Plan Operator Specialization using Reflective Compiler Techniques

Carl-Philip Haensch, Thomas Kissinger, Dirk Habich, Wolfgang Lehner

Technische Universität Dresden
Database Technology Group
Dresden, Germany
{firstname.lastname}@tu-dresden.de

Abstract: Query-specific code generation has become a well-established approach to speed up query execution. However, this approach has two major drawbacks: (1) code generators are in general hard to write and maintain, (2) code generators lack the ability to deal with custom operators. To overcome these limitations, we suggest to return to the traditional execution approach with precompiled generic operators which are parametrized and composed to query plans at query compile time. Nevertheless, to optimize such plan operators and speed up their execution, we introduce a novel specialization approach using reflective compiler techniques. Employing code annotations and an additional compiler pass, we are able to track and replace low-level load instructions that refer to operator parameters which remain constant during execution time. By dissolving such up-to-now unknown constant variables, the compiler can further optimize the code and is able to determine query-specific optimized operators out of generic operator code. In our evaluation, we show that our approach speeds up the execution of the traditional generic operator approach in terms of execution time without facing the drawbacks of code generators.

1 Introduction

Due to the tremendous increase in the amount of data managed by today's database systems, query optimization is still one of the most challenging issues in database research. In disk-based database systems most of the time is spent for waiting on data from disk and thus, the effective CPU computation time is more or less a negligible factor. However, the ongoing trend towards in-memory databases shifted the bottleneck from disk access to the main memory access, resulting in a higher bandwidth and a lower latency when accessing data objects. Due to these decreased data access times, the effective CPU computation time now takes a significant share of the overall query execution time in such in-memory database systems. Therefore, besides determining an optimal query execution plan, the efficient execution of query plans including all operators attracted attention in the domain of in-memory database systems. The goal of this research is to execute query plans and its operators with less instructions to save as many CPU cycles as possible.

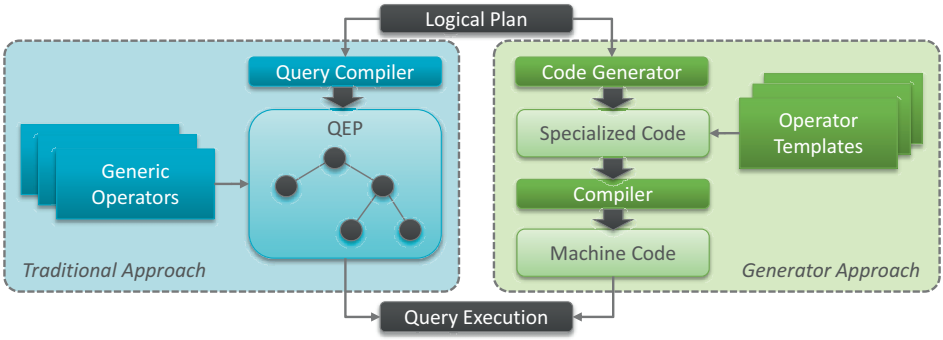


Figure 1: Comparison of Existing Query Compilation Approaches.

From a conceptual perspective, two different approaches for query execution have been developed as illustrated in Figure 1. On the left side of the figure, the traditional approach using a generic query operator code is shown. During each query compilation, a query execution plan (after query optimization) is composed using a set of precompiled generic operators and query-specific parameter settings for each plan operator. Afterwards, the plan is executed without any further optimizations. On the right side of Figure 1, the newly established generator approach is depicted, where a query–specific execution plan is generated using two main components: (a) a code generator and (b) operator templates. The advantages of such generator approaches are: (1) a highly specific query execution code is constructed and (2) this query execution code is further optimized using a compiler such as LLVM. Nevertheless, the significantly reduced execution time requires the costly development of hand-written code generators as well as operator code templates, whereas code generator and templates are hard to maintain and to extend. An additional disadvantage is the fact that the integration and optimization of custom code is challenging and not solved in any way.

To overcome the disadvantages of the generator approach without losing its performance benefits, we suggest in this paper to return to the traditional approach and extend it by a novel specialization concept at query compile-time using reflective compiler techniques. The general advantage compared to existing approaches is our novel architecture that combines the generic way of developing operators with the option to specialize operators fully automatically for a given execution plan which enables an easy development, maintenance and extension of operators. Moreover, this architecture even allows us to easily specialize a custom operator code which is not known at database compile time. To perform this specialization, we introduce code annotations to enable operator developers to mark certain parameters in their code as remaining constant after query compilation. Using these code annotations, the knowledge about the generic operator code and the actual parameters, we are able to further optimize the code. This happens with the help of a novel compiler pass that dissolves the constant behavior of parameters, so that we are able to generate query-specific optimized operators out of generic operator code. As we are going to show, using our constant annotations, we are able to specialize generic operators, which significantly speeds up query execution.

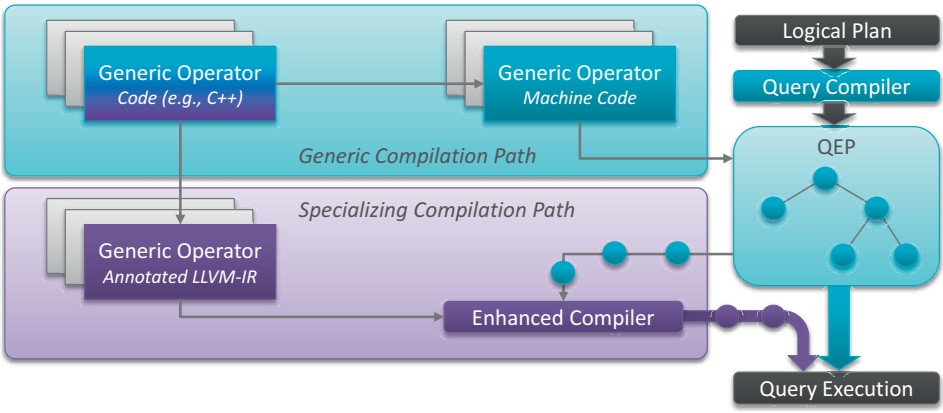


Figure 2: Hybrid Architecture Consisting of a Generic and Specializing Query Compilation Path.

The remainder of the paper is organized as follows: In Section 2, we are going to describe our hybrid architecture consisting of a generic and specializing query compilation path. Based on this overview, we introduce our plan operator specialization in Section 3. In this description, we use a minimal example to illustrate our approach, followed by a database operator example. At the end, we present our new compiler pass for plan operator specialization. Our evaluation in Section 4 is based on micro benchmarks as well as on the star schema benchmark to show the benefits. Before we conclude our paper in Section 6, we review related work in Section 5.

2 Architecture

In this section, we introduce our novel hybrid architecture that employs two different query compilation paths as depicted in Figure 2. The first path is the *Generic Compilation Path*, which takes the traditional way of composing a query execution plan (QEP) out of parametrized instances of generic plan operators. The additional second path is the *Specializing Compilation Path*, which is able to replace generic operator instances of the QEP by specialized query-specific ones. While current architectures either employ only one of those paths – facing their respective benefits and drawbacks – or both paths in a separated fashion, our novel architecture orchestrates both paths into a holistic query compilation framework that is able to choose between both query compilation approaches at the operator instance level. In the following we will detail on both compilation paths.

Generic Compilation Path This compilation path realizes the traditional way of QEP generation, where plan operators are implemented in a highly generic fashion to allow them to deal with all kinds of possible scenarios. Such generic operators are compiled to

executable machine code at database compile time and are instantiated and parametrized at query compile time to fit the actual requirements. Afterwards, all operator instances are composed into an executable QEP. Nowadays, when in-memory databases are becoming the standard, this traditional generic approach faces disadvantages in terms of query execution time because generically implemented operators exhibit a significant instruction overhead. However, common scenarios exist where the generic compilation path still outperforms the specializing compilation path due to the additional costs for invoking a compiler during code generation, e.g., for short-running ad-hoc queries, simple operators, or custom operators which can not be specialized by existing approaches.

Specializing Compilation Path This compilation path is able to transform parametrized instances of a generic operator, which were generated by the generic compilation path, into a specialized query-specific operator. Compared to existing approaches, which require the implementation and maintenance of a separate code generator, our novel approach tightly integrates with the generic compilation path. To do so, the first step is to enable plan operator developers to add annotations to the generic operator code that mark certain parameters as remaining constant after query compilation. As a second requirement, we need a representation of the generic operator code at database runtime that is as low-level as possible, but is still high-level enough to be efficiently optimizable. Hence, we decided to choose the LLVM Intermediate Representation (LLVM-IR) which is an intermediate result of the database compilation process and is used by the LLVM compiler infrastructure to apply several optimization passes. Thus, besides shipping the binary code of the generic operators with the DBMS, our framework additionally requires the inclusion of the LLVM-IR operator code in combination with the aforementioned annotations to allow the database system to reflect on its own operator code. To fully automatically compile specialized query-specific operator code, we take the LLVM-IR of a generic operator, corresponding annotations, as well as the actual parameters of a generic operator instance and feed them into our enhanced LLVM compiler version, which adds an additional optimization pass (cf. Section 3.3). Finally, we replace the generic operator instance by the specialized code if this is beneficial for the overall query execution time. This operator code specialization either happens synchronously at query compile time or asynchronously in parallel to the query execution, where generic operators are hot-swapped as soon as the specialized code is available.

3 Plan Operator Specialization

In this section, we will describe the process of plan operator specialization in three steps. First, we will examine a minimal example to develop a concept of algorithmically specializing generic operator code. Second, we will show with an example that the cognition obtained can also be applied to arbitrarily complex operators. Third, we describe how the specialization is actually implemented using the LLVM compiler framework.

3.1 Minimal Example

Generic operators in general consist of two parts: (1) A generic execution method and (2) parameters. While the generic execution method is already available during database compilation, the parameters are bound at query compile time. Figure 3 shows an operator after query compile time: There is an instance of a generic operator on the left side that already has its parameter bound. The generic execution method called `evaluate` (bottom of the figure) is written agnostically to the actual value of the parameter `x`. When considering the now known value of `x`, an other, smaller and therefore faster code would have the same semantics.

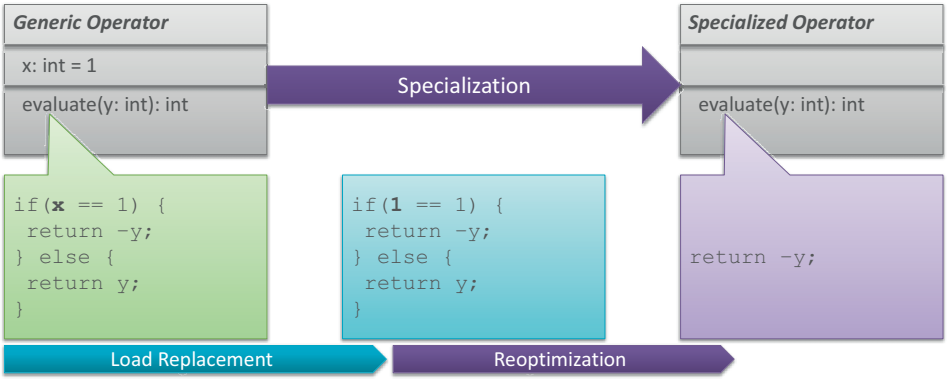


Figure 3: Minimal Specialization Example using A Simple Generic Operator.

High level programming language source code is rather unsuitable for our considerations. Such languages make use of a high variety of concepts. For instance, global variables, array elements, elements of a struct and attributes of an object all describe different concepts, but all describe a piece of memory that can be accessed through load instructions. For this reason we translated the example in Figure 3 into a pseudo assembly language as shown in Figure 4. This pseudo assembly only knows three kinds of instructions: arithmetics, jumps and load/store commands. The `if` condition is translated into a load instruction, a comparison and a conditional jump. The expression `return y` is translated into a load instruction and a jump and `return -y` is translated into a load instruction, an arithmetic instruction and a jump.

The left box of Figure 4 already shows an optimized assembly as it is spilled from a compiler. The problem with the generic operator is that it cannot be further optimized because the load instructions introduce data dependencies that cannot be resolved at database compile time. Since the parameter is known at query compile time, we propose to run the code optimizer of our compiler again over the code after the values of the parameters are known. To overcome optimization-blocking load instructions, we have to replace them by actual values.

After replacing `load x, %b` by `mov 1, %b`, traditional compiler passes are able to

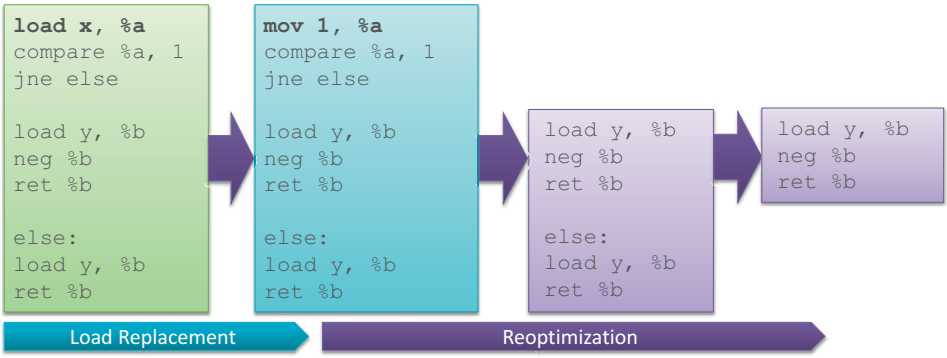


Figure 4: Minimal Example in Pseudo Assembly.

optimize our example even further: The *instruction combining* pass of our stock compiler is able to combine the `mov` and `compare` instruction and directly propagate *equal* instead of computing a comparison. The conditional jump operation `jne` can therefore be removed because the condition is never fulfilled. After removing the jump instruction, the label `else` is never reached and can therefore be removed, too.

In summary, we are able to specialize an operator completely just by using simple code transformation rules. We discovered that replacing load instructions of operator parameters by the values of the parameters offers a method of automatically creating specialized code without the need for writing a custom code generator.

3.2 Database Operator Example

The operator examined in section 3.1 is rather simple. Database operators are much more complex. They contain loops, nested branches and virtual function calls where template based code generators are likely to fail. This is not the case for our approach. We will show that in general lots of code primitives occuring in database operators are load instructions followed by some instructions that depend on the result of the load instruction. To make this tangible, we provide an example in Figure 5, which shows a generic group operator.

For our example we choose the following parameters: `group_attributes` has size 2. The first element has a `column` of `2` and the second element has a `column` of `3`. With the size parameter known, we can construct a specialized version of the loop such as in Figure 6. The method we used is called *loop unrolling*. It is a normal compiler pass that is implemented in nearly all modern optimizing compilers. The only reason why the loop cannot be unrolled was that `group_attributes.size()` was unknown. However, this value is constant and known for this operator instance after the query was compiled. After replacing the load instruction from the `size` field of the vector, the loop has a constant iteration count. The optimizer can now remove all control instructions and insert

```
1  void GroupOperator::execute(Tuple tuple) {
2          Key key = hash_initial_value;
3          for(int i = 0; i < group_attrs.size(); i++) {
4                  key.append(tuple.extract(
5                          group_attrs[i].column));
6          }
7          outputIndex.insert(key, tuple);
8  }
```

Figure 5: GROUP BY Operator Example Code.

```
1  void GroupOperator::execute(Tuple tuple) {
2          Key key = hash_initial_value;
3          key.append(tuple.extract(
4                  group_attrs[0].column));
5          key.append(tuple.extract(
6                  group_attrs[1].column));
7          outputIndex.insert(key, tuple);
8  }
```

Figure 6: GROUP BY Operator Example Code with Unrolled Loop.

the loop body n times. Figure 7 shows instruction-wise how big our savings are.

An interesting situation comes up when accessing arrays inside of loops. The upper example accesses `group_attributess` elements with the unknown index `i`. After unrolling the loop, this index is known and therefore a concrete memory address can be derived. Figure 8 shows the code of the group operator after specialization. Since this operator code (group by) is executed inside of a loop, even small savings in the instruction count yield remarkable speedups in the execution time. `tuple.extract` is a rather simple function. Inlining it would open up even more optimization potential. The following code patterns occur in real database operator code:

Loops with a Constant Number of Iterations Iterations over tuple elements are mostly operation specific and can therefore be specialized. Applying the specialization removes unneeded branching and unnecessary compare instructions. Also, unrolling loops opens new possibilities to access different data in each loop iteration, e.g., type information for each tuple element.

Constant Values (SQL Constants) Queries such as `SELECT x+1 FROM t` contain constants such as `1` which are better to be inlined than fetched from somewhere. Generic operators implement them as load instructions from some constant storage. Inlining these

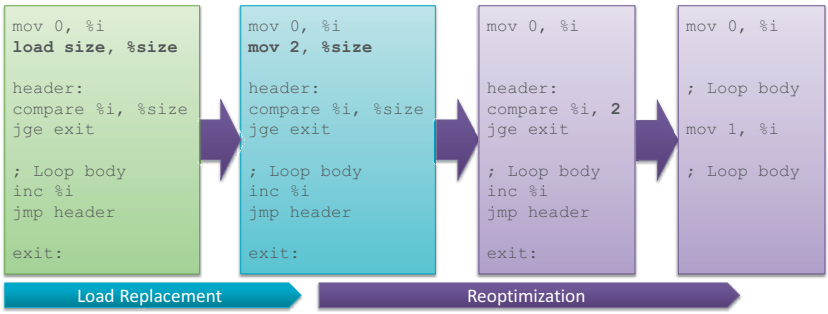


Figure 7: Loop can be Unrolled Because of Resolved Load Instruction.

```
1  void GroupOperator::execute(Tuple tuple) {
2          Key key = hash_initial_value;
3          key.append(tuple.extract(2));
4          key.append(tuple.extract(3));
5          outputIndex.insert(key, tuple);
6  }
```

Figure 8: GROUP BY Operator Example Code with Unrolled Loop and Inserted Parameters.

constants into the machine text also prepares other optimizations such as instruction combining. By using our approach, a SQL expression such as `2+2*2` could be computed during query compile time.

Virtual Function Calls Operators that are built calling virtual functions recursively. With an inlining pass, we can remove call overhead and also have the ability to perform interprocedural optimizations. One example is `tuple.extract` in Figure 8 where inlining the tuple extraction algorithm would open up additional optimization potential. Combined with inlining `key.append` even more aggressive optimizations would be possible. Fetching a function pointer from a virtual function table is a load instruction. We detect this pattern and inline the function call. For ahead-of-time compilers, virtual function inlining is not easy. In the worst case, all code has to be scanned, which is not always possible. Given a specific runtime object, virtual function inlining hence is trivial.

Branches Branching occurs when compiling control flow statements such as `if` and `while`. For branches that depend on operator parameters rather than input, we are able to remove branching and therefore remove unnecessary comparisons, jumps and dead code.

And a lot more Patterns ... Our approach does not handle constants, loop iteration counts, data type selectors or tuple offsets as corner cases. We rather see that replacing load instructions by constants, optimization passes already available in stock LLVM are triggered and specialize the code independent from its structure. In summary, we are able to deal with arbitrary complex generic operator code. All code is compiled to machine code that contains load instructions, jumps and arithmetic instructions.

3.3 Specializing Compilation Path

In this section, we will present our novel compiler pass. In the first subsection, we will give an overview on how this pass interacts with other passes inside the compilation process. In the second section, we will describe the algorithm behind our pass.

3.3.1 The Compilation Process

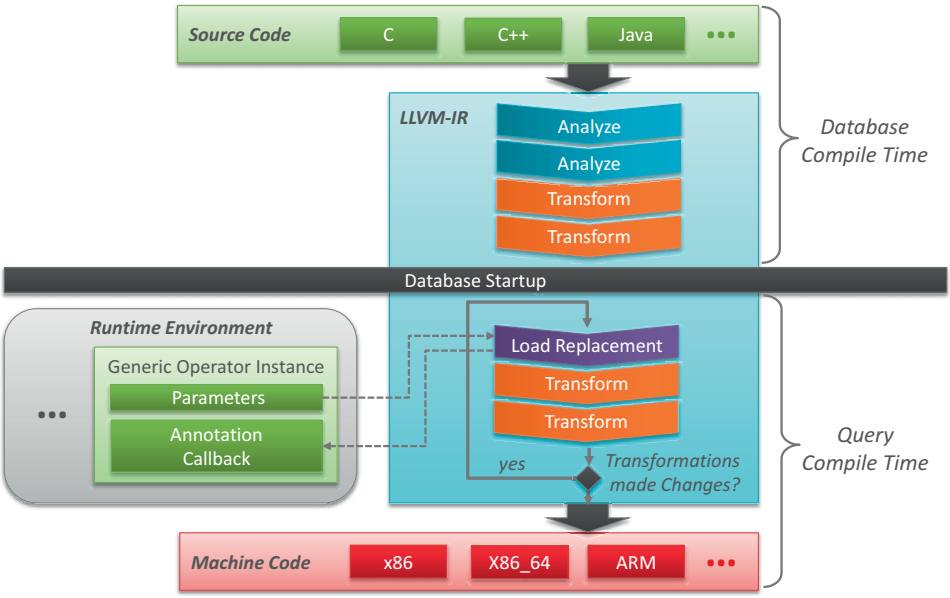


Figure 9: LLVM Compilation Process with Operator Specialization.

The traditional database compilation process is restricted to database compile time. Our approach also involves query compile time. The compiled code of the generic operators in their intermediate representation is kept for further specializations. During each query compilation, this code can be further specialized due to bound operator parameters.

We specialize operators by involving their already bound parameters. This happens by

loading the already optimized generic code (created at database compile time) and running compiler passes over it (Figure 9). Except from our *load replacement* pass we did not add any functionality to LLVM[1]. All optimizations are already known in compiler construction science. The *load replacement* pass opens up new optimization possibilities when traditional compiler passes can not further optimize the code. Therefore we try to run all beneficial traditional compiler passes over the code until they resign. Then we run our novel pass such that traditional passes can continue optimizing the code.

The first pass running over the code is the *load replacement*. We start with this pass because it removes load instructions that blocked further optimizations during database compile time and replaces them with compiler-friendly constants. After that, *instruction combining* can do traditional constant propagation in order to partially evaluate branch conditions and arithmetics. After that, *control flow graph simplification* can remove dead code and reorder the basic blocks to an optimal order.

For more complex operators a different pass order is applied. Operators containing loops have to be handled separately. We also begin with *load replacement*. After *instruction combining* some loop counts are known, so a *loop unrolling* pass can be run. After that, *load replacement* has to be run again because array access inside the loop makes use of the control variable of the loop. A load instruction can only be replaced by a constant when the memory address is constant which does not hold for addresses that are calculated from the control variable. After unrolling, new load instructions can be resolved. After loops are unrolled and their inner load instructions are resolved, *instruction combining* can be run again and *control flow graph simplification* cleans up the specialized code. The overall result after the specialization process is much smaller code. With less instructions, the code fits better into the instruction cache and can be executed faster. To save time, we do not run loop passes on rather simple operators such as arithmetics.

We also run the *virtual function call inlining* pass for functions that call other functions. This pass is placed before the last *instruction combining* pass to reveal further optimizable code patterns to the following passes. It enables us to inline deep expressions like `(2*i) + 1` directly into one function.

3.3.2 Load Replacement Pass

This pass is the key part of our approach. It pushes operator parameters into the generic operator code. In object oriented programming, a method is a function that operates in the context of an object. Most programming languages implement this by passing a parameter `this` to the function. The operators themselves are classes where some of the attributes are operator parameters. To achieve operator specialization, the *load replacement* pass has to be aware of the value of the `this` parameter of the operator method. Load instructions relative to `this` load attributes of the object which can by chance be constant operator parameters. To assure that an attribute of the object is in fact constant, we have to annotate the memory somehow.

[1] *virtual function inlining* is also written by us because we want to inline already optimized operators into their parent operators.

The pass itself iterates over all instructions of a method and looks for load instructions. For each load instruction it tries to resolve the memory address to load from. This is done by reverse traversion of the code. The memory address is calculated from some base value (e.g., `this`) and offsets. If some of the offsets are not constant but depend on a loop control variable or a load instruction, we can not calculate the address. In case we found a load instruction that loads from $mem[\texttt{this} + offset]$, $mem[mem[\texttt{this} + offset] + offset]$ or any deeper indirection, calculating the address is possible. These addresses are constructed from LLVM-IR's instructions `load`, `bitcast` and `getelementptr`, as well as `%this` at the end of the chain. By handling them in a recursive function, we can resolve all memory addresses to attributes in the code. Once the address is calculated from all offsets, we consult our annotation system whether there is a constant at this memory location or not. If this is the case, the load instruction can be executed at compile time by loading the value from runtime memory.

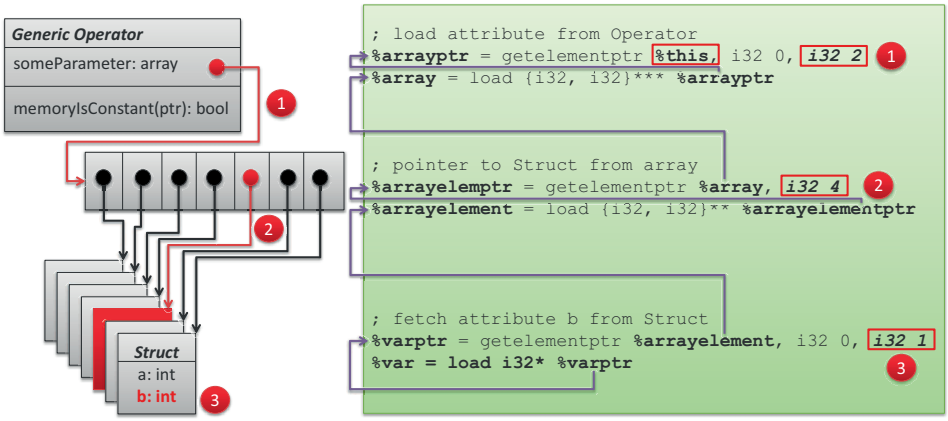


Figure 10: Loading Operator Parameters in LLVM-IR.

Figure 10 illustrates what operator parameters could look like. The operator parameter `%var`, in C `someParameter[4]->b`, is hidden behind two indirect loads. When the value of `%this` is known (and the offset is constant), the address `%varptr` can be computed. Computing the address is implemented by resolving `%varptr` recursively. `load` instructions in indirect memory addressing such as `%arrayelement` have to be executed during pass runtime. This is only allowed when the annotation system marks the loaded memory constant. A `getelementptr` instruction simply adds an offset to a pointer. The then computed address is passed to our annotation. If the annotation assures constantness of the memory at this address, an integer constant can be fetched from the address `%varptr`. This constant is packed into a LLVM value and all occurances of `%var` are replaced by that constant. Subsequent passes will remove the instructions that compute `%varptr` too when they are not referenced any more.

The annotation is managed by the operator (`this`) and is realized through a method called `bool memoryIsConstant(void *mem)` which returns *true* for a given memory

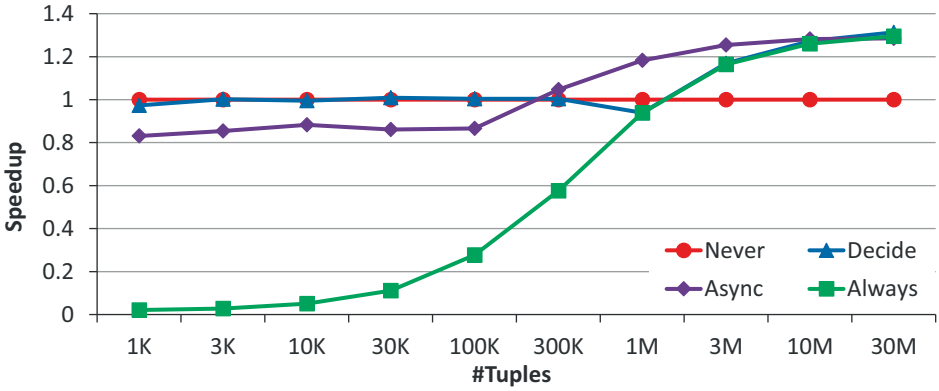


Figure 11: Execution Time Speedup of Ad-hoc Queries for Micro Benchmark with Different Data Sizes and Different Specialization Strategies.

location when there is a constant at this address. This way, the memory is mapped into regions that can change and regions that remain constant. The granularity of this mapping is byte-wise. The annotation system does not care about variables, attributes or data types. It also is not compiled into the LLVM code. Instead, it just manages a mapping that tells at which memory address there is a constant. Determining which address is constant is delegated to the operator objects that know which of their attributes are constant. The address ranges of the attributes are compared to the address queried by the annotation manager.

4 Evaluation

In column stores, operators work vector-wise[RBZ13]. This reduces complexity and makes it possible to generate all variations of operator parameters, such as input types, with templates. However, this is not possible for tuple based operators since the number of variations grows exponentially with the number of elements in a tuple.

We used two benchmarks to evaluate the performance of our specialized code. The first one is a micro benchmark that covers all combinations of the sum aggregation with single value addition. The second one is Star Schema Benchmark (SSB) which is widely used in data base research[OOCR09]. As a test setup, we used DexterDB, which was developed by the database research team of TU Dresden. It is an in-memory database, which has an indexed table-at-a-time processing model[KSHL13]. We implemented our approach into the already optimized operator system of DexterDB and measured what speedup we can achieve with our additional operator specialization.

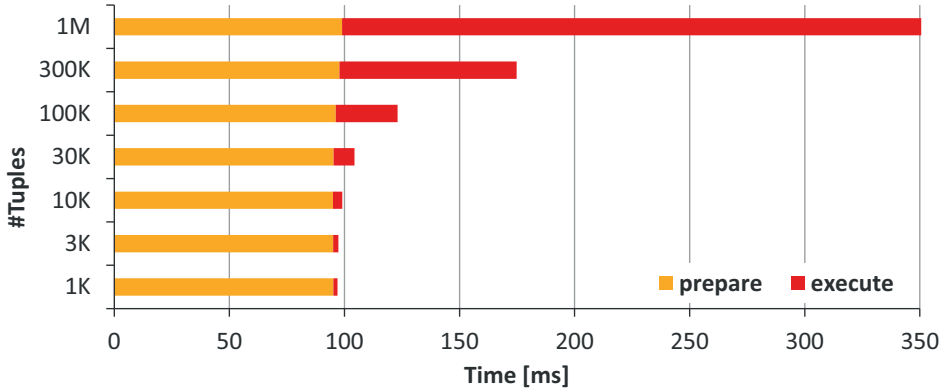


Figure 12: Execution and Prepare Time of Ad-hoc Statements for Micro Benchmark with Different Data Sizes.

4.1 Micro Benchmark

The micro benchmark consists of 5 queries which cover simple aggregation, aggregation combined with arithmetics (several bracketings), aggregation with a condition on each input tuple and a simple join with a table of 5 elements. All queries are executed together to get a median over some common queries. The benchmark varies over the size N of one input table.

All queries are executed 32 times with each of these four specialization strategies: **Never** strategy does never specialize and reflects the unoptimized result of the generic operators. **Always** reflects the result with specialized operators. **Decide** tries to find a balance between execution and preparation effort and only specializes when the input tables are big enough (threshold of N). **Async** runs the specialization in an extra thread and asynchronously swaps methods when the query runs a long time but cancels specialization if the query was faster.

The test machine is an Intel(R) Core(TM) i7-3960X CPU with 6 cores each 3.30 GHz with a total of 12 hyperthreads running linux. L1 instruction and data cache are 32 KiB each. L2 cache is 256 KiB and L3 is 15360 KiB.

One measurement consists of executing all 5 queries 32 times with table size N. On each execution two measures are taken: prepare and execute time. Prepare time is the time from parsing the statement, optimizing and specializing until it is ready for execution. During prepare time no data is touched. Prepare time is also the time it would take when a prepared statement is created. Execution time is the time it took to execute. Executing an ad-hoc query would take prepare+execute time while executing prepared statements would only take the execute time.

Preparing unspecialized queries took about 0.2-0.5 ms in total. However, preparing specialized queries however took about 95-99 ms for all five queries, which is quite costly

in comparison to unoptmized queries. Figure 12 shows that code generation is expensive, which especially concerns small queries. Therefore two additional strategies for handling small queries were introduced: **Decide** and **Async**. As one can see in Figure 11, these strategies give good results for both big and small queries. Figure 11 shows how they combine the advantages of both code paths: the generic execution path and the specialized operator path. While decide ammortizes relatively late, the asynchronous strategy already starts gaining advantage at about 100,000 items. The downside of the **Async** strategy is that each query takes some extra milliseconds for synchronization. **Async** has to stop the specialization thread which is not possible at any time.

4.2 Star Schema Benchmark

The Star Schema Benchmark[OOCR09] (SSB) is a benchmark derived from TPC-H. It aims at benchmarking database products that run classical data warehouse applications. The benchmark consists of 8 tables and 13 queries containing aggregations, joins and selections.

	Generic		Specialized	
Nr.	Prepare Time	Execution Time	Prepare Time	Execution Time
Q1.1	0.20 ms	8,312.01 ms	41.29 ms	4,979.34 ms
Q1.2	0.21 ms	8,734.92 ms	43.43 ms	5,319.47 ms
Q1.3	0.21 ms	8,692.01 ms	45.43 ms	5,357.07 ms
Q2.1	0.20 ms	10,900.00 ms	77.29 ms	9,117.51 ms
Q2.2	0.22 ms	7,758.99 ms	80.77 ms	6,098.14 ms
Q2.3	0.18 ms	5,508.16 ms	78.31 ms	3,907.43 ms
Q3.1	0.22 ms	11,163.30 ms	91.54 ms	8,766.37 ms
Q3.2	0.22 ms	7,285.59 ms	91.87 ms	5,690.01 ms
Q3.3	0.23 ms	5,574.93 ms	91.80 ms	4,131.47 ms
Q3.4	0.22 ms	5,805.91 ms	90.81 ms	4,121.73 ms
Q4.1	0.25 ms	10,382.60 ms	99.21 ms	8,203.17 ms
Q4.2	0.30 ms	9,797.42 ms	112.60 ms	7,742.18 ms
Q4.3	0.28 ms	6,896.63 ms	108.98 ms	5,174.66 ms

Figure 13: Results of the SSB-Benchmark.

The benchmark is executed with scale factor 20 which fits into 32 GiB RAM of the test machine. Figure 13 shows the results of our measurements. When executing queries with a large ammount of data, the average 70 ms of prepare time does not influence the overall result that much. Figure 14 especially shows that, in contrast to the micro benchmark, the difference in speedup between prepared statements and ad-hoc queries is not remarkable for big data queries like those from the SSB. This is also confirmed in Figure 15 where we ran SSB with different scale factors.

Figure 15 shows two SSB queries (Q1.1 and Q2.2) that we benchmarked with different scale factors both with and without indices. Figure 16 illustrates the speedup from Fig-

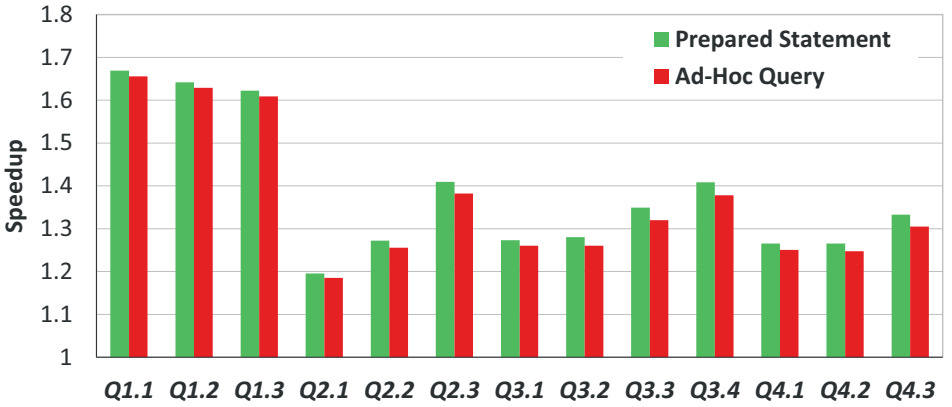
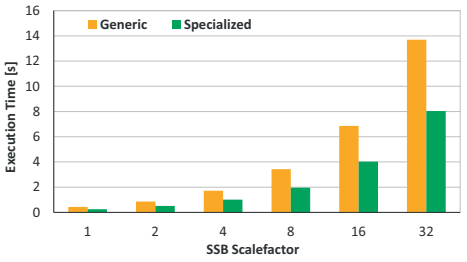
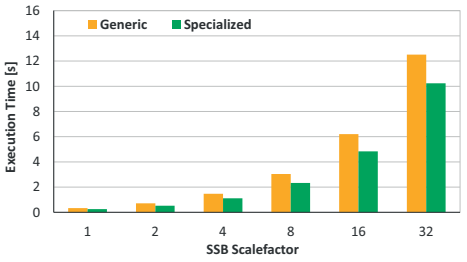


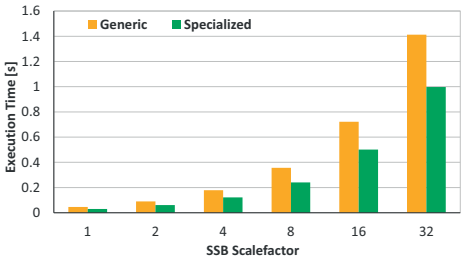
Figure 14: Speedup of Specialized SSB Queries.



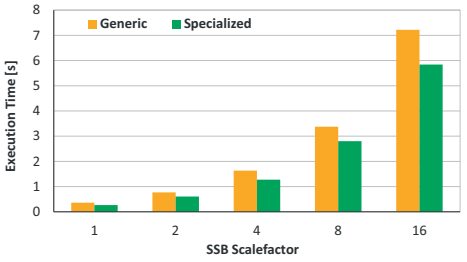
(a) SSB Query 1.1 w/o Indexing



(b) SSB Query 2.2 w/o Indexing



(c) SSB Query 1.1 w/ Indexing



(d) SSB Query 2.2 w/ Indexing

Figure 15: Execution Times of SSB Queries for Different Scale Factors and Storage Configurations.

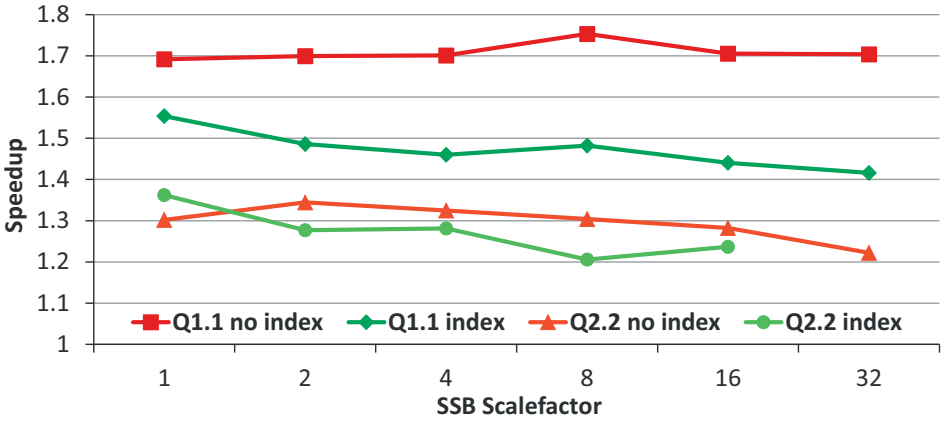


Figure 16: Speedup of Specialized SSB Queries 1.1 and 2.2 w/ and w/o indexing for Different Scale Factors.

ure 15's runs in one chart. While the run of Q1.1 without indexing has a rather constant speedup, speedup sinks in the other runs. Q1.1 does not have joins and therefore just iterates over a big ammount of data. We achieve the highest speedup with this setting. With enabled indexing, speedup is worse but still good. The declining speedup for enabled indexing and Q2.2, which consists of some joins, can be explained with smarter algorithms that avoid raw computations and therefore do not offer large optimization potential.

Our approach achieves a speedup between 1.2 and 1.6 in DexterDB for SSB which is a favorable result. Especially Q1.1, Q1.2 and Q1.3 perform great which can be attributed to their heavy algebraic load. The other queries rely more on multi way joins. With an additional recursion unroller, which we did not write, we could further speed up joins.

5 Related Work

There are various approaches to achieve low-level optimized operators. One approach is to transform and specialize existing code. An other approach is to generate already specialized code using code generators.

5.1 Code Specialization Approaches

Our approach is one instance of partial evaluation[2]. Our compiler pass partially executes generic operator code and generates the leftover specialized code. While we work with annotations for marking constants, there are also approaches to have partial evaluation as a language feature. The Java Programming language[GJSB05] for instance supports the `final` flag for variables which has exactly the semantics of our annotation. A Java Runtime Engine could specialize methods of an instanciated object in a background thread and inline the actual values of the final fields into the method code. The problem with this feature is that it is hard to decide whether a specialization is beneficial. Speculatively specializing code often yields performance penalties. A runtime engine can not determine how often code is executed in general. However, in our domain we have this information and use it to decide whether to specialize or not.

Partial evaluation is something Veldhuizen[Vel99] already described in 1999. Veldhuizen's approach is based on C++ templates which are restricted to integers at compile time. What he predicted is that it will be possible to do partial evaluation based on bind time data. Our approach is even better, since it can do partial evaluation during program runtime.

One approach did come close to ours. Skye Wanderman-Milne and Nong Li[WML14] proposed a complete architecture for code snippet composition based on LLVM called Cloud Impala. Parts of their code is generated by a hand-written code generator while other parts are C/C++ or Python code compiled to LLVM-IR and specialized for specific operator parameters. They make very specialized differenciations between conditionals, constant loads, runtime type information and try to manually specialize prepared code based on variable names. However, we discovered that any of these patterns contains a unresolved load instruction. Based on memory annotation, we can tell which load instructions can be resolved. Therefore we don't need extra variable loaders and can instead read out our runtime memory which makes our solution much more elegant. Also, their approach requires a completely new infrastructure for generic operators, which is not optimal for unspecialized code. Our approach allows further specialization of already optimized legacy generic operators.

5.2 Code Generators

Zhang et al.[ZSD12] introduced the term *micro-specialization* in DBMSs. While most optimization approaches focus on high level components, they focussed on parameterized operators in the classical operator approach. They focussed on variables - typically schema metadata or query-specific constants - that are invariant to the query evaluation loop. Micro optimizations remove unnecessary operations in frequently taken execution paths. The paper states that micro specialization is not possible using traditional compiler techniques

[2]Partial Evaluation means to not evaluate a statement or a piece of code completely. Instead, parts of the code are executed and the code left to execute is returned. For instance partially executing an `if` statement would mean to evaluate the branch condition and return only the code of the chosen path.

since the invariants are only known at runtime. Until this point the paper does not distinguish from this work. The difference is that Zhang et al. work with hand-written code templates while we derive our information from the original source code of the DBMS using compiler techniques. For us, the generic operator code is the template. Also, they are still restricted operator boundaries while we can inline code. They used code templates for some hot code operators which have to be specified by the developer. The developer has to define code snippets that are then assembled to so called bee routines. The gcc compiler is used to create the precompiled bee routines. Constants are inserted with preprocessor macros. For some values, so called magic numbers (1000, 1001, 1002) are used to be replaced by constants after loading the machine text into memory. The bee routines are loaded into memory and are then further specialized with constant values. Integrated into PostgreSQL they got a performance gain of about 12%, in rare cases even 33%. One thing that their paper does better than this paper is the preparation time. Loading a piece of machine code into RAM and replacing some constants is much faster than generating specialized code from the beginning.

Rao et al.[RPML06] implemented a query compiler for the Java Virtual Machine (JVM). They used it in their in-memory database called JAMDB. Besides interpreting a Query Execution Plan they generate java bytecode. Their implementation is said to run faster than the non-specialized C/C++ code from DB2[3]. They implemented both an interpreted engine and a JVM based compiler. This led to double expense of maintaining code and optimizing parts of the code. They admitted that this is harder to code and harder to maintain than simple interpreters.

Neumann[Neu11] proposes a completely new architecture of query processing that tries to keep all values of a processed tuple in CPU registers. This way, highly effective pipelining of multiple operations can be achieved. They used the LLVM framework to merge operations into one machine code function. They used a code generator to build specialized query plans on demand. Their generated code is said to be near optimal assembly code. Their code generator for full SQL-92 operators is about 11,000 lines of code which is not a lot in their perception. We used about 1,000 lines of code with our generalized approach (plus the code of the interpreted operators) which is even better.

The big advantage of our approach is flexibility. While other approaches have to maintain code generators which are harder to write, we can stick to generic operator code. Extending the execution engine with new operators consisting of custom code is possible and easy with our approach since every operator behaves like custom code to us. By using compiler techniques we are able to create optimized code with a minimal invasive programming model. The downside is a higher preparation time. Transforming generic code to specialized code is slower than generating only the needed parts of the optimal query execution code.

[3] It ran with a large bufferpool to have in-memory characteristics.

6 Conclusion and Future Work

We developed a hybrid architecture that can be put on top of traditional generic operator systems. This architecture enables us to specialize operators to their parameters when the benefit exceeds the effort of the specialization. This way we can speed up long running database queries.

Our framework takes the intermediate code of generic operators after database compilation and specializes them to a specific instance of that operator. This is done by resolving load instructions. Since the pointer to the instance of an operator is known to us, we know the memory locations of immutable operator parameters and can therefore resolve load instructions that load from these memory locations. This way, we are able to push additional information about the operator into the generic operator code, which enables further optimizations on the code to traditional compiler passes.

Adopted to a database which runs operator code millions of times, we measured speedups ranging from 1.2 to 2.0 in query processing. With these speedups, we revived the classical operator model and modernized it to meet the needs of modern in-memory databases. An implementation of about 1,000 lines of code is able to specialize arbitrarily complex operators independent of their semantics. This is useful especially when dealing with custom operators.

Our approach solves the same problem as hand-written code generators. However, we provide a much more elegant way to achieve a specialization. DBMS implementors do not have to write code generators from scratch to get speedup. They can just put their generic operator code into our algorithm and our enhanced compiler will specialize it to a certain environment, in this case to bound operator parameters.

This approach is not restricted to DBMSs. Load replacement enables to specialize generic code to any situation that can be encoded in memory. The reason why it is not widely adopted by scripting language interpreters, regular expression interpreters, image filters and other possible applications is the high effort for compilation. A problem with an automation of the specialization, for instance as a feature of a language's runtime, is to decide how often some specializable code is executed, which often depends on the size of the input. Compilation is expensive and speculative specialization can slow down execution when the specialized code is only executed a few times.

References

[Bar09] Edward Barrett. 3c - A JIT Compiler with LLVM. Technical report, Bournemouth University, 2009.

[Dan99] Olivier Danvy, editor. *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*. University of Aarhus, 1999.

[Fan10] *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[KSHL13] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. QPPT: Query Processing on Prefix Trees. In *CIDR*. www.cidrdb.org, 2013.

[LA03] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[MBK02] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[NW08] George C. Necula and Philip Wadler, editors. *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008.

[OOCR09] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, pages 237–252, 2009.

[RBZ13] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1231–1242, New York, NY, USA, 2013. ACM.

[RPML06] Jun Rao, Hamid Pirahesh, C. Mohan, and Guy M. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, page 23, 2006.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[Str07] Bjarne Stroustrup. *The C++ programming language - special edition (3. ed.)*. Addison-Wesley, 2007.

[UL08] Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: a generalization of java's final fields. In Necula and Wadler [NW08], pages 183–195.

[Vel99] Todd L. Veldhuizen. C++ Templates as Partial Evaluation. In Danvy [Dan99], pages 13–18.

[WML14] Skye Wanderman-Milne and Nong Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, pages 31–37, 2014.

[ZNMZ13] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal Verification of SSA Optimizations for LLVM, June 2013.

[ZSD12] Rui Zhang, Richard T. Snodgrass, and Saumya Debray. Micro-Specialization in DBMSes. In *ICDE*, pages 690–701, 2012.