When Vectorwise Meets Hyper, Pipeline Breakers Become the Moderator

Bala Gurumurthy Imad Hajjar David Broneske
Thilo Pionteck Gunter Saake
Otto von Guericke University
Magdeburg, Germany
firstname.lastname@ovgu.de

ABSTRACT

In today's database systems, there is an ongoing debate on which processing model is the best. While the vectorized processing model shows better performance for memory-bound operations and queries, the compiled processing model is beneficial for compute-intensive ones. However, a huge drawback of compiled query engines is their initial query compilation time, which especially sets them back in case of short query runtime, because compilation times overshadow the short query execution times. Hence, the first approaches aim to avoid compilation time by interpreting the code that is to be compiled. However, naturally, interpreted operators optimized for compiled execution cannot compete with pure vectorized operators that were designed for an interpreted execution. Hence, the intuitive approach to hide compilation time is to start query execution in a vectorized engine, and upon finished query compilation, switch to the compiled engine. In this paper, we investigate how to switch from a vectorized to a compiled execution engine by using different natural transition points (i.e., pipeline breakers) in query execution. Our investigations show that this procedure effectively hides compilation times and gives a speedup of 3x when comparing separate query runtimes of compiled execution.

1. INTRODUCTION

Compiled execution engines (e.g., Hyper) have an execution time close to hand-written code [10]. However, its faster execution time comes up with the overhead of compiling the given query. Even though the compilation cost is negligible compared to the query processing time on large datasets, it could be an overhead for small datasizes, as shown in Figure 1. Therefore, it is imperative to address the compilation overhead to improve query processing performance.

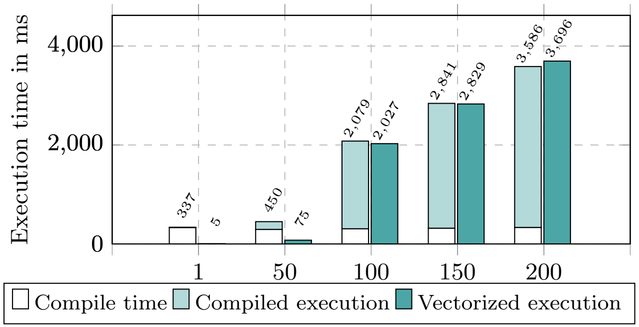


Figure 1: Single threaded performance graph for TPC-H Q6 in vectorized and compiled code execution.

Even with efficient compilation techniques, there is still an idle window between the time when a query is issued until the start of its execution. Depending on the complexity of a query, this window can be as high as 100ms [10]. Therefore to overcome the cost of compilation completely, it is advisable to use interpreted execution that hides compilation of the query [6]. We follow up on this idea and hide compilation using an interpretation based execution engine. A vectorized engine with its execution processing on cache-resident vectors is a natural candidate for efficient interpreted execution. Therefore, in this work, we focus on hiding the compilation overhead using vectorized execution.

Hiding compilation cost using vectorized engine (or rather its operators) requires partial results of vectorized execution to be forwarded to the compiled execution. Since compiled execution does not read/write partial results of intermediate operators in a pipeline, we propose to use their natural materialization points i.e., *pipeline breakers* as exchange points from vectorized to compiled execution.

In this work, we use the pipeline breakers: aggregation, hash join, and hash aggregation (i.e., group by) for forwarding results from vectorized to compiled execution. Based on the framework by Kersten et al. [5], we present how to adapt the two query engines in order to inter-operate in query execution. Specifically, we show how compiled execution uses the intermediate results of the vectorized execution engine.

Our approach shows a performance improvement of up to a factor of three by hiding compilation times and several further interesting details in the result.

In summary, we show that with a pipeline breaker as a connector, we can hide the compilation cost using vectorized execution. Our contributions are as follows:

11th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'20), August 31, 2020, Tokyo, Japan.

- We propose *Tether*, a hybrid execution engine that allows for vectorized data processing while query code is compiled.

- In Tether, we investigate different pipeline breakers to forward partial results from vectorized execution to compiled execution after code compilation is finished.

- Our evaluation analyzes different TPC-H queries and shows significant benefits of hybrid execution that go far beyond the performance gains of purely compiled or interpreted (vectorized) execution.

The remainder is structured as follows. First, we review related work in Section 2. We provide the background information of the two state-of-the-art execution models in Section 3. The features of our Tether execution model are explained in Section 4. We also present in this section, details of using pipeline breakers as switching points between the execution models. We conduct several experiments comparing our Tether framework to stand-alone compiled and vectorized engines using the standard TPC-H benchmark in Section 5. Based on the experimental results, we discuss the key performance factors and the next steps for our work in Section 6. Finally, we draw our conclusions in Section 7.

2. RELATED WORK

To improve on LLVM compilation overhead, Kohn et al. [6] interpret the LLVM IR of a query directly using a custom interpreter. Though similar to us, the work also blends interpreted and compiled execution, but still has to generate the LLVM IR in-order to execute a query (which takes about 0.7 ms from their measurements). Whereas in our work, we use the query plan and directly execute a query using the pre-compiled operators. Furthermore, the LLVM IR interpreter is fine-grained containing hundreds of custom written operators for efficient processing. Though this is efficient to switch between the execution modes, they show that the bytecode interpreter is three times slower than the un-optimized machine code. Our work mitigates this overhead by using coarse-grained operator implementation.

Another work on improving performance using materialization and vectorization in compiled execution is explored by Menon et al. [8]. Their relaxed operator fusion model improves performance by introducing staging points to materialize the results of intermediate operators in a compiled code engine. Our work can benefit from these staging points to decide on a more granular level to switch between the execution paradigms.

Due to execution time being comparable with hand-written code, compiled execution is widely adapted, our hybrid system can complement these works by improving their response time [9, 11, 7].

Kersten et al. [5] have built the vectorized and compiled query processing engines using compatible implementations for a fair comparison of these two processing models. Their implementation serves as the base for our work.

Various hybrid models combining pull and push-based models are available. These models can also include our technique to hide their compilation latency [2, 12]. Other than traditional DBMSs, Spark SQL contains the catalyst optimizer which supports code generation during runtime [1]. Our work can provide improved execution time for this system.

3. PRELIMINARIES ON IN-MEMORY QUERY EXECUTION MODELS

Query engines are the chassis of a DBMS. All other components revolve around how a query is being processed. The current DBMS engines can be split into either pull-based or push-based engines. The more traditional pull-based engine or the volcano model provides high portability of operations for the cost of materializing all intermediate data [4]. This penalty of materialization was later improved by the vectorized processing engine that uses cache resident data to improve performance [14]. A push-based or code-generation engine avoids such unnecessary materialization of values by keeping the data within registers as long as possible. However, this advantage comes at the cost of compilation time [9]. In this section, we detail these two execution models.

3.1 Vectorized Execution

Vectorized processing follows batched execution for query processing. It is an interpreted execution engine where each operator consumes a vector of input values for processing. The size of the vector thereby depends on the cache size of the CPU. Though the operators work on cache resident data, when it comes to operations like group-by, aggregation, or hash-join, intermediate data are materialized into the memory before executing the next operation in the pipeline. This forceful materialization of partial results leads to poor performance. Apart from its materialization overhead, vectorized processing also suffers from overhead of function calls. For example, a conjunctive selection repeatedly executes selections on each of the columns and combines the results either using a relaxed selection or a conjunction operation [5].

3.2 Compiled Execution

To improve data as well as code locality, a compiled query engine generates code directly for a given SQL query [9]. In the generated code, database operators are fused into one pipeline, improving execution time [3]. To facilitate code generation in this model, producer and consumer functions are included in these operators. Since we are compiling operators together before the start the execution, we can have arbitrary combinations of input (e.g., a conjunctive selection can be custom-built for the selection criteria based on the input query).

Though the compiled query execution model provides improved data and code locality along with faster execution time, it suffers from the overhead of compiling an SQL query before execution [10]. This compilation time can vary depending on the underlying compiler. Therefore, a poor compiler might lead to a compilation time that is higher than pure execution time of certain queries.

4. HIDING COMPILATION LATENCY USING VECTORIZED EXECUTION

From the introduction to query compilation, we know that compilation time has an important impact on performance. To hide the compilation overhead, we propose to start query execution concurrently with query compilation using an interpreted query engine. Vectorized execution suits this scenario well with its memory-friendly interpreted processing engine. Furthermore, vectorized engines are robust, well-maintained, and well-established engines for several years now [13]. Hence, we propose a hybrid engine with both execution models.

In the following, we first present the overall workflow of our hybrid query engine –*Tether*– and afterward, we introduce how to switch query execution given the pipeline breakers aggregation, hash join, and hash aggregation.

4.1 Tether: A Hybrid Query Execution Engine

The main task for Tether as the connecting engine between compiled and interpreted (i.e., vectorized) processing is to take an input query and transform it into a hybrid query that is started on the vectorized engine and finalized on the compiled engine. For a more detailed description, we use an exemplary query shown in Figure 2. The example query has: 2 selections, 1 simple aggregation, 1 group-by aggregation, and 2 equi-joins, which leads to five possible pipelines labeled P_1 to P_5. Given this query, a question arises: how to do a minimal-invasive switching between both execution models as they execute queries differently (cf. Section 3). This makes switching hard since a common way of handshaking is needed. In fact, there are two important considerations: we have to decide *how* and *when* to share intermediate results between both execution models. In the following, we answer the *how* by using pipeline breakers and the *when* as after finalizing the current processed vector.

Pipeline Breakers as Switching Points

A vectorized engine materializes intermediate results after each operator. However, these intermediate results cannot be simply shared with a compiled query at an arbitrary point in the query pipeline. This is because compiled execution, as a push-based model, does not consume/materialize intermediate data explicitly from every single operator.

One exception to the *no-materialization rule* of compiled execution is a pipeline breaker [10]. Pipeline breakers force compiled execution to materialize their intermediate values into memory (marking the end of the pipelines P_1 to P_4) before executing the next operator of the pipeline. Hence, the key idea of our hybrid processing model is to use these natural materialization points of a query for switching between vectorized and compiled query execution. From the TPC-H benchmark, we have identified three pipeline breakers commonly present in a query that are useful in connecting vectorized execution with compiled execution: aggregation, hash aggregation (i.e., group-by) and hash join. Hence, by implementing a common data structure for these operations, we can easily switch execution from vectorized to compiled execution.

Inter-Pipeline Switching

With pipeline breakers at both ends of a pipeline, our hybrid processing engine processes at least one pipeline of a query *partially* using vectorized execution. In our example query from Figure 2, this is P_1, which materializes results in a hash table for each vector. However, depending on the compilation overhead and the input size, more than one pipeline might be processed by vectorized execution. Hence, we have to compile the pipelines such that their processing can start at any pipeline breaker.

When vectorized execution hits a pipeline breaker and the compilation is done, a switch can happen. The intermediate results processed by the vectorized execution are then materialized and forwarded to the compilation execution. This provides minimal waiting time for compiled execution after the compilation is finished.

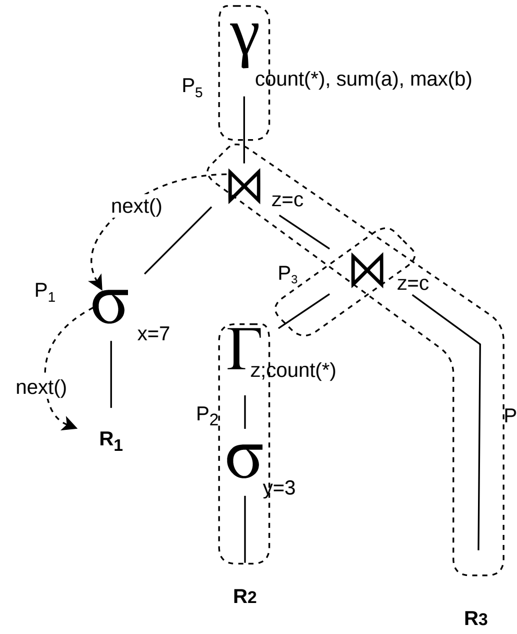


Figure 2: Exemplary hybrid query execution plan in Tether.

Tether's Workflow

The general workflow of our hybrid execution system Tether consists of three phases as given in Algorithm 1. First, we start by compiling the pipelines for compiled execution (`CmplPipeline`) using a compiler thread. The thread adds the instructions for merging the partial results of vectorized execution to the corresponding pipeline before compiling them. Once compiled, the thread sets the `CompilationDone` flag as TRUE.

Second, while compiling, we execute the vectorized operations in the pipeline P_1, which processes values until the following pipeline breaker. In the running example, we materialize the hash table built for the join operation ($z = c$). Additionally, the current index of the input scanned from relation R_1 is also remembered (`ProcessedData`) so that the compiled execution can continue building the partial hash table of vectorized execution.

Finally, once the `CompilationDone` flag is set, we interrupt the vectorized execution, forward the pointers of the hash table and relation R_1 plus the index of already-processed tuples to compiled execution. Afterward, the compiled execution finishes the materialization of the pipeline breaker values based on the current shared index and continues with the next pipelines (P_2 to P_5) producing the final result.

In the following sections, we detail how each pipeline breaker is realized in the two paradigms and how they are executed in our hybrid execution engine Tether. We base the operator implementations of the three pipeline breakers on those from the framework of Kersten et al. [5], as they have proven to be a sophisticated baseline for both models.

Algorithm 1: Tether execution flow.

```
Data: CmplPipeline,VecPipeline
Result: result
bool CmplDone = FALSE;
func* CmplFunc =
  CmplThread.spawn(CmplPipeline);
VecPipeline→PipelineBreaker.open();
ProcessedData = 0;
do
    ProcessedData += VecPipeline.start.read();
    VecPipeline→PipelineBreaker.materialize();
while (!EndOfStream & !CmplDone);
CmplFunc(VecPipeline→PipelineBreaker.data,
  ProcessedData);
```

4.2 Pipeline Breaker: Direct Aggregation

Direct aggregation is the least challenging of all the pipeline breakers we have considered. The aggregate results are computed after a single pass over the input columns. Hence, while switching, only partial results might have been processed by vectorized execution. Hence, we have to forward these partial aggregates from vectorized execution along with the last processed index to compiled execution for continuing aggregation. Notably, a query can have more than one independent aggregation to be computed, resulting in multiple partial results being forwarded from vectorized to compiled execution. Such a query with multiple independent aggregates (e.g., TPC-H Q1 without group by clauses) is computed differently by vectorized and compiled execution.

Compiled Aggregation

Compiled execution generates a custom aggregation function to aggregate all column inputs simultaneously. Therefore at any given instant, as depicted in Figure 3, all tuple values of the respective columns (`a` & `b` in the figure) are read and aggregated one after another. At any point in time, we obtain the partial results of all the independent aggregates (`count(*),sum(a),max(b)`). For example, the aggregates in pipeline - P_5 of Figure 2 will be computed together.

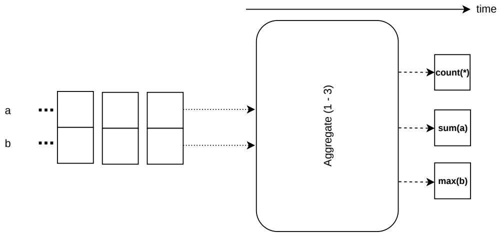


Figure 3: Compiled aggregation.

Vectorized Aggregation

In contrast to compiled execution, vectorized execution aggregates one vector at a time. Therefore, for our running example, partial aggregates will be produced for each processed vector creating the currently aggregated `count(*)`, the aggregated `sum(a)` as well as the aggregated `max(b)` as shown in Figure 4.

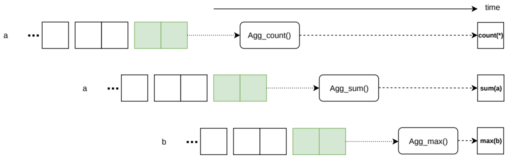


Figure 4: Vectorized aggregation.

Hybrid Aggregation

Connecting the execution paradigms using aggregation requires that the partial results from vectorized execution can be forwarded to compiled execution. Since compilation can be ready at any point during vectorized execution, predicting the exact number of independent aggregates computed by vectorized execution is a complex task. Instead, our idea is to simply update these partial results in compiled execution. To this end, we add the information about the current column and its last visited index along with the partial aggregates. Figure 5 shows the partial aggregates (`p_count`, `p_sum`, `p_max`) of vectors (in green) computed. These partial aggregates are the input for the compiled execution engine, which updates them with the aggregates of the remaining input in a tuple-at-a-time fashion.

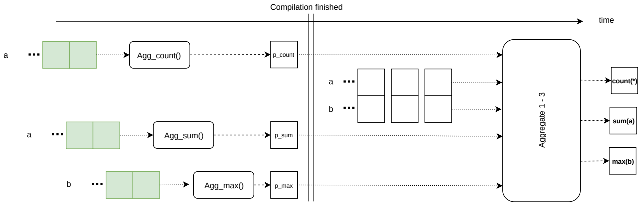


Figure 5: Hybrid aggregation.

4.3 Pipeline Breaker: Hash Join

A hash join breaks a pipeline by materializing input values into a hash table. Once the hash table is built, it is probed by the next pipeline for join pairs. In our hybrid engine, the hash join becomes the connector between the execution engines. To make the hash join compatible across these paradigms, we keep the same hash table implementation as well as the same hash function. This is mainly to have a common set of methods to insert and probe a key within the hash table. Since the execution switches from vectorized to compiled execution, the hash function that favors efficient processing in compiled execution is chosen. Such a suitable hash table and hash function combination are shown to be chained hashing with the *CRC32* hash function [5]. Even though we have common techniques across the execution engines, we still have variations in the way the runtime populates the hash tables in order to have the best performance.

In the following, we explain the two possible states that exist when switching from one engine to the other. In the first possible state – i.e., a hybrid hash build, the build phase is only partially done by vectorized execution when compilation finishes. Compiled execution follows up by building a separate hash table for the remaining tuples. The separation of hash built between the engines is to avoid any unnecessary penalties of populating a vectorized hash table in compiled execution. The main issue here is that the engines populate the hash tables differently. On the one hand,

```
if (ht.contains(o_orderkey[i])
&& (name = ht1.findOne(o_custkey[i])))
{
  entries.emplace_back( ... );
}
```

Listing 1: Compiled hash probe.

```
/* hybrid Code */
if(ht.contains(o_orderkey[i]) {

// Vectorized probe
runtime::CRC32Hash h1;
uint64_t output = h1(o_custkey[i]);
for (auto entry = vwHT.find_chain(output);
  entry!=runtime::Hashmap::end();
  entry = entry->next) {
    if (entry->o_custkey == o_custkey[i]) {
      entries.emplace_back( ... );
    }
}

// Compiled probe
if(ht0.contains(o_custkey[i])) {
    entries.emplace_back( ... );
}
}
```

Listing 2: Hybrid hash probe.

threads in hyper cooperatively populate a single hash table. On the other hand, threads in the vectorized engine populate their private hash table first followed by a global merge step. In the second possible state – i.e., a hybrid hash probe, vectorized execution has already finished building the hash table and now the compiled execution probes the hash table.

Hybrid Hash Build

During switching, the vectorized execution might have only built a partial hash table. Inserting into vectorized hash tables from compiled execution is not efficient. In this case, compiled execution first reads the remaining tuples aggregating them in a separate hash table.

From our example query in Figure 2, considering the switching point is at the join $R1.z = R3.c$, *R1.z* column values are present in two different hash tables. Once built, both hash tables (the vectorized and compiled hash table) will be probed (on *R3.c* from our example) for each tuple. Therefore, compiled execution must include code to probe the hash table of vectorized execution to find join pairs.

Hybrid Hash Probe

Vectorized execution, due to its working granularity of a whole vector, requires a sequence of steps for hash probing. First, the hash values are computed for input vectors. Next using the hash values, the target match locations for vectors are identified. Finally, the values in the target locations are compared to identify join pairs. We circumvent these steps in compiled execution by directly computing the hash value of a given key and probing through the table using the traversal functions of chained hashing.

The inclusion of the additional probe steps of the vectorized hash table increases the lines of code compared to the naive code given in Listing 1. In Listing 2, we depict all hybrid probing steps. For every input key, we first look into the partial hash table of the vectorized engine followed by probing the remaining values in the hash table of the compiled engine.

4.4 Pipeline Breaker: Hash Aggregation

The final pipeline breaker that we consider in our work is a hash aggregation (`z; count(*)` in the running the example). Like the hash join, a hash aggregation does two passes over the input data for computing the results.

In the first pass, the input is hashed and values are grouped in the hash table. In the second pass, the aggregates for each of the groups are calculated. Furthermore, with multi-threaded execution, each thread has to do these two passes plus an additional merge stage, which is necessary to aggregate the partial results from all threads.

Hybrid Hash Aggregation

To connect vectorized with compiled execution, we have to forward the partitioned group values from vectorized to compiled execution to aggregate them. Since hash aggregation follows a similar execution of hash join, the compiled execution also starts with building its own hash table for the remaining input values along with aggregating them. Finally, once the results for compiled execution are ready, we update them with the partial results from vectorized execution. Additionally, in the case of multi-threaded execution, compiled execution also takes care of merging the partial results from all the individual threads of vectorized execution.

5. EXPERIMENTS

In this section, we compare the performance of our hybrid system Tether with a stand-alone compiled and vectorized execution for different data sizes and present our observations. To this end, after a short introduction of the experimental setup, we first discuss the incurred compilation overhead due to our added switching points (e.g., from Listing 2). Afterward, we investigate the benefits and drawbacks of Tether on simple and complex queries with several pipeline breakers.

5.1 Experimental Setup

We conduct our experiments on an Intel® Xeon® Gold 6130 CPU. The machine runs an Ubuntu 18.04. with CLANG version 6.0. For our execution, we parallelized the queries using 16 threads.

Comparable Implementations. As mentioned earlier, the compatibility of data structures and operator implementations between compiled and vectorized execution is a key factor for our system. To this end, we use the operator implementations of *Tectorwise* and *Typer* with their common data structures[1]. Our Tether as a hybrid engine uses the vectorized operator implementations of Tectorwise and the compiled pipeline implementations of Typer and adds custom code for the switching points. The stand-alone engine of Typer does not contain direct code generation or compilation. Hence, we also directly compile the LLVM code of

[1] https://github.com/TimoKersten/db-engine-paradigms

a target query and link them with our execution in runtime and record the time as compilation time. To this end, we use clang 6.0 for compilation (with -O3 flag) and for building the machine code. Similar to Hyper, we consider only the time taken for compiling the code for our experiments.

Workload Description. We use the TPC-H benchmark with scale factors ranging between 100-200 and its queries Q1, Q3, Q6, and Q18 for our experiments. These queries in specific contain the different pipeline breakers that we discussed earlier in the exemplary query plan. we use the query plan of compiled execution for our execution [2]. For the comparison, we measure the execution time for our system and compare it to the runtimes of vectorized and compiled executions. Please note, we *always* include compilation times into the reported execution times of compiled execution and our hybrid system.

Experiments. From the selected TPC-H queries, we derive three important experiments that show the benefits and drawbacks of our hybrid engine Tether. In the first experiment, we are interested in the additional compilation overhead due to our compiled switching points that add extra code to the compiled engine. In the second experiment, we investigate simple queries like Q1 and Q6 that contain a single pipeline and compare Tether's performance to the performance of the other two standard execution engines. Since these two queries have rather small and short pipelines, switching between both models should add considerable overhead, we are interested whether we can still benefit from hybrid execution. In the third experiment, we look at queries with several pipeline breakers, i.e., Q3 and Q18, and investigate good switching points for Tether in order to outperform single-engine performance.

5.2 Experiment 1: Hybrid Compilation Overhead

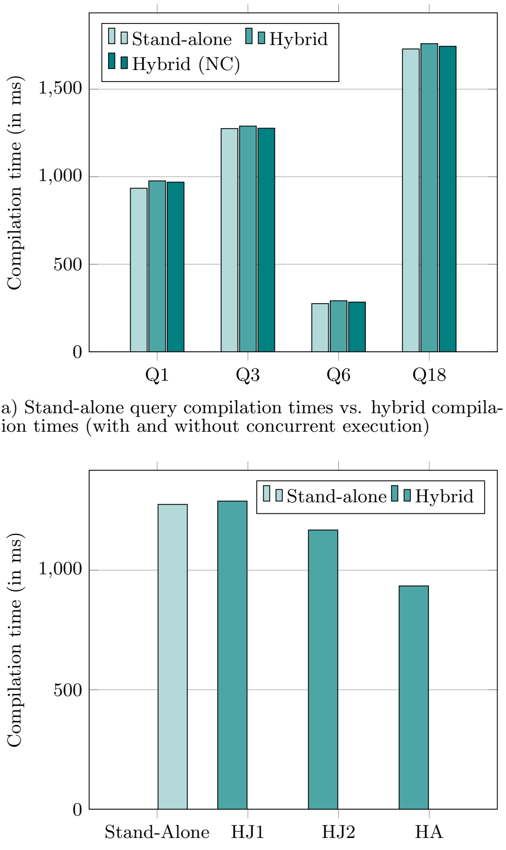
Since Tether includes additional compiled code to merge the partial results of its vectorized engine into its compiled engine, we first investigate the resulting overhead for compiling the queries at the first pipeline breaker. Subsequently, we investigate how the compilation time changes when compiling the query more cleverly at later pipeline breakers.

Compilation Time Comparison

The TPC-H queries we presented above have different pipeline breakers in their initial pipelines: Q1 contains hash aggregate, Q6 has a direct aggregation and Q3 & Q18 are using hash joins. The comparison of compilation times for naive (i.e., Typer), hybrid (i.e., Tether's compilation time when the vectorized engine runs concurrently) and hyrid without any concurrent execution for these queries are depicted in Figure 6. We use a single thread for compilation and use clang for compiling the C++ pipeline code[3]. We see that concurrent processing has a minor impact in the compilation time, and this is mainly due to the overhead of handling the compilation thread. We can also see that the overhead of compiling code with additional merge instructions is between 14ms to 42ms. The worst compilation time is recorded for Q1 or in other terms for merging results of the hash aggregation. This is mainly due to the additional aggregation step required to merge the partial aggregates of vectorized execution with the aggregates of compiled execution. Furthermore, due to parallel execution, we have multiple partial results from vectorized threads which have to be merged with the results of compiled threads.

On the other hand, Q6 (i.e., direct aggregation) leads to the smallest compilation overhead. In this case, we simply perform one additional aggregation step to merge the results of compiled execution with vectorized execution. The merge of the remaining queries (Q3 and Q18) simply includes the probe instructions of vectorized execution along with probing of compiled execution, which incurs only a small overhead for compilation.

Although we have additional costs for compilation, we will in the following hide the compilation totally with vectorized execution. Furthermore, depending on the data size and pipelines in a given query, vectorized execution might process more than one pipeline before we finalize compilation. Therefore with such workloads, we refrain from compiling the pipelines that are completely processed by vectorized execution. This reduces the overall execution time from the naive compilation of a complete query.



(a) Stand-alone query compilation times vs. hybrid compilation times (with and without concurrent execution)

(b) Compilation time for different merge points in Q3

Figure 6: Stand-alone vs. hybrid compilation time.

[2] Plans provided by hyper-DB interface: https://hyper-db.de/interface.html

[3] Our experiments have shown that, of course, different compilers will lead to different compilation times. However, the ratio of compilation times between the stand-alone and hybrid engine is always the same.

Compilation Time Comparison of Pipelines With Partial Merge Instructions

To understand the impact of compiling only partial pipelines of a query, let us consider the pipelines of Q3. In total there are three pipelines and pipeline breakers in the query: two hash joins (HJ1, HJ2) followed by a hash aggregation (HA) before producing the results. By using these different pipeline breakers as the switching points, we incur different compilation times. The compilation time w.r.t. the different pipelines for Q3 is given in Figure 6. By compiling partial pipelines, we outperform the naive compilation significantly leading to better exploitation of compiled query performance. However, this in turn means that we have to delegate a complete pipeline to vectorized execution. Hence, the question that we want to answer in Experiment 3 is: which of these pipelines provide the best trade-off between compilation time and vectorized execution.

5.3 Experiment 2: Single-Pipeline Queries

In this experiment, we compare the execution times of our hybrid system with the stand-alone compiled and vectorized execution engine for single-pipeline queries. We chose queries Q1 and Q6 because they have only a single pipeline to process. Therefore, we have only one possible merging point, i.e., we merge the partial results of vectorized execution into the partial ones from compiled execution for the complete query. For these queries, a high compilation time would lead to the circumstance that vectorized execution processes the input data completely before we could switch the processing engines. Such characteristics are visible in the results for the queries in Figure 7. In Figure 7(a) & (c), we compare the runtime of the three different engines for different scale factors of the TPC-H benchmark. Furthermore, we depict the compilation time for all queries, which is rather stable across different scale factors. In Figure 7(b) & (d), we break the execution time of Tether down to show the ratio of processed tuples in the vectorized engine compared to the remaining tuples that are processed with the compiled engine of Tether.

Due to the compilation time, we see that the switching point for the queries in Figure 7(a) & (c) are around scale factor 140 and 160, respectively. As expected, our system follows the performance of vectorized execution until the switching point. After the switching point, it is clearly visible that our hybrid system deviates from vectorized execution due to an increased fraction of tuples that are processed in the better performing compiled engine. For Q1, the change in performance is slightly affected by the additional merge step. However, this additional impact is negligible considering the data shared among the two systems. We see from Figure 7(b) that even with processing only 30% of data in the compiled engine, we already outperform both engines. A similar case can be also seen for Q6 (cf., Figure 7(c)). Since, it is a comparatively simple query, compiled execution processes only up to 20% for the chosen scale factors of input data at the moment. However, with increasing the scale factor, the amount of data processed by compiled execution will also increase and, hence, improve Tether's performance. Since only a fraction of total input is processed by compiled execution after the cut-off point, our execution time improves accordingly. Overall, our Tether approach is 2.3 times faster than stand-alone compiled execution and 1.2 times faster than vectorized execution after the cut-off point. Now that we saw the performance of single pipelines, we experiment on the impact of several switching points in a query in the next section.

5.4 Experiment 3: Informed Switching Points for Multi-Pipeline Queries

Queries with multiple pipelines require a closer look for selecting the right switching point. To better analyze the importance of the switching points, we show the performance difference of Q3 executed with all possible switching points in Figure 8.

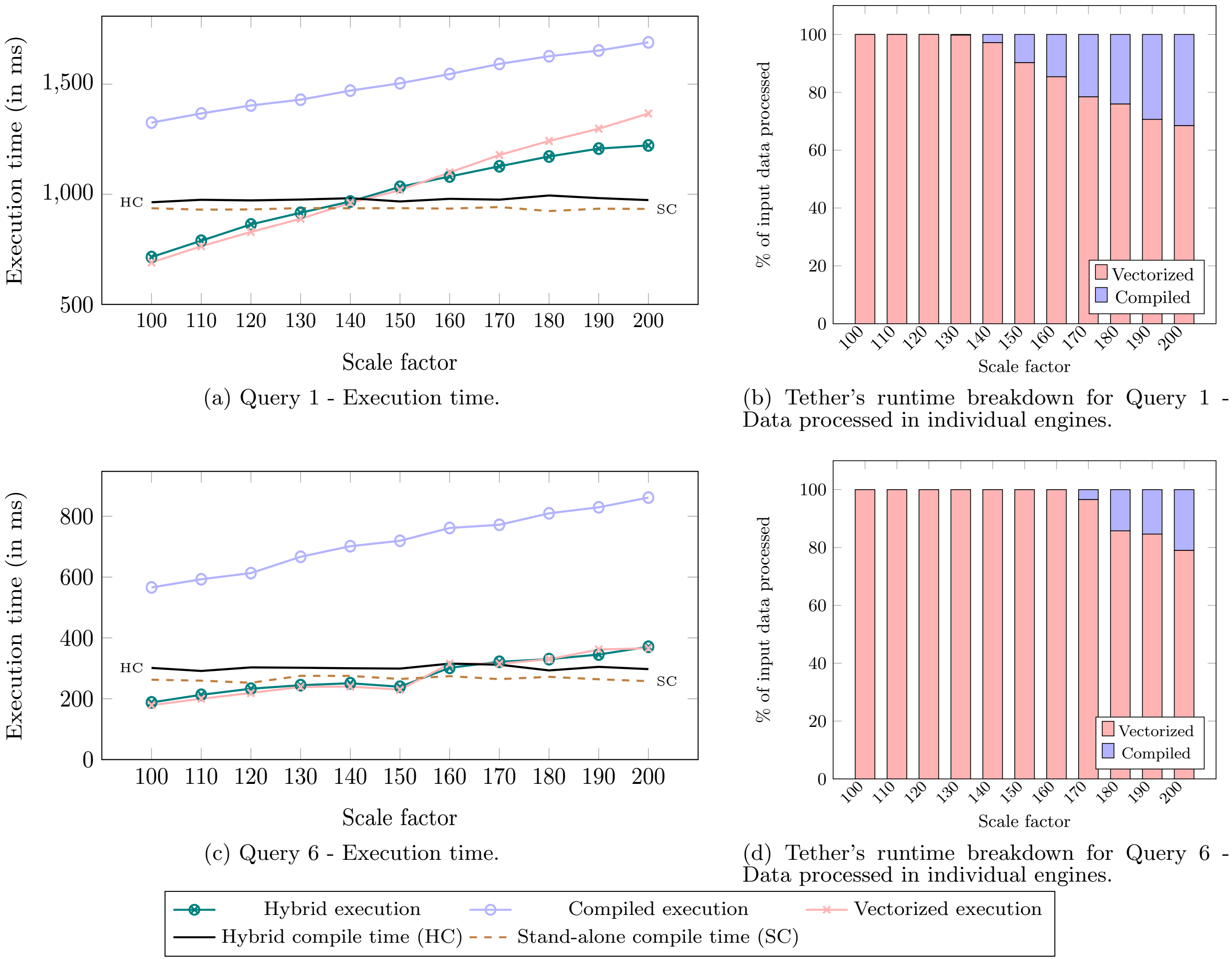
As discussed, Q3 is composed of three pipelines. Therefore, depending on the data size, vectorized execution could have finished some of the pipelines before compilation is complete. By injecting the switching point at the hash join, we see that we are either worse than or on par with the stand-alone compiled performance. This shows two implications: 1) a hybrid hash probe in the compiled engine of Tether incurs a performance penalty, 2) the vectorized engine of Tether is not completely busy until the compilation has finished. To have a better picture of the most performance-critical pipeline, we measured the execution time for individual pipelines and present them in Figure 8(b). We see that 60 % of Q3 is spent on the final pipeline (i.e., the hash aggregation). Using this pipeline as the switching point, we see a drastic improvement in performance. Since hash aggregation is the final pipeline in Q3, the execution characteristics are similar to that of Q1 and Q6. When compilation takes longer then vectorized execution, Tether follows the performance of vectorized execution. However, if the compilation finishes before vectorized execution, Tether outperforms the performance of vectorized execution due to compiled execution. Thus, similar to the case of Q1 and Q6, our overall improvement by Tether is about a factor of 2 compared to compiled execution and 1.5 to vectorized execution after the cut-off point.

As a final challenging query, we investigate Q18 with 5 pipelines to show the impact of changing the switching points between the engines. From Figure 9(a), we see that even with less than scale factor 100, compilation time is less than the total execution time of vectorized execution. Hence, during the transition from vectorized to compiled execution, Tether processes only a partial result of some internal pipeline. This connection pipeline, similar to that of Q3, has to be identified.

With compilation time being static for all the datasets, depending on the data size, vectorized will be processing a different part of the query pipeline (the bigger the amount of input data, the less pipelines have been already processed at the time of switching). Below are the pipelines for Q18:

1. Build customer hash table.
2. Group by lineitem.
3. Build lineitem hash table.
4. Probe orders over customer and lineitem tables and build the final table.
5. Probe lineitem and compute aggregates.

Out of these, the first pipeline on the customer table has only few values to process. Therefore the second pipeline with grouping of the lineitem table is taken as our cut-off



(a) Query 1 - Execution time.

(b) Tether's runtime breakdown for Query 1 - Data processed in individual engines.

(c) Query 6 - Execution time.

(d) Tether's runtime breakdown for Query 6 - Data processed in individual engines.

Figure 7: Evaluation for queries with single pipeline

pipeline-breaker. The execution of this query plan shows that we have the performance improvement from both engines only after SF 160. To better understand the impact of the switching points, we depict the percentage of time spent on each of the execution engines in Figure 9(b).

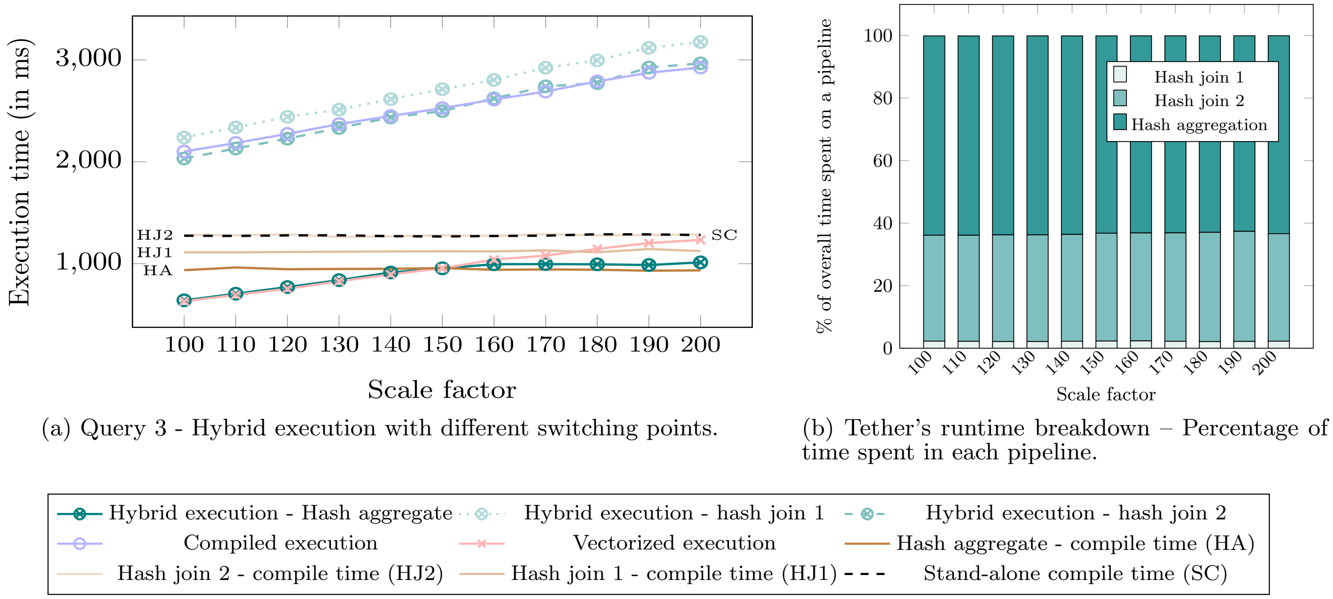
For scale factors 100 to 150, vectorized execution is able to process its given pipeline completely before the hybrid query is compiled. Therefore for these scale factors, we have a worse execution time than vectorized execution. With an increasing data size, this gap reduces as vectorized execution has to process more data.

One way to reduce the wait time is to move the switching point to the next pipeline, i.e., building the hash table of the lineitem table. However, this will not be an optimal choice. The subsequent pipeline, i.e., order's probing pipeline, probes over both the built hash tables of lineitem aggregates and customer from vectorized execution. With the first three pipelines executed by vectorized execution, compiled execution has to issue another probe call to the built customer table as well as a partially built lineitem table. These probe calls, as the results of Q3 show, are way too expensive. Hence, we keep using the second pipeline breaker as a switching point. Thus, with a penalty of smaller wait times, our Tether execution outperforms compiled execution by a factor of 1.5. Once we bridge the execution gap, we outperform vectorized execution by a factor of 1.2.

In summary, complex pipelined queries require a complete analysis to decide on the right switching points. Therefore, such queries could benefit from an optimizer providing these switching-points during runtime. We consider this as our future work.

6. DISCUSSION

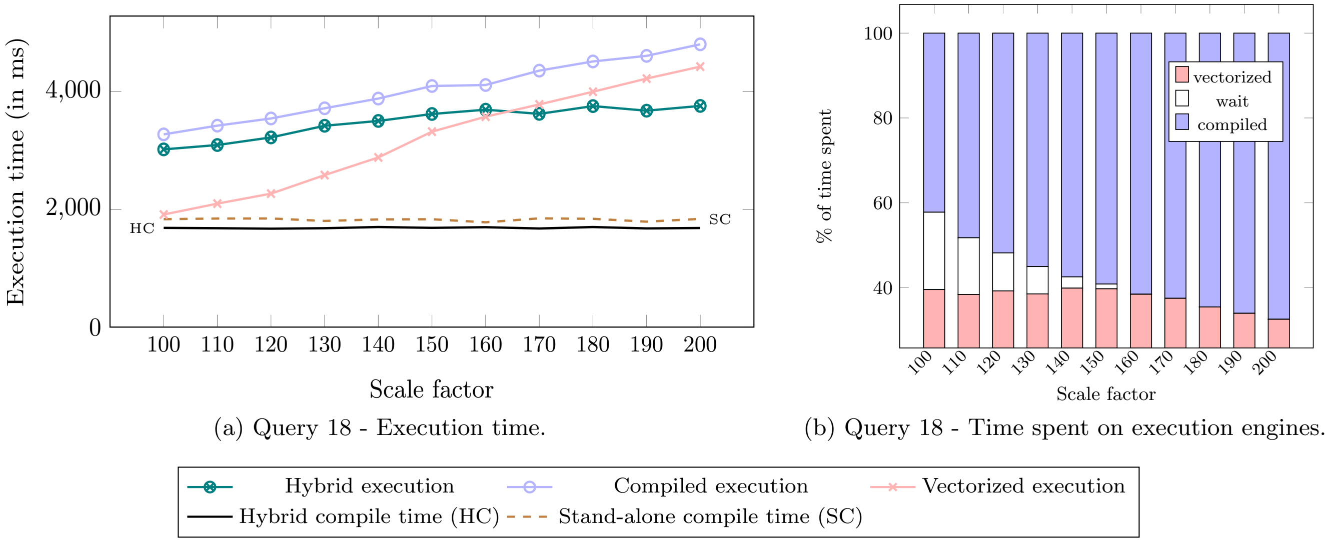
From our experiments, we show that our approach has achieved the overall best execution time. Our system is the fastest or it is on par with the execution time of vectorized execution, which completely hides the compilation time. On average, we are three times faster than the combined execution time for compilation and compiled query execution. We also outperform vectorized execution after the switching to compiled execution by up to a factor of two for bigger datasets. Based on the results, we have the following observations and discussion points:



(a) Query 3 - Hybrid execution with different switching points.

(b) Tether's runtime breakdown – Percentage of time spent in each pipeline.

Figure 8: Impact of switching points in Q3.



(a) Query 18 - Execution time.

(b) Query 18 - Time spent on execution engines.

Figure 9: Execution performance for Q18.

Hiding compilation: An overall positive result of our experiments is that Tether effectively hides compilation time. Hence, since starting the processing concurrent to compilation reduces the overall amount of data to be processed, the overall performance of Tether is above the performance of stand-alone systems.

Merging overhead: Despite the impressive performance of Tether, its main drawback is the inclusion of an extra merging step in the pipeline breaker. This leads to a sometimes increased compilation time and adds a processing overhead to Tether's compiled execution engine. Such overhead is visible in the performance difference of stand-alone compiled versus Tether's compiled execution in Q1 (cf., Figure 7(a)). Hence, one significant optimization space is the development of efficient merging strategies, that allow both engines to handshake more efficiently on switching.

Furthermore, especially when using the hash probe as a switching point, it is visible that the functional invocation of vectorized primitives inside compiled execution affects performance negatively. Therefore, important future work is to implement custom pipeline breakers that can be optimally executed in vectorized execution and read without overhead within compiled execution.

Performance critical pipelines: Another observation from Q3 is that different pipelines contribute to a different extent to the overall execution time. Especially pipelines that are composed of complex operators and process a large input dataset are promising for being started in vectorized execution. By identifying these pipelines and sharing their results, we can improve the overall performance significantly.

Selection of the right pipeline breaker: For the best performance of our hybrid system Tether, an optimal switching point is required. Such an optimal switching point is present at the cross-point of the performance impact of vectorized processing of the input query, input data size, and the compilation time. Hence, it is possible that vectorized execution processes more than one pipeline. As a consequence, an optimizer should choose the right pipeline breaker to switch between both engines of Tether.

Data size vs. compilation time: For datasets with considerably smaller data sizes, vectorized execution might complete processing the workload before compilation finishes. In this case, it is not beneficial to start compilation. Therefore, the query optimizer should decide on the right execution model based on the input sizes and the query compilation time.

7. CONCLUSION

In this work, we aim for solving the biggest pain-point of compiled execution: compilation times. To this end, we investigate whether its competing query engine, a vectorized engine, can help in effectively hiding compilation times. This leads to our hybrid engine Tether that starts query processing in its vectorized engine and after compilation finished, continues query processing in its compiled engine.

For switching between the execution paradigms, we use pipeline breakers as a natural switching point. We realize such a data sharing between the execution engines using the pipeline breakers: aggregation, hash aggregation, and hash join operators.

From our results, we show that by switching from vectorized to compiled execution we reach the best performance compared to both, vectorized and compiled execution. In fact, hiding compilation time using vectorized execution can improve performance by up to a factor of three compared to stand-alone vectorized or compiled query performance. Therefore, our *Tether* framework shows that query processing can be significantly improved when combining the features of the two state-of-the-art approaches.

However, our results also show that choosing the right pipeline breaker for switching between the engines is of significant importance. Hence, the query optimizer has to be extended in future work to incorporate these design decisions.

8. ACKNOWLEDGMENTS

This work was partially funded by the DFG (grant no.: SA 465/50-1, SA 465/51-1, and PI 447/9). Furthermore, we thank Pavlo Shevchenko for his implementation effort in this project.

9. REFERENCES

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1383–1394, 2015.

[2] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling UDF-centric workflows. *Proceedings of the VLDB Endowment*, 8(12):1466–1477, 2015.

[3] M. Dreseler, J. Kossmann, J. Frohnhofen, M. Uflacker, and H. Plattner. Fused table scans: Combining AVX-512 and JIT to double the performance of multi-predicate scans. In *Workshop Proceedings of the International Conference on Data Engineering (ICDEW)*, pages 102–109. IEEE, 2018.

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

[5] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment*, 11(13):2209–2222, 2018.

[6] A. Kohn, V. Leis, and T. Neumann. Adaptive execution of compiled queries. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 197–208. IEEE, 2018.

[7] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 613–624. IEEE, 2010.

[8] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.

[9] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[10] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Engineering Bulletin*, 37(1):3–11, 2014.

[11] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment*, 11(6):663–676, 2018.

[12] S. Wanderman-Milne and N. Li. Runtime code generation in Cloudera Impala. *IEEE Data Engineering Bulletin*, 37(1):31–37, 2014.

[13] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin*, 35(1):21–27, 2012.

[14] M. Zukowski, M. Van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. In *Proeedings of the International Conference on Data Engineering (ICDE)*, pages 1349–1350. IEEE, 2012.