Evolution of a Compiling Query Engine

Thomas Neumann
Technische Universität München
Garching, Germany
neumann@in.tum.de

ABSTRACT

In 2011 we showed how to use dynamic code generation to process queries in a data-centric manner. This execution model can produce compact and efficient code and was successfully used by both our own systems and systems of other groups. As the systems become used in practice, additional techniques were developed for shortcomings that did arrive, including low-latency compilation, multi-threading support, and others. This paper gives an overview of the evolution of our query engine within in the last ten years, and points out which problem have to be tackled to bring a compiling system into production usage.

PVLDB Reference Format:
Thomas Neumann. Evolution of a Compiling Query Engine. PVLDB, 14(12): 3207 - 3210, 2021.
doi:10.14778/3476311.3476410

1 INTRODUCTION

With increasing main memory sizes, it became clear that the performance of a query engine will often or even primarily be dominated by its memory access and its CPU usage [4]. Without I/O, the CPU basically executes a query as fast as it can. This means that to speed up query processing, we either have to reduce memory accesses (which is not always possible) or reduce the number of instructions. The traditional Volcano-style iterator model has a high interpretation overhead, which was fine when systems mainly were waiting for I/O, but which is very noticeable when query processing is CPU bound. Systems like VectorWise reduce that overhead by using a vectorized execution model [5] where each operator invocation produces a large number of tuples. This eliminates the call overhead of the iterator model, but it introduces additional instructions for materializing and loading tuples from vectors.

In 2011 we proposed a radically different approach in the context of our HyPer development [7]: We eliminated the interpretation overhead by using runtime code generation. And we generated the code in a way that memory is touched as rarely as possible [16]. This *data centric* code generation blurs the traditional operator barriers and focuses on the data flow instead. This leads to compact and very efficient code. The benefits are particularly pronounced for OLTP systems, where vector processing is often not beneficial, but also OLAP systems can benefit from the low instruction count. That processing approach was very successful, both in our HyPer system [7], which was acquired by Tableau and now processes millions of queries per day as their core data processing engine, and in other systems like for example [1, 6].

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476410

As our system saw more production usage, the system continued to evolve as we addressed issues that arose. First within HyPer, and then after HyPer became commercial, within our new Umbra engine, where we tested even more radical ideas [18]. In this paper, we give an overview of the necessary steps to bring a compiling query engine into production and point out common pitfalls and useful techniques for implementing such a system. We start with an overview of data-centric code generation in Section 2 and then discuss the problem of low latency compilation in Section 3. Afterwards, in Section 4 we look at the integration of morsel-driven execution into the compilation framework. Developer productivity is discussed in Section 5, followed by further extensions of the compilation model in Section 6.

2 DATA CENTRIC CODE GENERATION

The key idea of data-centric code generation is that we generate code in a way that minimizes memory access. Ideally, we loop over all tuples of a materializing operator (or a table scan), load each individual tuple into CPU registers, process the tuple as needed, and store it into the next materializing operator. We call such a source-operator/intermediate-operator(s)/sink-operator sequence a *pipeline*. This is illustrated in Figure 1, where the dotted boxes show the pipeline boundaries. Note that an operator can be part of multiple pipelines, potentially in different roles. For example, an in-memory hash join is the sink of its build pipeline and an intermediate operator of its probe pipeline.

After identifying the pipelines we generate code using a *produce/consume* paradigm: The source of a pipeline is asked to produce values, and each operator calls the consume function of their parent operators until the tuple reaches the sink. Note that these produce/-consume calls are only compile-time concepts that describe the data flow, in the generated code, neither produce nor consume is visible [16]. The generated (pseudo-)code for the example query is shown in Figure 2. The brackets correspond to the pipeline boundaries, and the code consists of tight loops around materialized data, which is beneficial for modern CPUs.

This core concept works well for a wide range of uses cases, but nevertheless, our implementation evolved over time. For example, in Umbra we no longer generate one function per query but instead generate roughly one function per pipeline and organize the pipelines in a state machine [18] (except for some special cases where we can prove that the number of tuples is low, e.g., for OLTP). The advantage of that is that we can easily suspend and resume a query without blocking a thread, which is useful for scheduling and I/O handling. Using a state machine instead of a simple list allows us to handle complex plans like, e.g., WITH RECURSIVE or clustering algorithms. Also, the explicit *produce* calls were eliminated,

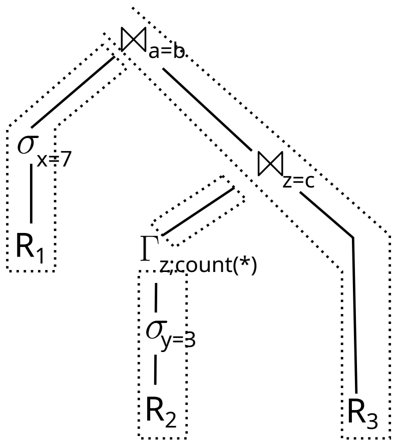

Figure 1: Example plan with visible pipeline boundaries for `select * from r1, (select z, count(*) from r2 where y=3 group by z), r3 where a=b and c=z and x=7` (from [16]).

```
initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and $\Gamma_z$
for each tuple $t$ in $R_1$
    if $t.x = 7$
        materialize $t$ in hash table of $\bowtie_{a=b}$
for each tuple $t$ in $R_2$
    if $t.y = 3$
        aggregate $t$ in hash table of $\Gamma_z$
for each tuple $t$ in $\Gamma_z$
    materialize $t$ in hash table of $\bowtie_{z=c}$
for each tuple $t_3$ in $R_3$
    for each match $t_2$ in $\bowtie_{z=c}[t_3.c]$
        for each match $t_1$ in $\bowtie_{a=b}[t_3.b]$
            output $t_1 \circ t_2 \circ t_3$
```

Figure 2: Compiled query for Figure 1.

instead each pipeline registers which other pipelines have to be evaluated first, potentially with additional reordering constraints. This gives us degrees of freedom in scheduling, for example, to reduce memory pressure or to improve the repeated evaluations of WITH RECURSIVE statements. And value handling becomes more sophisticated over time. At compile time, each pipeline abstraction keeps track of which *IUs* (a generalization of columns and computed values, see [15]) is available where in the pipeline and optimizes its placement. Usually, we want to load a value as late as possible and keep it in registers until we reach the sink. But in some cases, for example, when executing a cascade of outer joins that uses function calls to avoid code explosion, we voluntarily materialize earlier to reduce register pressure. These optimizations are hidden within abstraction layers and are usually not visible when implementing operators, but internally the materialization logic is quite complex.

A great strength of a compiling query engine is that we can use nice high-level abstractions without affecting query execution time. This observation is something that Christoph Koch called "abstraction without regret" [21]. At compile time, we can have composable layers of abstraction that allow for very high-level programming, but in the generated code, all these layers are gone, and get the good performance of straightforward code. Some people think that compiling engines are difficult to build (we will discuss this further in Section 5), but in practice, the implementation of, e.g., a hash join can be more high level and more abstract in a compiling engine than in a traditional query engine, as we do not have to pay the performance price of these extra abstraction layers.

3 LOW LATENCY COMPILATION

While dynamic code generation allows for very efficient execution plans, its key weakness is the compile time of the query. In the very beginning, HyPer experimented with generating C++ code, but that was clearly a terrible idea due to multi-second compilation times. We then switched to using the LLVM compiler backend, which is significantly faster: We could compile TPC-H queries in 40-90ms.

For prepared OLTP queries or long-running OLAP queries, these compilation times were perfectly fine, but nevertheless, they became a problem in practice. As Martin Kersten pointed out in a workshop discussion, end-to-end MonetDB could easily outperform HyPer on small data sets due to the high compilation time. Even for TPC-H SF1 the query compilation time was often higher than the query execution time. We are very grateful for that comment, as it forced us to think more about compilation time early on. This became even more urgent when we saw real-world customer queries, as often customers run complex queries on relatively small data sets. Thus offering low latency compilation is essential in practice when using a compiling engine.

We solved this problem by introducing adaptive compilation [12]: Instead of always compiling queries using LLVM, we introduced our own intermediate representation (IR), which is similar to the LLVM IR but is optimized for compile-time and includes database-specific operations. This allowed us to have different backends for query execution, including a LLVM based backend like we had before, and a virtual machine (VM) based backend to which we could compile very quickly. Then, at query execution time, we first compiled the query to the VM backend, which took very little time, and started executing the query. We then took measurements to predict the remaining execution time. In pipelines where we predicted that the remaining execution time justified the higher compile time of LLVM we compiled the code for that pipeline with LLVM in the background and then switched over to the LLVM generated code. Because all backends conceptually execute exactly the same code, switching is easy and does not lose any progress.

We later refined that approach by introducing an additional backend that directly produced x86_64 machine code [9]. Even though that might sound daunting at first, it is not very different from producing code for our register machine VM when the low-level details of machine code generation are hidden behind an emitter library like asmjit. With that backend, we can generate fast machine code very quickly, and we only rarely ever trigger the LLVM backend, as the still superior quality of the LLVM generated machine code rarely justifies the significantly higher compilation time. We can now compile the TPC-H queries in 1-2 ms, which is competitive even on small data scales, and we can still produce highly optimized machine code if needed.

Our experiences with using LLVM as a code generator have been a bit mixed. LLVM produces excellent, high-quality machine code, but it has the unfortunate tendency to use super-linear time algorithms, for example, when computing the lifetimes of values. This is very problematic for large, generated queries. We have seen real-world queries [22] with 300,000 disjunctions of non-trivial terms, and the compile-time of LLVM tends to explode for such huge code fragments. Our adaptive compiler correctly predicts these super-linear effects, and thus avoids calling LLVM if the compile-time is unreasonable. We need lifetime analysis as basis for register allocation in our backends, too, both for the VM backend and for the asmjit backend, but we made sure to use only linear-time algorithms for analysis [17]. In particular Ramalingam's loop identification algorithm works very well [20], and allows us to quickly compute tight lifetime bounds, which is useful for all kinds of optimizations. As a stress test, we ran a query with 10,000 joins, which Umbra can compile and execute in about 5s. Most of that time is spent in the query optimizer and not in the code generation part. When compiling exactly the same code with the LLVM backend, LLVM does not terminate within two hours.

4 MORSEL-DRIVEN EXECUTION

The core count of current CPUs is quite high, 16 cores are affordable even in the consumer range, and single-socket machines with 64 cores are available for less than 6K$. Therefore query execution will nearly always be multi-threaded, and the query engine must be built accordingly. Both HyPer and Umbra use a *morsel-driven* execution model for parallelization [13], where the work is split into a large number of small tasks that can then be scheduled by the engine. The systems use a 1:1 mapping between cores and threads, which means that we can explicitly assign work to a certain core if desirable for locality reasons. When running out of local work, the threads start stealing morsels from other threads, which helps in balancing out differences in execution speed.

This execution model was introduced early in the development of HyPer, but nevertheless, it evolved over time. The original model was that the query executer registered a job with the scheduler (e.g., execute one pipeline of the current query), and the worker threads would then repeatedly ask the job object for work morsels and execute them until the job was finished. In Umbra, we split that interaction with the job into two methods, one *pickMorsel* would just compute the description of a work morsel, and *executeMorsel* would then process a morsel that was picked earlier. The advantage of that split is that the two functions can be executed on different threads: When starting a new query many or even most threads will be sleeping. Instead of waking them all up and producing contention on the job object, the starting thread can pick morsels for all sleeping threads and then wake them with an explicit starting morsel assigned to them. This greatly reduces contention on systems with high core counts.

On the programming side the compile-time pipeline abstraction allows for convenient and type-safe registration of query-global and thread-local state, which simplifies implementing multi-threaded operators. The mental model there is that, if possible, we treat global state as read-only and only modify our thread-local state, except for well-defined synchronization points where the thread-local state is merged into the global state. This mode avoids any problems with race conditions. There are some exceptions to that rule, for example, outer joins use atomic operations to mark tuples that did find a join partner, but there are so few of them that we rarely had problems with racy behavior.

```
void Select::consume(ConsumerContext& context) const {
   // Check filter condition
   If::build(context.deriveTruth(*op.condition), [&]() {
      // Pass qualifying tuples to consumer
      ConsumerContext innerContext(context);
      innerContext.consume();
   });
}
```

Figure 3: Code generating code for selections.

5 INCREASING PRODUCTIVITY

Compiling database engines have a reputation of being hard to build, but we think that is not really true. The main difficulty is getting the initial infrastructure in place: A nice, type-safe mechanism for generating IR code [9] and an automatic build-time mechanism to expose the data structures of the engine in the IR, including data layout and function signatures. Once we have that, we can generate code in a high-level way that does not look very different from runtime code. The code generation for *select* is shown in Figure 3. It is nearly identical to how a standard select operator would look like. The only subtlety here is that we start a nested context within the if body, as the contexts cache computed values and we are not allowed to reuse values beyond the if that were computed within the if. But that is largely an implementation detail.

While generating code is not very difficult, debugging generated code is more challenging, as there are multiple levels involved: The compile time of the database engine, compile-time of the query, and the execution of the query. Therefore we developed several techniques to help with analyzing that connection. Some of them are simple, like attaching debug information to the generated code that allows for stepping through the generated code while inspecting the corresponding IR code. Or providing a C backend that translates the IR into C code, which has terrible compile times, but is nevertheless useful due to the wide range of debug tools and sanitizers available for C. But by far, the most interesting question when debugging generated code is, who generated that line and why. The "who" question is still relatively simple to answer, in debug builds we use the C++ *source_location* feature to keep track of the originator of an IR instruction and attach that to the IR program, which makes it visible in a debugger. The "why" question is more complex, but can be answered, too: By using a time-traveling debugger, we can switch from debugging the execution of a query to the moment in time that specific instruction was generated [10]. This allows for inspecting the full state of the compiler, which significantly simplifies finding the root cause of a specific code fragment. Similar layer problems arise when profiling query execution. Sometimes we want to see the performance counters on a per-instruction basis, sometimes on a per-operator basis, and sometimes per pipeline. By automatically keeping track in which context an instruction was generated, we can maintain mappings between the different layers [3], which allows for performance analysis on any of the layers we are interested in.

6 FURTHER CONCERNS

When building a compiling query engine, there are of course, many other design aspects that have to be taken into account. For example, it is attractive to decompose relational operators into more low-level operations [11], as this allows for much greater flexibility and easier code reuse. Other interesting questions are how tuples should be processed: Usually, we want to avoid unnecessary tuple materialization. But sometimes, we can hide cache misses better by buffering a small number of tuples, which leads to the idea of relaxed operator fusion [14]. Bringing higher-level logic into the query engine is attractive, too. Systems like Weld allow to combine application logic and query processing into an execution plan [19], which is a very good idea. Just like stored procedures help to improve the performance of OLTP systems greatly, we expect that in the future, integrating parts of the application query processing will significantly help with complex analytical tasks [2].

7 CONCLUSION

Using runtime code generation is attractive, as it allows for generating very compact and efficient code for any given query. At least conceptually, a compiling engine is superior to any other approach, as it can easily emulate whatever the other system is doing, minus the runtime overhead of interpreting data type, as our comparative analysis paper [8] showed. Therefore it is not surprising that code generation has been employed by many systems, such as Spark or Neo4j. The price we pay for that flexibility is increased system complexity. But proper abstraction layers, decomposing high-level operations into low-level steps that can be optimized independently, and proper tooling support make building a compiling query engine tractable. The integration of application logic into the compiled query is an interesting problem that we want to look at in the future.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44. https://doi.org/10.1145/3129246

[2] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *Proc. VLDB Endow.* 14, 11 (2021).

[3] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. Profiling dataflow systems on multiple abstraction levels. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 474–489. https://doi.org/10.1145/3447786.3456254

[4] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 54–65. http://www.vldb.org/conf/1999/P5.pdf

[5] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings.* www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[6] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[7] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[8] 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. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[9] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal* (02 Jun 2021). https://doi.org/10.1007/s00778-020-00643-4

[10] Timo Kersten and Thomas Neumann. 2020. On another level: how to debug compiling query engines. In *Proceedings of the 8th International Workshop on Testing Database Systems, DBTest@SIGMOD 2020, Portland, Oregon, June 19, 2020*, Pinar Tözün and Alexander Böhm (Eds.). ACM, 2:1–2:6. https://doi.org/10.1145/3395032.3395321

[11] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1001–1013. https://doi.org/10.1145/3448016.3457288

[12] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Making Compiling Query Engines Practical. *IEEE Trans. Knowl. Data Eng.* 33, 2 (2021), 597–612. https://doi.org/10.1109/TKDE.2019.2905235

[13] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[14] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.* 11, 1 (2017), 1–13. https://doi.org/10.14778/3151113.3151114

[15] Guido Moerkotte. 2020. *Building Query Compilers.* Retrieved July 26, 2021 from http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

[16] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[17] Thomas Neumann. 2020. *Linear Time Liveness Analysis.* Retrieved July 26, 2021 from https://databasearchitects.blogspot.com/2020/04/linear-time-liveness-analysis.html

[18] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[19] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[20] G. Ramalingam. 1999. Identifying Loops in Almost Linear Time. *ACM Trans. Program. Lang. Syst.* 21, 2 (1999), 175–188. https://doi.org/10.1145/316686.316687

[21] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building Efficient Query Engines in a High-Level Language. *ACM Trans. Database Syst.* 43, 1 (2018), 4:1–4:45. https://doi.org/10.1145/3183653

[22] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, Alexander Böhm and Tilmann Rabl (Eds.). ACM, 1:1–1:6. https://doi.org/10.1145/3209950.3209952