On Another Level: How to Debug Compiling Query Engines

Timo Kersten
kersten@in.tum.de
Technical University of Munich

Thomas Neumann
neumann@in.tum.de
Technical University of Munich

ABSTRACT

Compilation-based query engines generate and compile code at runtime, which is then run to get the query result. In this process there are two levels of source code involved: The code of the code generator itself and the code that is generated at runtime. This can make debugging quite indirect, as a fault in the generated code was caused by an error in the generator. To find the error, we have to look at both, the generated code and the code that generated it.

Current debugging technology is not equipped to handle this situation. For example, GNU's gdb only offers facilities to inspect one source line, but not multiple source levels. Also, current debuggers are not able to reconstruct additional program state for further source levels, thus, context is missing during debugging.

In this paper, we show how to build a multi-level debugger for generated queries that solves these issues. We propose to use a time-travelling debugger to provide context information for compile-time and runtime, thus providing full interactive debugging capabilities for every source level. We also present how to build such a debugger with low engineering effort by combining existing tool chains.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Relational Query Execution, Code Generation, Debugging

ACM Reference Format:
Timo Kersten and Thomas Neumann. 2020. On Another Level: How to Debug Compiling Query Engines. In *Workshop on Testing Database Systems (DBTest'20), June 19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3395032.3395321

1 INTRODUCTION

With the advent of in-memory databases, high-bandwidth solid state drives, and recently also persistent memory [11], high-performance relational query execution engines compile machine code for query execution. This approach creates optimal code for each query and thus makes best use of available computing resources [12]. Consequently, code generating execution engines are able to make the most of the large available bandwidth.

DBTest'20, June 19, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8001-0/20/06…$15.00
https://doi.org/10.1145/3395032.3395321

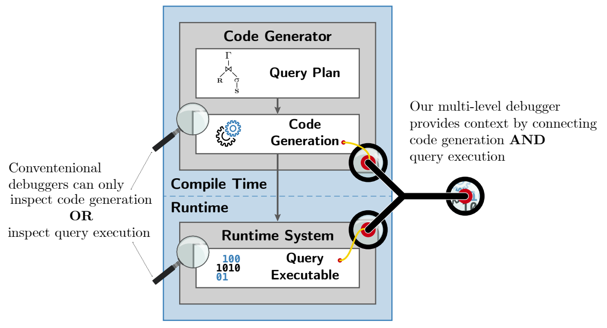

Code Generator
Query Plan
Code Generation
Compile Time
Runtime
Runtime System
100
1010
01
Query Executable
Conventenional debuggers can only inspect code generation **OR** inspect query execution
Our multi-level debugger provides context by connecting code generation **AND** query execution

Figure 1: Compiling relational engines process queries in two steps: *Code generation and execution. Conventional debuggers can only attach to one step, so that debugging execution misses lots of context information. Our multi-level debugger provides this context.*

Query execution in a compiling query engine is done in a two-step process (cf. Figure 1). First, the engine generates code for the query plan. Second, the machine's processors execute this code to compute the query result [15]. For the developer of a compiling engine this two-step process can become a challenge. When, during development, they find their computation results are wrong, they need *debugging tools* to efficiently triangulate the cause of the fault.

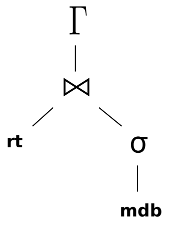
Conventional debuggers support the search of errors by allowing the developer to stop the execution at any point. The developer can then inspect the program state, view the value of variables, explore data structures, and examine the call-stack to decide whether the observed behavior is as expected or already affected by an error. To make this process efficient, the debugger should show the developer a full view of the program state in the source language and the format that the developer wrote it. In other words, the debugger should present the state in terms the developer is familiar with.

In a compiling query engine, however, this integrated experience is not possible with a regular debugger. A compiling engine splits the query execution into the two phases shown in Figure 1: *Compile time*, which generates code for a query plan and compiles it to machine instructions, and *runtime*, which runs the machine instructions to produce the query result. To debug this two-level setup, most toolchains already offer the means to step through either the code generator or the runtime code. However, the *link* between the generated code and the source code that generated it, is *missing*. Without the link the developer is missing most of the query context.

Currently, there are two limitations that cause this disconnect: First, current debuggers are not built for this kind of debugging. GDB, for example, supports only to *stop at one position* in the machine code and map that position to one source location. There is currently no support to handle a second source location that

```
select count(*)
from
  RotatingTomatoes rt,
  MovieDatabase mdb
where
  rt.name = mdb.name and
  rt.rating = mdb.rating and
  mdb.reviews > 10;
```

(a) Query – How many movies receive the same rating in both sources?

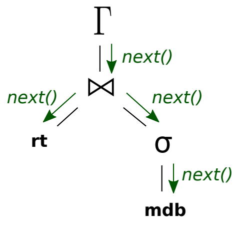


(b) Execution plan.

Figure 2: Example query with execution plan.

```
Tuple JoinOperator::next()
 hashTable.buildFrom(leftChild)
 # Probe with tuples from right side
 while(right = rightChild.next())
  for(left in hashTable.find(right))
   yield left.concat(right)
```

(a) The join operator in a Volcano-style interpreter retrieves tuples from left and right child, passes matches to parent.



(b) Tuple passing between operators by `next()` calls.

Figure 3: Control flow of Volcano-style query processing and implementation of the hash join operator.

generated the first source location. Second, as generating code and running it is a two-step process, there is a *lifetime* issue between multiple source locations. When the debugger stops a multi-level program, it can map the current program state to a source line on the first level. Mapping also to a second source line that generated the first source line is difficult, because the second source line was executed much earlier in time. That means that the current program state does correspond to the first source line, but not to the second. Therefore, the debugger can't use the program state to inspect the call-stack and variables for the second source line. To this day, we are not aware of any debuggers that fully bridge this gap.

In this paper, we present how to build a *multi-level debugger* that can reconnect an arbitrary number of source levels and fully inspect the program state at any level. This allows us to provide the required context at any point in the program and thus significantly boost developer productivity. Our solution is built in large parts from *existing* debugging technology, so implementing it for any mature compiler-debugger toolchain is only a *small development effort*. We propose to use a time-travelling debugger to bridge between generated code and generating code and to use unique markers during code generation to reliably perform the connection.

We show that our approach is feasible by implementing it for the Umbra database system [16]. During the development of Umbra's query engine the multi-level debugger setup has proven immensely useful.

2 INTERACTIVE DEBUGGING

Locating the root cause of a failure in a relational query engine works much the same as in any other large code base. A developer first tries to isolate the smallest scenario that exhibits the *erroneous behavior*. Then, they create and refine hypotheses about the cause of the failure and accept or reject them based on observations. This way they trace back from the observably wrong behavior to the root cause. To execute that process efficiently the developer requires debugging tools that can stop the program at a location and observe variables, data-structures, the call-stack, etc.

To show how this process can be applied to a relational query engine and to introduce our proposed tooling, we use a *running example*: Assume that there are two sources of movie ratings, Rotating Tomatoes and the Movie Database. Our example query in Figure 2a counts how many movies receive the same rating in both sources. Also, it only considers movies with more than 10 reviews in the Movie Database. Unfortunately, our example database system returns a wrong result for this query. It returns *count* $= 0$, even though through inspection of the data set we found a movie that fulfills the criteria.

As a first step to find the fault, we check whether the database frontend works correctly. We find that it produces the reasonable execution plan shown in Figure 2b. Therefore, we decide to search for the error in the execution of that plan (as opposed to in the creation of the plan).

In the remainder of this section, we discuss the process and information required for a debugging workflow to find such errors. First, we examine how to debug an execution engine that is built as a Volcano-style interpreter. Here we show how debugging an execution engine should work and which context should be available. Second, we contrast that workflow with debugging an execution engine built with code generation. We show that context information is lost between compile-time and runtime and propose a solution to reconstruct it for debugging purposes.

2.1 How Debugging Should Work: Volcano-style Interpreter

Conventional debuggers are already well suited to debug query execution engines that are built as Volcano-style interpreters. In this section we show how the debugging workflow works and the context information that is available.

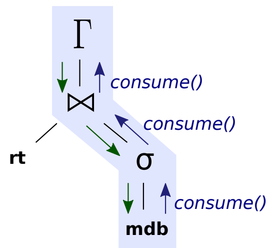
In a Volcano-style interpreter, the execution plan is represented in an object-oriented fashion as *tree of operators* [9]. These operators execute the query plan and are, thus, also well suited for conventional debugging. Figure 3b shows such an operator tree for the example plan. The execution of the plan is coordinated through a small *iterator interface* between operators. Each operator calls `next()` on its child operators to receive the next tuple. When the call returns, the operator performs its own work and passes the tuple on. In this manner, tuples are passed between operators until the query result is computed.

This happens, for example, in the `next()` implementation of the hash join operator in Figure 3a. First, the operator builds a hash table for all the tuples from its left child operator. Second, the operator iterates all the tuples from the right child. For each, it searches matching tuples in the hash table and passes any matches on to the parent operator.

When a developer searches for an error, e.g., for the query plan in Figure 3b, they can *attach a debugger* to the database executable.

```
1 void JoinOperator::consume(ConsumerScope scope)
2 # probe side consume
3
4   hashTable.find(keys, scope, entry -> {
5     ConsumerScope nestedScope(scope)
6     unpack(leftValues, entry, nestedScope)
7     parent.consume(nestedScope)
8   })
```

(a) Code-generator for join hash table lookup.



(b) Control flow with produce-consume in the last pipeline.

```
1  ...
2  block2:
3   %2 = load double %col, %localTid
4   %3 = fptosi i64 %2
5   %4 = sitofp double %3
6   %5 = cmpne double %4, %2
7   condbr %5 %block3 %block1
8  block3:
9   %6 = crc32 i64 5961697176435608501, %3
10  %7 = crc32 i64 2231409791114444147, %3
11  %8 = rotr i64 %7, 32
12  %9 = xor i64 %6, %8
13  %10 = call i64 TextRuntime::hash(%4924, %9)
14  %11 = call ptr HashTable::lookup(%ht, %10)
15  %12 = isnotnull ptr %11
16  condbr %12 %block4 %block1
```

(c) Snippet code generated for the example query.

Figure 4: Code generation with produce-consume fuses all operators of a pipeline into one function.

They can set a breakpoint, e.g. in the hash join operator, so that the debugger stops right in the operator code. By using the debugger's stepping features they can follow a single tuple through multiple operators. At any point when the debugger stops at a breakpoint the developer is able to *inspect variables and data structures* that the operator uses for query processing. This lets the developer check whether the actual query execution still matches their expectation. Furthermore, the debugger allows to unwind the call-stack and thus not only inspect the current operator, but also operators higher up in the query plan. That context helps to understand the current step and allows the developer to decide whether the current program state is still ok or already affected by the error.

Recall that in the example query an error causes *count* to be zero. A good starting point for debugging might be to set a breakpoint in the hash join in Line 5. Once the debugger stops there, we could *inspect* the `hashTable` and check whether it contains any tuples. If so, we can also look at some of those tuples and check if the placed data is ok. In case it is, we could then decide to step into the hash table's `find` function and investigate further. A debugging workflow as just described is already well supported by conventional debuggers, e.g., gdb, lldb, Visual Studio debugger, etc.

2.2 Debugging Code Generating Engines

In contrast to Volcano-style interpreters, compilation-based engines execute a query plan in a two-step process. This results in high query execution speed, but also entails that the previously described debugging workflow—stepping through the operators—is not possible.

2.2.1 Background: Code Generation and Execution. In the first step—called *compile time*—the execution engine *generates code* for the query plan and compiles it to machine code. In Umbra we generate a custom intermediate representation, modelled after LLVM IR, that we call Umbra IR and which we will use in our example. The architecture we use for code generation is the produce-consume method [15]: To generate code for an operator tree the topmost operator calls `produce()` on its child operators. The response from the child operator is that eventually it calls back the operator's `consume()` function and passes an input tuple. Here, the operator has access to the input tuple and generates code to process it. After the code generation, the code is passed to a compiler to produce natively executable machine code.

When again applying this to the example of a hash join we get the implementation of Figure 4a. The `consume()` function gets a scope in which it can find all the values that previous operators produced. It uses these to generate a hash table lookup with the join keys. In this example, the `hashTable` takes care of generating code for hashing the keys, lookup, etc. All matching hash table entries are then passed to the lambda function in Lines 5 to 7. Our implementation then takes the values from the found entry, puts them into a nested scope, and passes them on to the next operator. In contrast to interpretation based engines, we traverse the operator tree during code generation and, instead of directly processing tuples, we produce code to process tuples.

As the second step—called *(query) runtime*—the generated code is *executed* to process tuples and compute the query result. In this step all the effort invested at compile time pays off through high-speed execution of native code.

2.2.2 Missing Debug Context at Query Runtime. Due to this two-part process, the debugging situation in a code generating query engine is very different. We can use a conventional debugger to place breakpoints in the generated code, e.g., in Umbra we can place breakpoints in Umbra IR, *step through the IR program* and inspect the values. Figure 4c shows an excerpt of code that the hash join operator generates for the example query. Here, we could set a breakpoint Line 3 where data is loaded from memory. By stepping to the next line with the debugger, we can then trace the execution and print values, but we can *only guess* which operator generated the instructions and what the values mean.

Generally, this method can work for an *expert developer* who knows the code generator very well and is familiar with what the generated code usually looks like and which patterns usually occur. In that case, stepping through Umbra IR only helps to find the most obvious programming mistakes. However, if the fault is caused by a more complex interaction of operators, the IR quickly becomes a *confusing* place. The experience of debugging Umbra IR that is generated from a query plan is very much similar to stepping through x86-assembly that was generated from C++, but without any debug information to link the assembly to C++ source lines. Furthermore, newcomers to a code generating project lack the experience to read and understand the rather low-level intermediate representation and it can represent a *high entry barrier.*

2.2.3 Reconstructing Context. What this situation calls for is that the developer gets context information about the operator that generated the code and which specific purpose it serves. All that context is available in the code generation phase. Unfortunately, due to the two-step process, the *context information* is at query runtime *no longer available* to a conventional debugger. In this situation we believe that the debugging experience can be greatly improved by providing developers with the necessary context when debugging generated code. Ideally, the same information about context, operators, and variables as when debugging a Volcano-style interpreter should be available.

To make this possible and supply the necessary context at query runtime, we propose to build a *multi-level debugger.* The required components are a time-travelling debugger and unique identifiers that map instructions in the generated code back to the code generation. The time-travelling debugger allows us to record the program execution and to replay code generation as often as necessary. With unique identifiers we can navigate to the generation of specific instructions in the replay. That is, we can stop the replay when, e.g., instruction 5 (Line 6) is appended to the program.

To put it all together, during query runtime we can set breakpoints and step through generated code with a conventional debugger. If at any line of generated code we need to understand which operator generated it and why, we use the time-traveling debugger to replay the recording of the code-generation process up to exactly where the line is generated. This *reconstructs* the exact program state during code generation and we can inspect it with all the usual debugging tools so that we can explore *all the required context.*

2.2.4 Debugging the Example Query. We can use this approach to debug the example query: We set a breakpoint in the generated code (as previously) in Line 3 of Figure 4c. The debugger breaks at that position and we start stepping line by line to reach the bottom part of the snippet where the instructions seem related to a hash table lookup. Unfortunately, execution does not reach to that point. The conditional branch in Line 7 always branches to `%block1` and thus never continues to `%block3`. At this point, we need to decide whether that behavior is ok, but we don't understand why the branch is there and what the comparison in Line 6 should accomplish.

In order to get the missing context information we start an additional debugger session with the time-travelling debugger and replay to the point where Line 6 is generated. By unwinding the call-stack from there, we observe that the join operator is currently performing a hash-table lookup (Line 4 in Figure 4a). Next, we go down in the call-stack and learn that the hash table currently collects the join keys to compute a hash from them. But why are there two floating-point conversions in the code? We inspect the join keys that are just being hashed and learn that they are of type string and double! A short check of the other side of the join reveals that those join keys are of type string and integer.

Going down one more stack frame into the function that collects keys for hashing, we learn that the rating value is casted from double to integer in order to compute the hash for the equality comparison. The cast implementation performs one cast to integer and another cast back to double. Only when the round-trip cast gives the same value as the original double value for rating there can be a join partner from the left side. Otherwise, the double value is outside the domain of integers. From the current position on the call-stack we also learn that Line 6 is generated for the comparison of round-trip casted value to the original value and that the comparison result is stored in a variable named `outsideDomainIndicator`. We can then use the time-travelling debugger session to step forward and follow the uses of `outsideDomainIndicator`. This way, we learn that the hash table uses it to skip the lookup. The current code performs a lookup when the value is outside the domain, however, it should be the other way around. This is easily fixed, e.g., by negating the indicator.

We have seen in this process, that generated code can be rather low-level and piecing together what it should accomplish can be like solving a puzzle. Thus, the ability to connect the generated code back to the code generator is an invaluable tool to debug complicated cases.

3 EVALUATION

In this section we check the following hypotheses:

- Creating a multi-level debugger using time-travelling debuggers is feasible.
- The effort to implement such a solution is low.
- The runtime overhead of the time-travelling debugger is acceptable for database system development.

3.1 Multi-level Debugging for Umbra

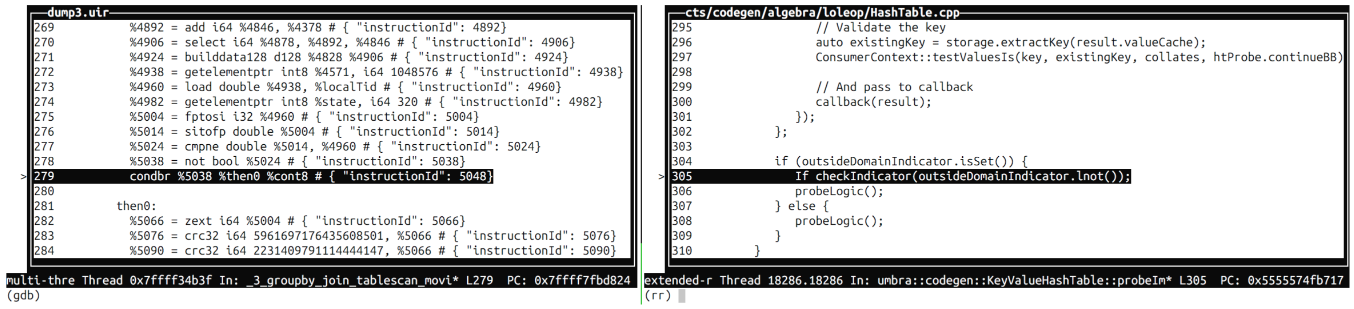
We implemented the proposed solution for Umbra [16], our code-generating database system. Umbra is written in C++ and generates Umbra IR as intermediate representation. We use the LLVM compiler framework to generate optimized machine code from Umbra IR. Our existing infrastructure uses LLVM's debug information mechanisms to attach debug information to the machine code. This already enables us to use a debugger, e.g., the GNU debugger gdb, to stop the program at query runtime, step through the generated code, and print variables from Umbra IR.

To extend this setup for multi-level debugging, we employ Mozilla's RR debugger [17]. RR is a deterministic time-travelling debugger based on gdb. It can record a program execution, in this case how Umbra processes a query, and replays it any number of times exactly as during the recording. During a replay it offers all features of gdb, e.g., breakpoints, printing and stepping. We chose RR because it is readily available and light-weight, but other time-travelling debuggers may also be used for this.

As RR is based on gdb we extended RR through gdb's Python interface. We implemented a `goto-instruction` command. It takes one instruction identifier (from the generated code) as argument and replays execution to the point where the instruction is generated. The command's core is a temporary conditional breakpoint:

```
gdb.execute("tb IRProgram.cpp:972 if ip == " + instructionId)
```

This sets a breakpoint at the source location where instructions are appended to Umbra's intermediate representation. The condition on the breakpoint ensures that the debugger only stops when the requested instruction is generated (otherwise it would stop at every instruction).



```
dump3.uir
269            %4892 = add i64 %4846, %4378 # { "instructionId": 4892}
270            %4906 = select i64 %4878, %4892, %4846 # { "instructionId": 4906}
271            %4924 = builddata128 d128 %4828 %4906 # { "instructionId": 4924}
272            %4938 = getelementptr int8 %4571, i64 1048576 # { "instructionId": 4938}
273            %4960 = load double %4938, %localTid # { "instructionId": 4960}
274            %4982 = getelementptr int8 %state, i64 320 # { "instructionId": 4982}
275            %5004 = fptosi i32 %4960 # { "instructionId": 5004}
276            %5014 = sitofp double %5004 # { "instructionId": 5014}
277            %5024 = cmpne double %5014, %4960 # { "instructionId": 5024}
278            %5038 = not bool %5024 # { "instructionId": 5038}
> 279          condbr %5038 %then0 %cont8 # { "instructionId": 5048}
280
281          then0:
282            %5066 = zext i64 %5004 # { "instructionId": 5066}
283            %5076 = crc32 i64 5961697176435608501, %5066 # { "instructionId": 5076}
284            %5090 = crc32 i64 2231409791114444147, %5066 # { "instructionId": 5090}
multi-thre Thread 0x7ffff34b3f In: _3_groupby_join_tablescan_movi* L279  PC: 0x7ffff7fbd824
(gdb)
```

```
cts/codegen/algebra/loleop/HashTable.cpp
295                  // Validate the key
296                  auto existingKey = storage.extractKey(result.valueCache);
297                  ConsumerContext::testValuesIs(key, existingKey, collates, htProbe.continueBB)
298
299                  // And pass to callback
300                  callback(result);
301               });
302            };
303
304            if (outsideDomainIndicator.isSet()) {
> 305             If checkIndicator(outsideDomainIndicator.lnot());
306               probeLogic();
307            } else {
308               probeLogic();
309            }
310         }
extended-r Thread 18286.18286 In: umbra::codegen::KeyValueHashTable::probeIm* L305  PC: 0x5555574fb717
(rr)
```

Figure 5: GDB on the left, stepping through generated code. RR on the right, providing context from the code generator

With this tooling we can run two debug sessions side-by-side as shown in the screenshot in Figure 5. In the left panel, we use gdb to step through the execution of generated code. In the right panel, we used the `goto-instruction` command to navigate to the source code that generated the code in the left panel. Note that the full context of the generated code is available in the time-travelling session on the right. In the shown example it is possible to unwind the call-stack and reach the implementation of the hash-join operator. It is also possible to go down in the stack to lower abstraction layers of the code generator and observe which instruction is generated. At all positions in the code we can print variables and inspect data structures. Additionally, the debugger offers the ability to step forwards and backwards through the code generation process. This implementation shows that the concept is feasible and in our experience it proved useful for the development of Umbra.

3.2 Implementation Effort

To estimate the effort to build a multi-level debugger, let us note that implementing the core functionality—the `goto-instruction` command—only takes 11 lines of Python code. It already gives users the ability to replay to the generation of a specific instruction and provides all the necessary context.

Additionally, we added a convenience feature that, after replaying to the point where one instruction was generated, unwinds the call-stack to the first operator translator. That code location gives a quick overview of where the translation process currently stands and the developer finds operator objects there to inspect.

We can also control RR from other programs to show context information. An http interface exposes the `goto-instruction` command so that it can be triggered from outside RR. For example it can be controlled from a gdb session that debugs the query runtime or from a text editor where a developer inspects the generated code.

When also accounting for these additional features, the Python plugin to RR has 74 lines of code. After the initial investigation and development of the core ideas for the multi-level debugger, the implementation took less than a week of work. Given this short time and how short the implementation is, we conclude that the overall effort to build such a tool is rather low.

3.3 Runtime Overhead

In the default setup our multi-level debugger uses RR to record the whole process of query execution. It records query parsing, optimization, code generation, compilation, and execution. In order to be able to replay that exact behavior, RR must record also all the data that is loaded from disk, thus write a copy of it into a recording file. Obviously, in the context of a database system this can amount to large volumes of data, which ultimately impacts the recording and replaying speed.

For example running TPC-H query 1 at scale factor 1 with Umbra generates 467 MB of recorded data. The runtime without recording is 1 second, wheres with recording it is 10 seconds, so the introduced overhead is a factor of 10x slow-down. This large overhead may be acceptable for some tricky debugging cases, where the full power of RR's deterministic replay features are actually helpful.

However, we find that if we want to use the multi-level debugger as a fast-paced tool that is quick to provide feedback to developers it is sufficient to only create a recording of the code generation process with RR. Afterward we start a new debugging session with the conventional debugger gdb to step through the execution quickly and use the previous recording to provide context. That approach generates a smaller recording of 254 MB and only takes 2 seconds. In the majority of cases we found the latter technique to be adequate, as the two program runs perform the exact same operations. Thus the RR recording supplies accurate information at a low runtime overhead.

4 RELATED WORK

4.1 Debugging Relational Code Generators

In a comparison of interpreting and compiling query engines we observed that a major difference between the approaches is that interpreters are debuggable with conventional tools [12].

Kohn et al. propose as an approach to debug compiling query engines to collect information about the call-stack at compile time [13]. This information can then be used in a purpose-built debugger. It executes the generated code with a virtual machine, can step through it, and uses the collected call-stack information to provide context from compile time. Consequently, the available context is limited to the collected information. It is not possible to inspect variables and data structures from compile time. In contrast, our approach is more comprehensive, as all context that is available at compile time is also available for debugging. It is even possible to step through compile time code while debugging runtime code. Furthermore, our approach is easier to build as it reuses available tools and when those are improved it is directly reflected in our debugger.

Another approach to debugging compiling query engines is presented by Tahboub et al. with the LB2 system [20]. Although LB2's primary goal is not ease of debugging, it offers an elegant way around the two-phase problem of code generators. LB2 uses extensions to the Scala compiler to instantiate a (somewhat) Volcano-style interpreter and also a code generator for relational queries from the same source base. That means that implementing an operator once yields an interpreter and a compiler. Conveniently, debugging can thus be performed mostly in the interpreter with conventional debuggers. This approach is an excellent idea, however, the reported code generation times of LB2 are on average 300× longer than those we observe in Umbra. Thus, for reasons of practicality, we stay with our C++ and LLVM based approach instead of switching to Scala, and make use of our multi-level debugger.

4.2 Time Travel Debuggers

Recording program execution and replaying the exact execution deterministically was an active research field for at least two decades. The proposed record and replay techniques are powerful, yet the user must consider certain trade-offs between available techniques. Engblom provides a comprehensive overview and classification [7].

One group of techniques works in user-space and replays execution at the machine level, thus they are simple to deploy. PinPlay [18], iDNA [4], UndoDB [1] and TotalView ReplayEngine [8] use binary instrumentation to track data coming in from outside the bounds of recording. RR [17], on the other hand, intercepts system calls to record their effects and traps certain non-deterministic instructions. As this approach does not account for inter-thread data races, RR forces execution to use only a single thread at each point in time, thus slowing execution of highly parallel programs.

Other approaches to record and replay include extending language runtime environments [2], frameworks [5] or libraries [10], OS Kernel support [3, 14, 19], and replayable virtual machines [6, 21]. However, these solutions are too intrusive or heavy-weight for a multi-level debugger.

5 FUTURE WORK

Our current multi-level debugger implementation with RR already serves us well, yet certain aspects can still be improved. Using RR snapshots for the `goto-instruction` command may enable us to jump to instruction creation instead of replaying from the start. Furthermore, to reduce the overhead of RR, especially when handling large databases, it may be interesting to move code generation to a separate process and only record that.

6 CONCLUSION

We showed that debugging compiling query engines with the currently available tools can be a lengthy and involved process. We identified that the main issue of debugging code generators is that at query runtime essential context information from query compile time is missing. This makes debugging a relational code generator a daunting task for newcomers and experts alike.

As a solution, we proposed to build a multi-level debugger that supplies the necessary context. It facilitates a more efficient debugging process and also can also serve as an exploratory tool for beginners. We showed how to build a multi-level debugger from existing technology with low engineering effort and proved its feasibility as Umbra's debugger.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286). We thank Andy Pavlo for the idea of describing our debug tools here.

REFERENCES

[1] [n.d.]. UndoDB. https://undo.io. Accessed: 2020-02-18.

[2] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, and John M. Vlissides. 2001. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. In *IPDPS-01, San Francisco, CA, USA, 2001*. 23.

[3] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010. Deterministic Process Groups in dOS. In *OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 177–191.

[4] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*. 154–163.

[5] Brian Burg, Richard Bailey, A. J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. In *UIST'13, St. Andrews, UK, 2013*. ACM, 473–484.

[6] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *OSDI 2002, Boston, Massachusetts, USA, December 9-11, 2002*, David E. Culler and Peter Druschel (Eds.). USENIX Association.

[7] Jakob Engblom. 2012. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE, 1–6.

[8] Chris Gottbrath. 2008. Reverse debugging with the TotalView debugger. In *Cray User Group Conference*. Citeseer, 5–8.

[9] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. 209–218.

[10] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 193–208.

[11] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[12] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* 11, 13 (2018), 2209–2222.

[13] André Kohn, Viktor Leis, and Thomas Neumann. 2019. Making Compiling Query Engines Practical. *IEEE Trans. Knowl. Data Eng.* (2019).

[14] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *SIGMETRICS 2010, New York, New York, USA, 14-18 June 2010*. ACM, 155–166.

[15] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011).

[16] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[17] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *USENIX ATC 17*. USENIX Association, Santa Clara, CA, 377–389.

[18] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO 2010, Toronto, Ontario, Canada, April 24-28, 2010*. ACM, 2–11.

[19] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. 2004. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *2004 USENIX, June 27 - July 2, 2004, Boston, MA, USA*. USENIX, 29–44.

[20] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD*. 307–322.

[21] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissmann. 2007. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Workshop on Modeling, Benchmarking and Simulation, June, 2007*.