(e <- this) if(p(e)) k(e)
 }
 def hashJoin[S](list2: QueryMonad[S])
  (leftHash: T => Int)
  (rightHash: S => Int): QueryMonad[(T, S)] = build { k =>
  val hm = new MultiMap[Int, T]()
  for(e <- this)
   hm.addBinding(leftHash(e), e)
  for(e2 <- list2) {
   val key = rightHash(e)
   hm.get(key) match {
    case Some(list1) =>
     for(e1 <- list1)
      if(leftHash(e1) == rightHash(e2))
       k((e1, e2))
    case None => ()
   }
  }
 }
 /* This method is only a consumer */
 def count: Int = {
  var result = 0
  for(e <- this)
   result += 1
  result
 }
}
```

Figure 6: Producer-consumer encoding of QMonad operators. Note that in Scala, `for(e <- R) f(e)` is the same as `R.foreach(e => f(e))`.

be expressed using a single QMonad operator. This transformation needs to be expressed using operations provided by a lower level DSL. Hence, for these cases loop fusion is no longer an optimization but a lowering transformation (refer to Section 2).

Deforestation [86] is a well-known technique in the PL community that is used to remove intermediate data structures. This is the approach we use for performing loop fusion. There are several implementations of this technique, among which *short-cut fusion* (known as *cheap deforestation* or *foldr/build fusion*) [27, 28] has been proven to achieve pipelined query execution [35]. This approach requires defining every operator in the language using two primitive combinators: 1) a `build` combinator that *produces* a list; and 2) a `foldr` combinator that *consumes* a list. This way, the number of rewrite rules needed for QMonad with n constructs become $O(n)$ (the implementations of the n operators using the `build` and `foldr` combinators). Loop fusion is then achieved by eliminating adjacent occurrences of `build` and `foldr`.

We implement a variant of short-cut fusion [40] in which every operator is expressed using the *church-encoding of lists* [66], or *tranducers* [72] . For simplicity, we use the `foreach` operator and side effects, instead of the pure `foldRight` operator. This transformation is implemented as a lowering step from QMonad to ScaLite[Map, List]. Figure 6 shows the implementation of a few QMonad operators using `build` and `foreach`. Inlining this high-level implementation leads to pipelining transformation.

Figure 5 shows a simple example in which short-cut fusion is used to apply loop fusion (long path at the bottom). Transformation (T) is what allows us to transition from the code at the bottom-left corner to the code at the bottom-right. This example shows that short-cut fusion has the same impact as the corresponding algebraic rewrite rule from Monad Calculus (short path, on top).

Figure 4d shows the code resulting from the example in Figure 4c, after the pipelining. The `filter` operation is fused with the first loop of the `hashJoin` operation, which is responsible for

creating a hash table based on the join key of the first relation. The `count` operation is fused with the second loop of `hashJoin`, which is responsible for iterating over the second relation and probing relevant partitions from the hash table. Our observations show that short-cut fusion has the same effect as the push-engines proposed in [64]. This is not a surprise, since every operator in the latter is modeled after a producer/consumer pattern, which directly corresponds to the foldr/build model described above.

After pipelining the query engine and lowering it to ScaLite[Map, List], there are new optimization opportunities to be applied on the resulting mutable data structures, as we saw in Section 4.3. Next, we discuss how to synthesize specialized data structures from ScaLite[Map, List] programs.

5.2 Specialized Data-Structure Synthesis

A pipelined query uses mutable data structures to perform in-place updates, instead of cloning the list every time a single element is updated. The resulting program is no longer in QMonad (or QPlan), but has been lowered to ScaLite[Map, List], and is thus no longer purely functional. This makes optimizations harder to express than in higher level, purely declarative DSLs, mainly because of the presence of side-effects. To resolve this issue, we enforce several restrictions on this DSL, which facilitate program analysis. For example, by not allowing nested mutability, we simplify side-effect analysis. This analysis helps identifying data dependencies among statements, after which we can safely reorder them without changing the semantics of the program.

`MultiMap`s, which are used to implement hash joins, can be specialized depending on the way they are used. For example, under circumstances described below, and if there is a one-to-one relationship between a write operation and its corresponding read, we can remove the `MultiMap` altogether. In Figure 7a, we iterate over a relation `R` and add its tuples into a `MultiMap`. We then access the relevant partitions of `R` while iterating over a second relation `S`. Instead of these two steps, we would like to directly access the elements of `R` while iterating over `S`. For it to be safe, such a transformation requires reordered statements to be free of data dependencies related to the two iterations, outside of read/write dependencies on the elements of the `MultiMap` we want to elide.

However, naïvely removing the intermediate `MultiMap` and substituting its read operation with an iteration over `R` clearly will not improve performance (it is equivalent to a nested loop join). Figure 7b shows the result of that naïve transformation. We can see that for each element of `S`, instead of iterating over the relevant tuples in `R`, we iterate over the whole relation. In order to correct this, we can materialize `R` based on the join key `sid`. First, we make sure `R` is not an intermediate relation, but an input relation (otherwise, the transformation is not applicable). Then, at query loading time, we materialize an array of lists which is indexed based on `sid`. As a result, we can now iterate only over the relevant parts of `R`, as is shown in Figure 7c.

Furthermore, in case `sid` is a primary key, the materialized data structure can be specialized further: since there will only be one tuple associated with each key (by definition of a primary key), there is no need for buckets anymore: a one dimensional array is sufficient, instead of an array of lists. We remove the corresponding bucket iteration in the main loop. Figure 7d shows the resulting code, in case the join key is a primary key.

Data-structure synthesis is not limited to `MultiMap`s. `HashMap`s, which are used to implement aggregations, can also be specialized. In that case, we synthesize materialized data structures which partition the `HashMap`s based on their grouping key, automatically inferring the grouping indices.

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

Table 2: Mapping of string operations to integer operations through the corresponding types of string dictionaries. x and y are string arguments which are mapped to integers.

5.3 String Dictionaries

Operations on non-primitive data types, such as Strings, incur a very high performance overhead. This is true for two reasons: There is a function call associated with such operations, and most of these operations typically need to execute loops to process the encapsulated data types. Hence, such operations significantly affect branch prediction and cache locality.

For strings, we use String Dictionaries [10] to remove their abstraction overhead. One dictionary is maintained 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. Then, at query execution time, string operations are mapped to their integer counterparts, as shown in Table 2. This mapping allows to significantly improve the query execution performance, as it completely eliminates underlying loops and, thus, significantly reduces the number of CPU instructions executed.

Special care is needed for string operations that require ordering. For example, Q14 of TPC-H needs to perform the `startsWith` string operations with a constant string. 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 all input data is already materialized, and thus after computing the list of distinct values, we can then sort this list lexicographically, and then assign to the string attributes the integer value of the index in the corresponding sorted list for using during query execution. More specifically, the constant string is 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 this string. Then, this range is used for lowering the operation, as shown in Table 2. This *two-phase* string dictionary allows to map all operations that require some notion of ordering.

Finally, it is important to note that string dictionaries, even though they significantly improve query execution performance, have a negative performance impact on data loading. In addition, string dictionaries can actually degrade performance when they are used for primary keys or for attributes that contain many distinct values. Thus, one should avoid using string dictionaries in such cases.

6. PUTTING IT ALL TOGETHER – THE DBLAB/LB QUERY ENGINE

We have implemented the DSLs of our multi-level stack and the associated transformers in DBLAB,[9] a framework for building efficient database systems via high-level programming. Using the components provided by DBLAB we have re-created the LegoBase [50] query engine. We refer to this re-implementation as DBLAB/LB.

First, developers write their queries in DBLAB/LB using one of the front-end languages of DBLAB/LB (QPlan or QMonad). DBLAB/LB uses the Yin-Yang [41] framework to construct the

[9] http://github.com/epfldata/dblab

```
val hm = new MultiMap[Int, R]
for(r <- R) {
  if(r.name == "R1")
    hm.addBinding(r.sid, r)
}
var count = 0
for(s <- S) {
  hm.get(s.rid) match {
    case Some(rList) =>
      for(r <- rList) {
        if(r.sid == s.rid)
          count += 1
      }
    case None => ()
  }
}
count
```

(a) The optimized version of the query in ScaLite[Map, List].

```
/*
The iteration over the first
relation is moved to the
next step.
*/
var count = 0
for(s <- S) {


  for(r <- R) {
    if(r.name == "R1")
      if(r.sid == s.rid)
        count += 1
  }

}
count
```

(b) Naïvely removing the `MultiMap` abstraction in ScaLite[List].

```
/* Precomputation */
val MR: Array[List[R]] =
  // Indexed R on sid

/* Actual Query Processing */
var count = 0
for(s <- S) {

  val rList = MR(s.rid)
  for(r <- rList) {
    if(r.name == "R1")
      if(r.sid == s.rid)
        count += 1
  }

}
count
```

(c) The optimized version of the query in ScaLite[List] when `R.sid` is a foreign key. Note that List is a mutable data structure.

```
/* Precomputation */
val MR: Array[R] =
  // Indexed R on sid

/* Actual Query Processing */
var count = 0
for(s <- S) {


  val r = MR(s.rid)
  if (r.name == "R1")
    if(r.sid == s.rid)
      count += 1

}

count
```

(d) The optimized version of the query in ScaLite[List] when `R.sid` is a primary key.

Figure 7: Representations of an example query after applying pipelining and data-structure synthesis.

corresponding IR from these queries. As already discussed in Section 4, the front-end languages are progressively lowered and optimized until they reach the abstraction level of the C programming language. Finally, the generated C programs are compiled using any traditional C compiler such as CLang or GCC. At this point, our stack has generated a stand-alone executable for the given query, which includes data loading and data processing and whose execution produces the final query results. The overall DSL code base (without the optimizations, whose lines of code are presented in Section 7) is around a thousand lines of Scala code.

DBLAB heavily relies upon the functionality provided by SC ("Systems Compiler") [1], a *generic* DSL compiler framework. SC provides a complete tool-chain for easily defining DSLs and the corresponding transformations as well as a number of *general-purpose* optimizations out-of-the-box. Note that applying one of the optimizations mentioned throughout this paper does not necessarily lead to performance improvements for a given query. In general, finding the combination of optimizations that leads to optimal performance is a very hard problem; for this reason, the SC DSL compiler does not try to automatically infer the optimal combination of optimizations for each incoming query. Instead, SC was designed so that it provides full control over the compilation process to the DSL developers, while hiding the complicated internal implementation details of the compiler itself. Like DBLAB/LB, SC is also written in Scala, and currently consists of around 27K LoC.

In this work, we extend the library of optimizations provided by SC with several *domain-specific* optimizations which we apply on and across the presented DSLs using the interfaces provided by SC. These domain-specific optimizations can be found in previous query compilation approaches, like those found in the HyPer [64] database, the HIQUE [56] query compiler and the LegoBase [50] query engine. More specifically, the DBLAB/LB DSL stack provides support for: 1) A push-based query engine [64], 2) Operator Inlining [64], 3) Specialization of hash-table data structures [50], 4) Control flow optimizations to improve branch prediction and cache locality [64], 5) String Dictionaries [10] [10], 6) Optimization of memory allocations, by converting *malloc* calls to using memory pools instead and, finally, 7) Standard compiler optimizations like Partial Evaluation, Function Inlining, Scalar Replacement, DCE and CSE [50].

[10]All performance numbers reported for LegoBase in [50] were obtained by including string dictionaries.

7. EXPERIMENTAL RESULTS

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 compiling the generated programs throughout our evaluation we use version 2.9 of the CLang compiler with the default optimization flags. Finally, for C data structures we use the GLib library (version 2.42.1).

For our evaluation we use TPC-H [81], a benchmark suite which simulates data-warehousing and decision support; it provides a set of 22 queries which represent actual business analytics operations to a database with sales information. These queries have a high degree of complexity and express most SQL features.

As a reference point for all results presented in this section, we use the LegoBase query engine [50], an 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 uses *generative programming* [70, 77] and compiles the Scala code to optimized, low-level C code for each SQL query. 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. LegoBase already significantly outperforms both a commercial in-memory database and an existing state-of-the-art query compiler.

Our evaluation is divided into three parts. First, we present experimental results which demonstrate that by utilizing our multi-level DSL stack, developers can build a query engine that matches or even significantly outperforms the LegoBase system. This is because, by splitting optimizations across different abstraction layers and separating concerns, it becomes easier to detect performance bottlenecks that are evasive otherwise. Note that some optimizations of the SC DSL stack are not compliant with the TPC-H rules. For this reason, we also present results for a TPC-H compliant set of optimizations[11]. Second, we examine how much memory footprint is required for running the optimized queries (generated with DBLAB/LB 5) and report on the compilation times. This is impor-

[11]To obtain this TPC-H compliant configuration we have disabled: 1) String Dictionaries 2) Data-Structure Partitioning 3) Automatic Index Inference and, finally, 4) Removing unused table attributes.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
LegoBase	168	108	195	283	2220	100	209	462	447	488	105	281	604	188	134	1326	75	245	371	495	669	191
DBLAB/LB 2	5936	1510	6176	12162	4904	580	4815	20069	27686	5500	797	3763	1846	1832	1633	16962	23765	6329	3075	1629	18636	1320
DBLAB/LB 3	1298	749	4978	10779	3514	289	2714	20069	27686	3411	339	3068	1002	1238	955	14956	18471	3055	3075	938	11595	885
DBLAB/LB 4	177	58	178	141	159	64	109	56	628	433	59	311	860	39	97	4285	38	198	64	194	385	91
DBLAB/LB 5	177	58	117	141	158	46	109	20	537	433	59	120	591	12	27	723	11	196	19	167	385	91
TPC-H Compliant	1298	611	4099	5628	2194	290	2745	6012	19944	3524	80	2567	855	446	1022	4285	4795	4228	2693	427	12070	900

Table 3: Performance results (in milliseconds) for TPC-H (scaling factor 8) for (a) the generated optimized C code of LegoBase [50], (b) the generated C code of DBLAB/LB using an increasing number of levels in our DSL stack and, finally, (c) the TPC-H compliant DSL stack.

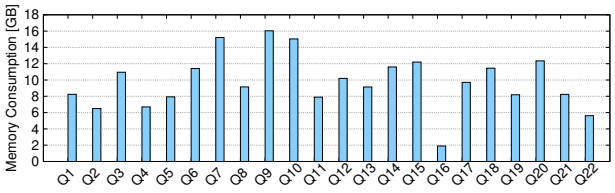


Figure 8: Memory consumption for generated C code of TPC-H.

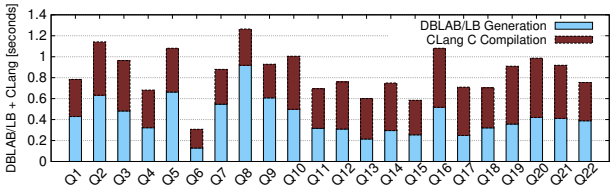


Figure 9: Compilation time for generated C code of TPC-H.

tant, as both these metrics are important indicators about whether our approach scales to large datasets and whether it is usable in practice. Finally, we are addressing the question of how much implementation effort is required for introducing additional abstraction layers or DSLs to an existing stack. We do so by reporting how many lines of code each optimization requires.

7.1 Performance Evaluation

In this section we experimentally evaluate the performance of the DBLAB/LB query compiler. Table 3 presents experimental results for all 22 queries of the TPC-H benchmark for both our system (while incrementally introducing DSLs and their corresponding optimizations as presented in Section 4) and LegoBase [50]. We observe the following three things.

DBLAB/LB achieves a speedup of up to 23× (Q8), with an average performance improvement of 5×. More specifically, for 20 out of 22 queries, DBLAB/LB significantly outperforms the state-of-the-art LegoBase system. This improvement was achieved by introducing optimizations related to removing intermediate materializations that have a significant negative performance impact at query execution. These materialization points (and other performance bottlenecks) were not easy to be detected in the complicated generated code of the original LegoBase system. In addition, we have introduced automatic inference of database indices, based on database statistics, a technique that provided significant performance improvements and was not expressed as a compiler optimization before. However, we also note that in cases where DBLAB/LB is slower than LegoBase (Q1, Q9), this is because the latter was more aggressively optimizing the data structures, e.g. by partitioning data using a composite set of attributes at data loading time. We plan to investigate such optimization opportunities, which are easily expressible with our DSL stack, in the future.

Second, Table 3 also presents a more detailed break-down of how performance improves as DSLs and their corresponding optimizations are introduced in DBLAB/LB. In general we observe that as more DSLs are introduced, performance can be significantly improved. More specifically, when moving from a three to a four-level DSL stack we get an additional 56× performance improvement. This is because the introduction of the additional DSL "unlocks" many optimization opportunities that were not expressible before with a three-level stack. Observe also that the introduction of additional DSLs does not always improve performance for all queries. For example, when moving from a two-level to a three-level DSL stack, performance is marginally improved (if at all) for 10 out of 22 queries. Yet, the introduction of an additional DSL significantly improves the performance of Q1. Thus, we see that it is definitely possible to express even query-specific optimizations with the DBLAB/LB DSL stack. More importantly, Table 3 clearly illustrates that performance is never negatively affected by the introduction of an additional DSL level, for all 22 TPC-H queries.

Finally, we observe that the TPC-H compliant set of DBLAB/LB optimizations typically leads to a better performance than that of a three-level DSL stack, however it does not outperform the four-level DSL stack. This is because, at the four-level stack we start introducing optimizations that are no-longer TPC-H compliant.

7.2 Memory and Compilation Overheads

Figures 8 and 9 show the memory consumption and compilation times, respectively, for all TPC-H queries (generated by DBLAB/LB with 5 DSL levels). For memory profiling we use Valgrind as well as a custom memory profiler, while the JVM is always first warmed up. Compilation time includes the time that (a) DBLAB/LB requires in order to perform program optimization and C code generation, as well as the (b) CLang compilation time.

First, we observe that the allocated memory is at most twice the size of the input data for most TPC-H queries. Second, the compilation time is divided almost equally between DBLAB/LB and CLang. These results suggest that our approach is usable in practice, as even for complicated, multi-way join queries both the memory footprint and the compilation time remain small. Observe that compilation overheads can be further reduced by generating LLVM code directly, a direction that we plan to investigate in the future.

7.3 Productivity Evaluation

In this section we investigate how hard it is to introduce new optimizations in the DSL stack of DBLAB/LB. This is a very important question, as it directly relates to the adaptivity of our proposed solution to new requirements not covered so far in our work. We address this question by presenting in detail our own experience with developing DBLAB/LB so far. More specifically, Table 4 presents the amount of code lines required in order to introduce new IR nodes for the associated transformations of each DSL in our stack. We make three observations.

First, it becomes clear from this table that new transformations can be introduced in DBLAB/LB with relative small programming

Column Store Transformer	184
Automatic Index Inference	318
Memory Allocation Hoisting	186
Pipelining in QPlan	0
Pipelining in QMonad	303
Horizontal Fusion in QMonad	152
Constant-Size Array to Local Vars	125
Flattening Nested Structs	118
Data-Structure Partitioning	505
Scala Constructs to C Transformer	1294
Total	3185

Table 4: Lines of code of several transformations in DBLAB/LB.

effort. This becomes evident when one considers complicated transformations like those of Automatic Index Inference and Horizontal Fusion[12] which can both be coded for merely $\sim$ 500 lines of code. More importantly, the overall coding effort of programming DSLs and their transformations in DBLAB/LB ($\sim$ 3.7KLoC) is almost the same as that reported by LegoBase ($\sim$ 4.8KLoC [50])[13]. Second, we see that pipelining in QPlan does not require any coding effort, while for QMonad it requires a transformation of $\sim$ 500 lines. This difference is due to the fact that QPlan inherently encodes a push engine; this is however not possible to do with the purely functional characteristics of QMonad. Finally, we observe that around half of the code-base concerns converting Scala code to C; however this is a naïve task to be performed by 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 the low-level languages of DBLAB/LB.

8. OUTLOOK: PARALLELISM

One possible question regarding the extensibility of our approach would be adding parallelism to the query engine. There are many different variants of parallelization for database systems. Here, we focus only on the *intra-operator* (or *partitioned*) parallelism which can be achieved by (a) partitioning the input data of each operator in the operator tree, (b) applying the sequential operator implementations on each partition and, finally, (c) merging the result obtained on each partition [31]. Next, we show how our DSL stack can be enriched with parallelization through demonstrating the modifications needed for the DSLs and the transformations.

First, we present the required modifications for the DSLs in our stack. The parallelization logic is encoded in the QPlan DSL by adding the split and merge operators [60]. As these two operators are not expressible by middle level DSLs, the expressibility principle is violated. To solve this issue, the intermediate DSLs require an appropriate set of facilities for parallelism. This is achieved by adding threading facilities (i.e. forking and joining threads) to the ScaLite DSL. As ScaLite[List], ScaLite[Map, List], and C.Scala are extensions to ScaLite, these DSLs are also enriched with parallelism *for free*. The framework generates parallel code by unparsing the parallel C.Scala constructs to the corresponding C code (e.g. by using pthreads).

Second, we analyze what modifications are required for the transformations. If the queries are not using parallel physical query plans at all (e.g. there is no use of split and merge operators), then no change is required for the transformations. However, the generated code for a query with split and merge operators should use an appropriate set of parallelization constructs, as we described above. To do so, we add the implementation of these two operators using the threading constructs. This implementation leads to adjust the pipelining transformation rules for these two operators, as was described in Section 5.1. The merge operator needs special care based on the class of the aggregation (e.g. `SUM` is distributive and `AVG` is algebraic [32]). Apart from these two operators, there is no other modification needed for the existing query operators. Also, this modification is introduced only once and it is reused for all parallel physical query plans. Furthermore, as the rest of existing transformations are not modified, the generated code for each thread benefits from the optimizations provided for the sequential version of the DSL stack *for free*.

It has been shown that in some cases using shared data structures (which requires a locking mechanism) is better than using private data structures for each thread (which is the approach we demonstrated here) [16]. Also, there is a possibility of using lock-free data structures and work-stealing for scheduling the workload across the worker threads [57]. Finally, although we use hash partitioning, there are a number of other approaches for partitioning the input data (e.g. range partitioning, etc.): the performance of each scheme is dependent on the underlying workload scenario and data characteristics [60]. We plan to investigate these directions by employing program analysis and compilation techniques in the future.

9. CONCLUSIONS

In this paper we argue that it is time for a rethinking of how query compilers are designed and implemented. Current solutions are mainly using template expansion to generate low-level code from high-level queries in one single step. Our experience with systems built using this technique indicates that, in practice, it becomes very difficult to maintain and extend such query compilers over time; the complexity of their code-bases increases to unmanageable levels as more and more optimizations are added.

Our suggested approach advocates modularizing a query compiler by defining several abstraction levels. Each abstraction level is responsible for expressing *only* a subset of optimizations, a design decision which creates a separation of concerns between different optimizations. In addition, we propose *progressively* lowering the high-level query to low-level code through multiple abstraction levels. This allows for a more controlled code-generation approach.

We show that our approach, while introducing multiple abstraction layers, does not in fact negatively impact the performance of the generated code, but instead improves it. This is because it allows for expressing compiler optimizations that are already available in existing query compilers, but also *new* optimizations not available before. More importantly, it does so while actually providing for a great degree of programmer productivity, a property not found in any previous query compilation approach to date.

Acknowledgments

We thank Vojin Jovanovic and Manohar Jonalagedda for insightful discussions that helped to improve this paper. This work was supported by NCCR MARVEL and ERC grant 279804.

[12] To perform a good loop fusion, 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 [9, 29]. Good loop fusion is still an open research topic in the PL community [76, 18, 27].

[13] DBLAB/LB comes with additional optimizations not listed in Table 4; however their combined code size does not exceed 200LOC.

10. REFERENCES

[1] SC - Systems Compiler. http://data.epfl.ch/sc.

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

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[4] A. W. Appel. SSA is functional programming. *SIGPLAN notices*, 33(4):17–20, 1998.

[5] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[7] K. Asai, H. Masuhara, and A. Yonezawa. Partial evaluation of call-by-value λ-calculus with side-effects. PEPM '97, pages 12–21. ACM, 1997.

[8] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A Tracing JIT for a Functional Language. ICFP 2015, pages 22–34. ACM, 2015.

[9] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *ICDT '90*, volume 470, pages 72–88. 1990.

[10] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD '09*, pages 283–296. ACM, 2009.

[11] V. Breazu-Tannen, P. Buneman, and L. Wong. *Naturally embedded query languages*. Springer, 1992.

[12] V. Breazu-Tannen and R. Subrahmanyam. *Logical and computational aspects of programming with sets/bags/lists*. Springer, 1991.

[13] J. Carette, O. Kiselyov, and C.-C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[14] M. M. Chakravarty, G. Keller, and P. Zadarnowski. A functional perspective on SSA optimisation algorithms. *Electronic Notes in Theoretical Computer Science*, 82(2):347–361, 2004.

[15] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. King III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. 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. *CACM*, 24(10):632–646, 1981.

[16] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. VLDB '07, pages 339–350. ACM, 2007.

[17] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *TOPLAS*, 17(2):181–196, Mar. 1995.

[18] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP '07*, 2007.

[19] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware:" big" data, big analytics, small clusters. In *CIDR*, 2015.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[21] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175 – 1193, 2000.

[22] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, Dec. 2000.

[23] M. Felleisen. On the expressive power of programming languages. In *ESOP'90*, pages 134–151. Springer, 1990.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, July 1987.

[25] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

[26] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. PLDI '05, pages 315–326.

[27] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. FPCA, pages 223–232. ACM, 1993.

[28] A. J. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.

[29] A. Goldberg and R. Paige. Stream processing. LFP '84, pages 53–62, New York, NY, USA, 1984. ACM.

[30] G. Graefe. Query evaluation techniques for large databases. *CSUR*, 25(2):73–169, June 1993.

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

[32] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[33] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. SIGMOD 2009, pages 1063–1066. ACM.

[34] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1-2):162–172, Sept. 2010.

[35] T. Grust and M. Scholl. How to comprehend queries functionally. *Journal of Intelligent Information Systems*, 12(2-3):191–218, 1999.

[36] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), Dec. 1996.

[37] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten, et al. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

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

[39] S. Jones. Compiling haskell by program transformation: A report from the trenches. In H. Nielson, editor, *Programming Languages and Systems - ESOP '96*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44. Springer Berlin Heidelberg, 1996.

[40] M. Jonnalagedda and S. Stucki. Fold-based fusion as a library: A generative programming pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pages 41–50. ACM, 2015.

[41] V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky. Yin-Yang: Concealing the deep embedding of DSLs. GPCE 2014, pages 73–82. ACM, 2014.

[42] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[43] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, Jan. 1976.

[44] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *CIDR*, 2015.

[45] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Notices*, volume 30, pages 13–22. ACM, 1995.

[46] A. Kennedy. Compiling with continuations, continued. In *ACM SIGPLAN Notices*, volume 42, pages 177–190, 2007.

[47] K. Kennedy. *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division, 1979.

[48] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320. Springer Berlin Heidelberg, 1994.

[49] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst.*, 7(3):443–469, Sept. 1982.

[50] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[51] Y. Klonatos, A. Nötzli, A. Spielmann, C. Koch, and V. Kuncak. Automatic Synthesis of Out-of-core Algorithms. In *ACM SIGMOD*, pages 133–144, 2013.

[52] C. Koch. Incremental query evaluation in a ring of databases. PODS 2010, pages 87–98. ACM, 2010.

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

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

[55] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDBJ*, 23(2):253–278, 2014.

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

[57] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. SIGMOD '14, pages 743–754, New York, NY, USA, 2014. ACM.

[58] D. Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *FOCS*, pages 460–469, Nov 1983.

[59] L. Libkin. Expressive Power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, Mar. 2003.

[60] M. Mehta and D. J. DeWitt. Managing intra-operator parallelism in parallel database systems. In *VLDB*, volume 95, pages 382–394, 1995.

[61] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. SIGMOD '06, pages 706–706. ACM, 2006.

[62] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. *PVLDB*, 7(12):1095–1106.

[63] S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: Quoted domain-specific languages. PEPM 2016, pages 25–36, New York, NY, USA, 2016. ACM.

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

[65] V. Pankratius, F. Schmidt, and G. Garreton. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In *ICSE 2012*, pages 123–133.

[66] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[67] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[68] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, 2nd edition, 2000.

[69] G. Ramalingam. The undecidability of aliasing. *TOPLAS*, 16(5):1467–1471, Sept. 1994.

[70] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *CACM*, 55(6):121–130, June 2012.

[71] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. POPL '88, pages 12–27. ACM, 1988.

[72] O. Shivers and M. Might. Continuations and transducer composition. PLDI '06, pages 295–307. ACM, 2006.

[73] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. CGO '14, pages 23:23–23:32. ACM, 2014.

[74] J. Stanier and D. Watson. Intermediate representations in imperative compilers: A survey. *CSUR*, 45(3):26:1–26:27, July 2013.

[75] 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. VLDB '05, pages 553–564. VLDB Endowment, 2005.

[76] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. ICFP '02, pages 124–132. ACM, 2002.

[77] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. PEPM '97, pages 203–217, NY, USA, 1997. ACM.

[78] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. POPL '09, pages 264–276. ACM.

[79] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., 2nd edition, 2011.

[80] S.-A.-A. Touati and D. Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 147–156, 2006.

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

[82] P. Trinder. Comprehensions, a Query Notation for DBPLs. In *Proc. of the 3rd DBPL workshop*, DBPL3, pages 55–68, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[83] D. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

[84] S. Viglas, G. M. Bierman, and F. Nagel. Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes. *IEEE Data Eng. Bull.*, 37(1):12–21, 2014.

[85] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. ICFP '98, pages 13–26, 1998.

[86] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, pages 344–358. Springer, 1988.

[87] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[88] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST Interpreters. DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.

[89] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. NSDI'12. USENIX Association.

APPENDIX

In this section we provide more information in two areas. First, we discuss about other IRs introduced in the PL community, and compare those with the ANF IR used in DBLAB/LB. Second, we present more optimizations supported by the DBLAB/LB stack for a number of different database components.

A. MORE ON INTERMEDIATE REPRESENTATIONS

There are several IRs proposed in the PL community which simplify data-flow analysis, chief among them 1) SSA [71, 20], 2) CPS [5, 46], and 3) ANF [25]. Although it has been proven that these IRs are semantically equivalent [25, 4, 45], from a practical point of view there are advantages and disadvantages for using each one. Even in the PL community, there is no consensus on which IR is the best [46, 25, 14]. It is, however, undisputed in the PL community that a simple use of ASTs, the dominant choice in work by the database community, creates the problems mentioned before and is inferior to these three for many uses.

The reasons for using ANF can be summarized as follows: first, ANF simplifies data-flow analysis by allowing a single definition of variables [14], which has a great impact on simplifying optimizations that use data-flow information, such as CSE – indeed, it facilitates the mix of effectful and pure computations, and has been shown to make optimizing transformations and analyses easier to write [8]; second, both ANF and SSA preserve the natural organization of programs (called "direct-style"), making them more understandable than CPS (which uses "continuation-passing style", whereby functions call continuations instead of returning values [25]); third, as ANF is a direct-style representation of λ-calculus [25], there are very well-known frameworks for the analysis and verification of λ-calculus developed in the PL community [58]. Hence, reasoning about compiler optimizations in ANF is simpler [14]; finally, by converting many semantically equivalent programs into a canonical representation, expressing optimizations becomes simpler [49], as there is no more any need to express optimizations for all semantically equivalent cases: it suffices to only express one for the canonical form.

There are other IRs proposed in the literature, which mainly target optimizations for parallel architectures (e.g. PDG [24]), or are meant to help and combine compiler optimizations (e.g. a *sea of IR nodes* [17] and E-PEG [78]). However, these IRs complicate the debugging process of program compilation, and are seldom used in real-world, mainstream compilers [74]. Although we did not consider using these IRs, it would be an interesting direction for our future work, particularly when targeting parallel architectures.

B. DATA-STRUCTURE SPECIALIZATION

Data-structure specialization is crucial for achieving high-performance query execution, as it eliminates the unnecessary abstraction overhead of generic data structures like hash tables. Next, we present in more detail how our system performs such optimizations.

B.1 Indexing and Partitioning

The most important form of data-structure specialization concerns optimizing the data structures that hold the data of the *input* relations. This is true even for multi-way join-intensive queries. To enable several DBLAB/LB optimizations for these structures, developers must annotate the primary and foreign keys of their relations, at schema definition time. DBLAB/LB then creates optimized data structures for those relations, as follows.

First, for each input relation, DBLAB/LB creates a replicated data structure which is accessed in constant time 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. This is easy to do for the relations with integer primary keys in the range of $[1...\#num\_tuples]$. However, even when the primary key is not in a continuous value range, DBLAB/LB currently makes an aggressive system memory trade-off to hold a sparse array for that primary key.
- For composite primary keys, DBLAB/LB does not create a 1D array and instead uses the foreign key for this purpose.

Second, DBLAB/LB 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 the tuples having a particular value for that foreign key. We also apply the same partitioning technique for relations that have primary composite 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 the primary keys.

This technique improves query performance, but trades-off memory consumption as a consequence. 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 use all of them. Thus, DBLAB/LB loads the proper partitioned table based on select and join conditions available in the given query only, and the attributes referenced on those conditions.

In addition to the aforementioned partitioning mechanism, the DBLAB/LB system re-uses this mechanism in order to create indices on attributes of `date` types. These are used to significantly speed up select predicates performed on such attributes.

These indices are automatically constructed at data loading time when an SQL query is given to the DBLAB/LB query compiler. These indices are created by repartitioning the data from the original relation array into a two dimensional array, where each bucket holds the tuples belonging to a specific year (or even a particular year-month combination).

As a result of these indices, the corresponding predicates on date attributes are converted into simple `if` conditions capturing whether the tuples belonging to a bucket should be processed or not. These indices are particularly important for scan operators that process large input relations.

B.2 Hash-Table Specialization

One of the most important data structures in query processing for handling different operators such as hash join and aggregation is the hash-table data structure. By default, DBLAB/LB uses GLib hash tables for generating C code out of the hash-table constructs of the Scala programming language. Independent of the language chosen, however, the overhead of using such generic data structures is significant for at least three reasons: (a) there typically exists data redundancy between the key and value stored in the map, (b) there are numerous function calls occurring for processing a single tuple, and, (c) resizing operations may be needed, so that the data structure remains efficient as more data is placed into it. DBLAB/LB avoids these issues by lowering hash tables to native arrays and linked lists and inlining the corresponding operations.

C. OPTIMIZING MEMORY HIERARCHY ACCESSES

Unused Struct Field Removal: DBLAB/LB provides an optimization for removing struct fields that are not accessed by a particular SQL query. DBLAB/LB optimizes data loading, so that it avoids loading into memory the values for the unnecessary fields. It does so, by analyzing the input code and removing the set of unused fields from the record definitions. This reduces memory pressure and improves cache locality.

The removal of unnecessary fields improves the performance of row layout the most. This is because of two reasons. First, even without field removal optimization the columnar layout is implicitly applying this optimization, as the unused columns are never cached, thus not negatively affecting cache locality. Second, for row layout the field removal optimization basically allows to store the record attributes in a more compact way in memory and hence, improves the cache locality for this layout.

Removing Unnecessary Let-Bindings: The SC DSL compiler uses the ANF IR (cf. Section 3.3) for applying transformations and while performing code generation. Although this IR simplifies analysis, transformations, and code generation for the compiler, it has the negative effect of introducing many unnecessary intermediate variables during code generation. 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, the SC DSL compiler removes any intermediate variable that satisfies the following three conditions: it (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 possible expensive computation). SC then replaces any appearance of this variable later in the code with its initialization value. We have observed that this optimization makes the generated code much more compact, reduces register pressure, and results in significant performance improvements. More importantly, we have observed that the granularity at which this optimization is applied does not allow this optimization to be detected by low-level compilers like LLVM.

Finally, the SC DSL compiler applies a technique known as *Scalar Replacement*. 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, since the field of a struct can be referenced immediately without requiring referencing the variable holding the struct itself. We have observed that this optimization significantly improves cache locality.

D. DOMAIN-SPECIFIC CODE MOTION

Domain-Specific Code Motion subsumes optimizations that remove code segments that have negative impact on query execution performance from the critical path, and instead execute the logic of those code segments during data loading. Thus, the optimiza-

tions in this category trade-off increased loading time for improved query execution performance. There are two main optimizations in this category, described next.

D.1 Hoisting Memory Allocations

Memory allocations can cause significant performance degradation in query execution. By taking advantage of type information available in each SQL query, we can completely eliminate such allocations from the critical path, as described next.

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` expressions in the program, once the latter has been lowered to the C.Scala DSL (cf. Section 4.1). The obtained types may represent either the initial database relations or the intermediate results. Based on this information, the DBLAB/LB query compiler initializes memory pools during data loading, one for each type. For example, for TPC-H queries, DBLAB/LB generates pools for allocating the data types of all TPC-H relations, as well as supporting pools for specific operators, e.g. a pool for allocating the aggregate records.

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, and significantly contributes to improving cache locality. We note that it is not sufficient to naïvely generate one pool per data-type in the order of their appearance, as there may be dependencies between 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 then generating the pools in the proper order.

Finally, the size of these pools is estimated by performing worst-case cardinality analysis on a given query. Currently worst-case estimation is driven by two factors: (a) the collection of statistical metrics like cardinality estimation and attribute skew (e.g. number of distinct values per attribute) as well as (b) the usage of annotations that are specified by developers at schema definition time. As an example of the former, DBLAB/LB gathers information about how many distinct years exist in all attributes of date type (and utilizes this information in the join indices presented in Section B.1), while an example of the latter includes the definition by developers of *1-N* relationships when joining two relations. Based on this information, DBLAB/LB calculates the worst-case selectivity of even multi-way join queries.

As the cardinality is estimated in a worst-case scenario, the transformer may lead to allocation of much more space than what is actually needed. However, we have confirmed that our estimated statistics are accurate enough so that they do not unnecessarily create memory pressure, thus negatively affecting query execution performance. Furthermore, our general framework is not tied to this transformation. Hence, based on the workload we can use different variants of this transformation. For example, one could completely disable this transformation and leave the memory management to the runtime. Alternatively, one could preallocate certain amount of memory pools and in the case of overflowing the memory pool increasing its capacity by a certain factor (e.g. a factor of two) to amortize the runtime complexity.

D.2 Hoisting Data-Structure Initialization

Key-value stores are used in DBLAB/LB to calculate aggregations. During this evaluation, it is necessary to first check whether the corresponding key for the aggregation already exists in the aggregation 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, that exist purely for the purpose of data structure initialization, significantly affect branch prediction and overall query execution performance.

DBLAB/LB provides a transformation to remove such initialization of data structures from the critical path by taking advantage of domain-specific knowledge. For instance, if the key to the aggregation is a relation primary key, we know from data loading time the possible values this key can take. By doing so, we can initialize possible aggregations for each key with the value zero. 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.

It is important to note 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. Particularly for TPC-H, we note two things. First, 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 estimated accurately during data loading. Second, for TPC-H queries the value range is very small, typically ranging up to a couple of thousand sequential key values. These two properties combined allow to completely remove initialization overheads and the associated unnecessary computation for the TPC-H queries. As an example, this optimization improves the performance of TPC-H query 1 by 10%. The workload-specific information is guided through annotations (cf. Section 3.3). These annotations are used for checking the applicability of this optimization. If the given query does not have these properties (e.g. a query without the properties of the TPC-H queries mentioned above) the transformation is no longer applied. In other words, this transformation is smart enough to be applied only to queries which statisfy the mentioned properties.

E. FINE-GRAINED OPTIMIZATIONS

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

DBLAB/LB 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 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 this array access is mapped to). Second, the query compiler provides an optimization to change the boolean condition `x && y` to `x & y` where `x` and `y` both evaluate to boolean, an optimization which, according to our observations, improves branch prediction. However, to make sure that this optimization is semantic preserving, the second operand must not have any side-effect. Finally, the compiler can be instructed to apply tiling to `for` loops whose range are known at compile time (or can be accurately estimated). Based on our observations these fine-grained optimizations, which can be typically written in less than a hundred lines of code, significantly improve query execution performance. More importantly, since they have very fine-grained granularity, their application does not introduce additional performance overheads.