Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask

Timo Kersten Viktor Leis Alfons Kemper Thomas Neumann Andrew Pavlo[◇] Peter Boncz[⋆]
Technische Universität München Carnegie Mellon University[◇] Centrum Wiskunde & Informatica[⋆]
{kersten,leis,kemper,neumann}@in.tum.de pavlo@cs.cmu.edu boncz@cwi.nl

ABSTRACT

The query engines of most modern database systems are either based on vectorization or data-centric code generation. These two state-of-the-art query processing paradigms are fundamentally different in terms of system structure and query execution code. Both paradigms were used to build fast systems. However, until today it is not clear which paradigm yields faster query execution, as many implementation-specific choices obstruct a direct comparison of architectures. In this paper, we experimentally compare the two models by implementing both within the same test system. This allows us to use for both models the same query processing algorithms, the same data structures, and the same parallelization framework to ultimately create an apples-to-apples comparison. We find that both are efficient, but have different strengths and weaknesses. Vectorization is better at hiding cache miss latency, whereas data-centric compilation requires fewer CPU instructions, which benefits cache-resident workloads. Besides raw, single-threaded performance, we also investigate SIMD as well as multi-core parallelization and different hardware architectures. Finally, we analyze qualitative differences as a guide for system architects.

PVLDB Reference Format:
T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, P. Boncz. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB*, 11 (13): 2209 - 2222, 2018.
DOI: https://doi.org/10.14778/3275366.3275370

1. INTRODUCTION

In most query engines, each relational operator is implemented using Volcano-style iteration [14]. While this model worked well in the past when disk was the primary bottleneck, it is inefficient on modern CPUs for in-memory database management systems (DBMSs). Most modern query engines therefore either use *vectorization* (pioneered by VectorWise [7, 52]) or *data-centric code generation* (pioneered by HyPer [28]). Systems that use vectorization include DB2 BLU [40], columnar SQL Server [21], and Quickstep [33], whereas systems based on data-centric code generation include Apache Spark [2] and Peloton [26].

Proceedings of the VLDB Endowment, Vol. 11, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3275366.3275370

Like the Volcano-style iteration model, vectorization uses pull-based iteration where each operator has a *next* method that produces result tuples. However, each *next* call fetches a block of tuples instead of just one tuple, which amortizes the iterator call overhead. The actual query processing work is performed by primitives that execute a simple operation on one or more type-specialized columns (e.g., compute hashes for a vector of integers). Together, amortization and type specialization eliminate most of the overhead of traditional engines.

In data-centric code generation, each relational operator implements a push-based interface (*produce* and *consume*). However, instead of directly processing tuples, the produce/consume calls generate code for a given query. They can also be seen as operator methods that get called during a depth-first traversal of the query plan tree, where produce is called on first visit, and consume on last visit, after all children have been processed. The resulting code is specialized for the data types of the query and fuses all operators in a pipeline of non-blocking relational operators into a single (potentially nested) loop. This generated code can then be compiled to efficient machine code (e.g., using the LLVM).

Although both models eliminate the overhead of traditional engines and are highly efficient, they are conceptually different from each other: Vectorization is based on the pull model (root-to-leaf traversal), vector-at-a-time processing, and interpretation. Data-centric code generation uses the push model (leaf-to-root traversal), tuple-at-a-time processing, and up-front compilation. As we discuss in Section 9, other designs that mix or combine ideas from data-centric compilation and vectorization have been proposed. In this paper, we focus on these two specific designs, as they have been highly influential and are in use in multiple widespread systems.

The differences of the two models are fundamental and determine the organization of the DBMS's execution engine source code and its performance characteristics. Because changing the model requires rewriting large parts of the source code, DBMS designers must decide early on which model to use. Looking at recent DBMS developments like Quickstep [33] and Peloton [26], we find that both choices are popular and plausible: Quickstep is based on vectorization, Peloton uses data-centric code generation.

Given the importance of this choice, it is surprising that there has not yet been a systematic study comparing the two state-of-the-art query processing models. In this paper, we provide an in-depth experimental comparison of the two models to understand when a database architect should prefer one model over the other.

To compare vectorization and compilation, one could compare the runtime performance of emblematic DBMSs, such as HyPer and VectorWise. The problem is, however, that such full-featured DBMSs differ in many design dimensions beyond the query execution model. For instance, HyPer does not employ sub-byte com-

pression in its columnar storage [19], whereas VectorWise uses more compact compression methods [53]. Related to this choice, HyPer features predicate-pushdown in scans but VectorWise does not. Another important dimension in which both systems differ is parallelism. VectorWise queries spawn threads scheduled by the OS, and controls parallelism using explicit *exchange* operators where the parallelism degree is fixed at query optimization time [3]. HyPer, on the other hand, runs one thread on each core and explicitly schedules query tasks on it on a morsel-driven basis using a NUMA-aware, lock-free queue to distribute work. HyPer and VectorWise also use different query processing algorithms and structures, data type representations, and query optimizers. Such different design choices affect performance and scalability, but are independent of the query execution model.

To isolate the fundamental properties of the execution model from incidental differences, we implemented a compilation-based relational engine and a vectorization-based engine in a single test system (available at [16]). The experiments where we employed data-centric code-generation into C++[1] we call "Typer" and the vectorized engine we call "Tectorwise" (TW). Both implementations use the same algorithms and data structures. This allows an apples-to-apples comparison of both approaches because the only difference between Tectorwise and Typer is the query execution method: vectorized versus data-centric compiled execution.

Our experimental results show that both approaches lead to very efficient execution engines, and the performance differences are generally not very large. Compilation-based engines have an advantage in calculation-heavy queries, whereas vectorized engines are better at hiding cache miss latency, e.g., during hash joins.

After introducing the two models in more detail in Section 2 and describing our methodology in Section 3, we perform a micro-architectural analysis of in-memory OLAP workloads in Section 4. We then examine in Section 5 the benefit of data-parallel operations (SIMD), and Section 6 discusses intra-query parallelization on multi-core CPUs. In Section 7, we investigate different hardware platforms (Intel, AMD, Xeon Phi) to find out which model works better on which hardware. After these quantitative OLAP performance comparisons, we discuss other factors in Section 8, including OLTP workloads and compile time. A discussion of hybrid processing models follows in Section 9. We conclude by summarizing our results as a guide for system designers in Section 10.

2. VECTORIZED VS. COMPILED QUERIES

The main principle of vectorized execution is *batched* execution [30] on a columnar data representation: every "work" primitive function that manipulates data does not work on a single data item, but on a vector (an array) of such data items that represents multiple tuples. The idea behind vectorized execution is to amortize the DBMS's interpretation decisions by performing as much as possible inside the data manipulation methods. For example, this work can be to hash 1000s of values, compare 1000s of string pairs, update a 1000 aggregates, or fetch a 1000 values from 1000s of addresses.

Data-centric compilation generates low-level code for a SQL query that fuses all adjacent non-blocking operators of a query pipeline into a single, tight loop. In order to understand the properties of vectorized and compiled code, it is important to understand the structure of each variant's code. Therefore, in this section we present example operator implementations, motivate why they are implemented in this fashion, and discuss some of their properties.

[1]HyPer compiles to LLVM IR rather than C++, but this choice only affects compilation time (which we ignore in this paper anyway), not execution time.

```
vec<int> sel_eq_row(vec<string> col, vec<int> tir)
  vec<int> res;
  for(int i=0; i<col.size(); i++) // for colors and tires
    if(col[i] == "green" && tir[i] == 4) // compare both
      res.append(i) // add to final result
  return res
```

(a) Integrated: Both predicates checked at once

```
vec<int> sel_eq_string(vec<string> col, string o)
  vec<int> res;
  for(int i=0; i<col.size(); i++) // for colors
    if(col[i] == o) // compare color
      res.append(i) // remember position
  return res

vec<int> sel_eq_int(vec<int> tir, int o, vec<int> s)
  vec<int> res;
  for(i : s) // for remembered position
    if(tir[i] == o) // compare tires
      res.append(i) // add to final result
  return res
```

(b) Vectorized: Each predicate checked in one primitive

Figure 1: **Multi-Predicate Example** – *The straightforward way to evaluate multiple predicates on one data item is to check all at once (1a). Vectorized code must split the evaluation into one part for each predicate (1b).*

2.1 Vectorizing Algorithms

Typer executes queries by running generated code. This means that a developer can create operator implementations in any way they see fit. Consider the example in Figure 1a: a function that selects every row whose color is green and has four tires. There is a loop over all rows and in each iteration, all predicates are evaluated.

Tectorwise implements the same algorithms as Typer, staying as close to it as possible and reasonable (for performance). This is, however, only possible to a certain degree, as every function implemented in vectorized style has two constraints: It can (i) only work on *one* data type[2] and it (ii) must process multiple tuples. In generated code these decisions can both be put into the expression of one `if` statement. This, however, violates (i) which forces Tectorwise to use two functions as shown in Figure 1b. A (not depicted) interpretation logic would start by running the first function to select all elements by color, then the second function to select by number of tires. By processing multiple elements at a time, these functions also satisfy (ii). The dilemma is faced by all operators in Tectorwise and all functions are broken down into primitives that satisfy (i) and (ii). This example uses a column-wise storage format, but row-wise formats are feasible as well. To maximize throughput, database developers tend to highly optimize such functions. For example, with the help of predicated evaluation (`*res=i; res+=cond`) or SIMD vectorized instruction logic (see Section 5.1).

With these constraints in mind, let us examine the details of operator implementations of Tectorwise. We implemented selections as shown above. Expressions are split by arithmetic operators into primitives in a similar fashion. Note that for these simple operators the Tectorwise implementation must already change the structure of the algorithms and deviate from the Typer data access patterns. The resulting materialization of intermediates makes fast caches very important for vectorized engines.

2.2 Vectorized Hash Join and Group By

Pseudo code for parts of our hash join implementations are shown in Figure 2. The idea for both, the implementation in Typer and

[2]Technically, it would be possible to create primitives that work on multiple types. However, this is not practical, as the number of combinations grows exponentially.

```
query(...)
  // build hash table
  for(i = 0; i < S.size(); i++)
    ht.insert(<S.att1[i], S.att2[i]>, S.att3[i])
  // probe hash table
  for(i = 0; i < R.size(); i++)
    int k1 = R.att1[i]
    string* k2 = R.att2[i]
    int hash = hash(k1, k2)
    for(Entry* e = ht.find(hash); e; e = e->next)
      if(e->key1 == k1 && e->key2 == *k2)
        ... // code of parent operator
```

(a) Code generated for hash join

```
class HashJoin
  Primitives probeHash_, compareKeys_, buildGather_;
  ...
int HashJoin::next()
  ... // consume build side and create hash table
  int n = probe->next()// get tuples from probe side
  // *Interpretation*: compute hashes
  vec<int> hashes = probeHash_.eval(n)
  // find hash candidate matches for hashes
  vec<Entry*> candidates = ht.findCandidates(hashes)
  // matches: int references a position in hashes
  vec<Entry*, int> matches = {}
  // check candidates to find matches
  while(candidates.size() > 0)
    // *Interpretation*
    vec<bool> isEqual = compareKeys_.eval(n, candidates)
    hits, candidates = extractHits(isEqual, candidates)
    matches += hits
  // *Interpretation*: gather from hash table into
  // buffers for next operator
  buildGather_.eval(matches)
  return matches.size()
```

(b) Vectorized code that performs a hash join

Figure 2: **Hash Join Implementations in Typer and Tectorwise** – *Generated code (Figure 2a) can take any form, e.g., it can combine the equality check of hash table keys. In vectorized code (Figure 2b), this is only possible with one primitive for each check.*

Tectorwise, is to first consume all tuples from one input and place them into a hash table. The entries are stored in row format for better cache locality. Afterwards, for each tuple from the other input, we probe the hash table and yield all found combinations to the parent operator. The corresponding code that Typer generates is depicted in Figure 2a.

Tectorwise cannot proceed in exactly the same manner. Probing a hash table with composite keys is the intricate part here, as each probe operation needs to test equality of all parts of the composite key. Using the former approach would, however, violate (i). Therefore, the techniques from Section 2.1 are applied: The join function first creates hashes from the probe keys. It does this by evaluating the `probeHash` expression. A user of the vectorized hash join must configure the `probeHash` and other expressions that belong to the operator so that when the expressions evaluate, they use data from the operator's children. Here, the `probeHash` expression hashes key columns by invoking one primitive per key column and writes the hashes into an output vector. The join function then uses this vector of hashes to generate candidate match locations in the hash table. It then inspects all discovered locations and checks for key equality. It performs the equality check by evaluating the `cmpKey` expression. For composite join-keys, this invokes multiple primitives: one for every key column, to avoid violating (i) and (ii). Then, the join function adds the matches to the list of matching tuples, and, in case any candidates have an overflow chain, it uses the overflow entries as new candidates for the next iteration. The algorithm continues until the candidate vector is empty. Afterwards, the join uses `buildGather` to move data from the hash table into buffers for the next operator.

We take a similar approach in the group by operator. Both phases of the aggregation use a hash table that contains group keys and aggregates. The first step for all inbound tuples is to find their group in the hash table. We perform this with the same technique as in the hash join. For those tuples whose group is not found, one must be added. Unfortunately, it is not sufficient to just add one group per group-less tuple as this could lead to groups added multiple times. We therefore shuffle all group-less tuples into partitions of equal keys (proceeding component by component for composite keys), and add one group per partition to the hash table. Once the groups for all incoming tuples are known we run aggregation primitives. Transforming into vectorized form led to an even greater deviation from Typer data access patterns. For the join operator, this leads to more independent data accesses (as discussed in Section 4.1). However, aggregation incurs extra work.

Note that in order to implement Tectorwise operators we need to deviate from the Typer implementations. This deviation is not by choice, but due to the limitations (i) and (ii) which vectorization imposes. This yields two different implementations for each operator, but at its core, each operator executes the same algorithm with the same parallelization strategy.

3. METHODOLOGY

To isolate the fundamental properties of the execution model from incidental differences found in real-world systems, we implemented a compilation-based engine (Typer) and a vectorization-based engine (Tectorwise) in a single test system (available at [16]). To make experiments directly comparable, both implementations use the same algorithms and data structures. When testing queries, we use the same physical query plans for vectorized and compiled execution. We do not include query parsing, optimization, code generation, and compilation time in our measurements. This testing methodology allows an apples-to-apples comparison of both approaches because the only difference between Tectorwise and Typer is the query execution method: vectorized versus data-centric compiled execution.

3.1 Related Work

Vectorization was proposed by Boncz et al. [7] in 2005. It was first used in MonetDB/X100, which evolved into the commercial OLAP system VectorWise, and later adopted by systems like DB2 BLU [40], columnar SQL Server [21], and Quickstep [33]. In 2011, Neumann [28] proposed data-centric code generation using the LLVM compiler framework as the query processing model of HyPer, an in-memory hybrid OLAP and OLTP system. It is also used by Peloton [26] and Spark [2].

To the best of our knowledge, this paper is the first systemic comparison of vectorization and data-centric compilation. Sompolski et al. [45] compare the two models using a number of microbenchmarks, but do not evaluate end-to-end performance for full queries. More detailed experimental studies are available for OLTP systems. Appuswamy et al. [4] evaluate different OLTP system architectures in a common prototype, and Sirin et al. [43] perform a detailed micro-architectural analysis of existing commercial and open source OLTP systems.

3.2 Query Processing Algorithms

We implemented five relational operators both in Tectorwise and Typer: *scan, select, project (map), join, and group by*. The scan operator at its core consists of a (parallel) `for` loop over the scanned

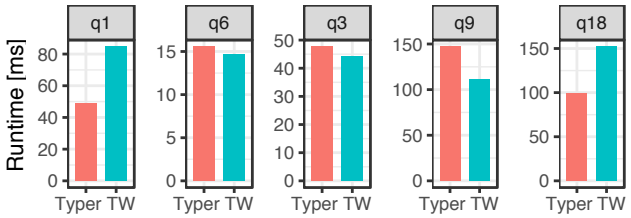


Figure 3: **Performance** – *TPC-H SF=1, 1 thread*

relation. Select statements are expressed as `if` branches. Projection is achieved by transforming the expression to the corresponding C code. Unlike production-grade systems, our implementation does not perform overflow checking of arithmetic expressions. Join uses a single hash table[3] with chaining for collision detection. Using 16 (unused) bits of each pointer, the hash table dictionary encodes a small Bloom filter-like structure [22] that improves performance for selective joins (a probe miss usually does not have to traverse the collision list). The group by operator is split into two phases for cache friendly parallelization. A pre-aggregation handles heavy hitters and spills groups into partitions. Afterwards, a final step aggregates the groups in each partition. Using these algorithms in data-centric code is quite straightforward, while vectorization requires adaptations, which we describe in Section 2.1.

3.3 Workload

In this paper we focus on OLAP performance, and therefore use the well-known TPC-H benchmark for most experiments. To be able to show detailed statistics for each individual query as opposed to only summary statistics, we chose a representative subset of TPC-H. The selected queries and their performance bottlenecks are listed in the following:

- **Q1:** fixed-point arithmetic, (4 groups) aggregation
- **Q6:** selective filters
- **Q3:** join (build: 147 K entries, probe: 3.2 M entries)
- **Q9:** join (build: 320 K entries, probe: 1.5 M entries)
- **Q18:** high-cardinality aggregation (1.5 M groups)

The given cardinalities are for scale factor (SF) 1 and grow linearly with it. Of the remaining 17 queries, most are dominated by join processing and are therefore similar to Q3 and Q9. A smaller number of queries spend most of the time in a high-cardinality aggregation and are therefore similar to Q18. Finally, despite being the only two single-table queries, we show results for both Q1 and Q6 as they behave quite differently. Together, these five queries cover the most important performance challenges of TPC-H and any execution engine that performs well on them will likely be also efficient on the full TPC-H suite [6].

3.4 Experimental Setup

Unless otherwise noted, we use a system equipped with an Intel i9-7900X (Skylake X) CPU with 10 cores for our experiments. Detailed specifications for this CPU can be found in the hardware section in Table 4. We use Linux as OS and compile our code with GCC 7.2. The CPU counters were obtained using Linux' perf events API. Throughout this paper, we normalize CPU counters by the total number of tuples scanned by that query (i.e., the sum of the cardinalities of all tables scanned). This normalization enables intuitive observations across systems (e.g., "Tectorwise executes 41 instructions per tuple more than Typer on query 1") as well as interesting comparisons across other dimensions (e.g., "growing the data size by a factor of 10, causes 0.5 additional cache misses per tuple").

[3] Although recent research argues for partitioned hash joins [5, 41], single-table joins are still prevalent in production systems and are used by both HyPer and VectorWise.

Table 1: **CPU Counters** – *TPC-H SF=1, 1 thread, normalized by number of tuples processed in that query*

		cycles	IPC	instr.	L1 miss	LLC miss	branch miss
Q1	Typer	34	2.0	68	0.6	0.57	0.01
Q1	TW	59	2.8	162	2.0	0.57	0.03
Q6	Typer	11	1.8	20	0.3	0.35	0.06
Q6	TW	11	1.4	15	0.2	0.29	0.01
Q3	Typer	25	0.8	21	0.5	0.16	0.27
Q3	TW	24	1.8	42	0.9	0.16	0.08
Q9	Typer	74	0.6	42	1.7	0.46	0.34
Q9	TW	56	1.3	76	2.1	0.47	0.39
Q18	Typer	30	1.6	46	0.8	0.19	0.16
Q18	TW	48	2.1	102	1.9	0.18	0.37

4. MICRO-ARCHITECTURAL ANALYSIS

To understand the two query processing paradigms, we perform an in-depth micro-architectural comparison. We initially focus on sequential performance and defer discussing data-parallelism (SIMD) to Section 5 and multi-core parallelization to Section 6.

4.1 Single-Threaded Performance

Figure 3 compares the single-threaded performance of the two models for selected TPC-H queries. For some queries (Q1, Q18), Typer is faster and for others (Q3, Q9) Tectorwise is more efficient. The relative performance ranges from Typer being faster by 74% (Q1) to Tectorwise being faster by 32% (Q9). Before we look at the reasons for this, we note that these are not large differences, especially when compared to the performance gap to other systems. For example the difference between HyPer and PostgresSQL is between one and two orders of magnitude [17]. In other words, the performance of both query processing paradigms is quite close—despite the fact that the two models appear different from the point of someone implementing these systems. Nevertheless, neither paradigm is clearly dominated by the other which makes both viable options to implement a processing engine. Therefore, in the following we analyze the performance differences to understand the strengths and weaknesses of the two models.

Table 1 shows some important CPU statistics, from which a number of observations can be made. First, Tectorwise executes significantly more instructions (up to 2.4×) and usually has more L1 data cache misses (up to 3.3×). Tectorwise breaks all operations into simple steps and must materialize intermediate results between these steps, which results in additional instructions and cache accesses. Typer, in contrast, can often keep intermediate results in CPU registers and thus perform the same operations with fewer instructions. Based on these observations, it becomes clear why Typer is significantly faster on Q1. This query is dominated by fixed-point arithmetic operations and a cheap in-cache aggregation. In Tectorwise intermediate results must be materialized, which is similarly expensive as the computation itself. Thus, one key difference between the two models is that Typer is more efficient for computational queries that can hold intermediate results in CPU registers and have few cache misses.

We observe furthermore, that for Q3 and Q9, whose performance is determined by the efficiency of hash table probing, Tectorwise is faster than Typer (by 4% and 32%). This might be surprising given the fact that both engines use exactly the same hash table layout

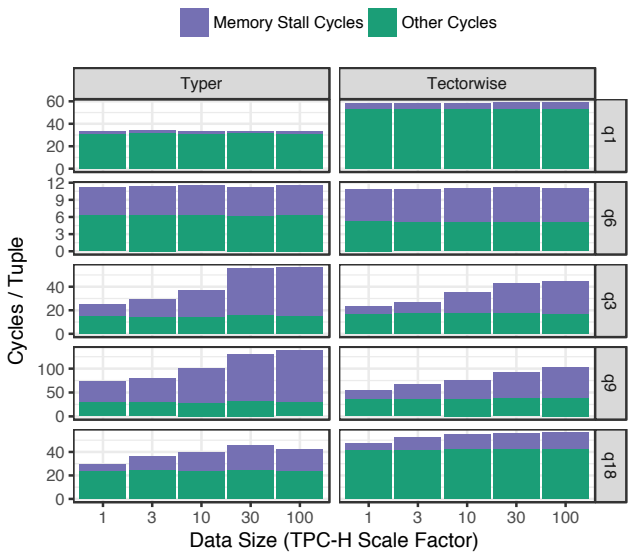


Figure 4: **Memory Stalls** – *TPC-H, 1 thread*

and therefore also have an almost identical number of last level cache (LLC) misses. As Figure 4 shows, Tectorwise's join advantage increases up to 40% for larger data (and hash table) sizes. The reason is that vectorization is better at hiding cache miss latency, as observed from the *memory stall* counter that measures the number of cycles during which the CPU is stalled waiting for memory. This counter explains the performance difference. On the one hand, Tectorwise's hash table probing code is only a simple loop. It executes only hash table probes thus the CPU's out-of-order engine can speculate far ahead and generate many outstanding loads. These can even be executed out of order. On the other hand, Typer's code has more complex loops. Each loop can contain code for a scan, selection, hash-table probe, aggregation and more. The out-of-order window of each CPU fills up more quickly with complex loops thus they generate less outstanding loads. In addition every branch miss is more expensive than in a complex loop as more work that is performed under speculative execution is discarded and must be repeated on a miss. Overall, Tectorwise's simpler loops enable better latency hiding.

Another difference between the two executions models is their sensitivity regarding the hash function. After trying different hash functions, we settled on Murmur2 for Tectorwise, and a CRC-based hash function, which combines two 32-bit CRC results into a single 64-bit hash, for Typer. Murmur2 requires twice as many instructions as CRC hashing, but has higher throughput and is therefore slightly faster in Tectorwise, which separates hash computation from probing. For Typer, in contrast, the CRC hash function improves the performance up to 40% on larger scale factors—even though most time is spent waiting for cache misses. The lower latency and smaller number of instructions for CRC significantly improve the speculative, pipelined execution of consecutive loop iterations, thereby enabling more concurrent outstanding loads.[4]

As a note of caution, we remark that one may observe from Table 1 that Tectorwise generally executes more instructions per cycle (IPC) and deduce that Tectorwise performs better. However, this is not necessarily correct. While IPC is a measure of CPU utilization, having a higher IPC is not always better: As can be observed in Q1, Tectorwise's IPC is 40% higher, but it is still 74% slower due

[4]Despite using different hash functions, this is still a fair comparison of join performance, as each system uses the more beneficial hash function.

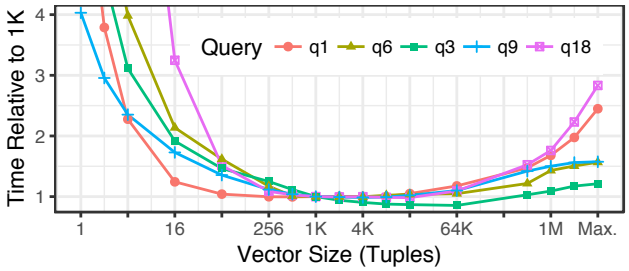


Figure 5: **Tectorwise Vector Sizes** – *Times are normalized by 1K vector time.*

to executing almost twice the number of instructions. This means that one has to be cautious when using IPC to compare database systems' performance. It is a valid measure of the amount of free processing resources, but should not be used as the sole proxy for overall query processing performance.

To summarize, looking at the micro-architectural footprint of the two models we found that (1) both are efficient and fairly close in performance, (2) Typer is more efficient for computational queries with few cache misses, and (3) Tectorwise is slightly better at hiding cache miss latency.

4.2 Interpretation and Instruction Cache

Systems based on Volcano-style iteration perform expensive virtual function calls and type dispatch for each processed tuple. This is a form of interpretation overhead as it does not contribute to the actual query processing work. Generating machine code for a given query, by definition, avoids interpretation overhead. Vectorized systems like VectorWise are still fundamentally interpretation-based engines and use Volcano-style iteration. In contrast to classical database systems, the interpretation overhead is not incurred for each tuple but is amortized across the eponymous *vector* of tuples. Each primitive is specialized for a particular data type and is called for (e.g., 1,000 values). This amortization is effective: Using a profiler, we determined that across our query set the interpreted part is less than 1.5% of the query runtime (measured at scale factor 10). Thus, the DBMS spends 98.5% of its time in primitives doing query processing work. From Table 1 we observe that vectorized code usually executed more instructions per tuple than compiled code. Since the vast majority of the query execution time is spent within primitives, also the time to execute these extra instructions must be spent within primitives. As primitives know all involved types at compile time, we conclude that the extra instructions are not interpretation code that is concerned with interpretation decisions and virtual function calls. It is rather due to the load/store instructions for materializing primitive results into vectors.

Recent work has found that instruction cache misses can be a problem for OLTP workloads [43]. To find out whether this is the case for our two query engines, we measured L1 instruction cache misses for both systems and found that instruction cache misses are negligible, thus not a performance bottleneck for OLAP queries. For all queries measured, the L1 instruction cache (32 KB) was large enough to contain all hot code.

4.3 Vector Size

The vector size is an important parameter for any vectorized engine. So far, our Tectorwise experiments used a value of 1,000 tuples, which is also the default in VectorWise. Figure 5 shows normalized query runtimes for vector sizes from 1 to the maximum (i.e., full materialization). We observe that small (<64) and large vector sizes (>64 K) decrease performance significantly. With a

vector size of 1, Tectorwise is a Volcano-style interpreter with its large CPU overhead. Large vectors do not fit into the CPU caches and therefore cause cache misses. The other end of the spectrum is to process the query one column at a time; this approach is used in MonetDB [9]. Generally, a vector size of 1,000 seems to be a good setting for all queries. The only exception is Q3, which executes 15% faster using a vector size of 64K.

4.4 Star Schema Benchmark

So far, we investigated a carefully selected subset of TPC-H. To show that our findings are more generally applicable, we also implemented the Star Schema Benchmark (SSB), which consists of 4 query templates (with different selections) and which is dominated by hash table probes. We use one thread and scale factor 30:

		cycles	IPC	instr.	L1 miss	LLC miss	branch miss	mem stall
Q1.1	Typer	28	0.7	21	0.3	0.31	0.69	6.33
Q1.1	TW	12	2.0	23	0.4	0.29	0.05	2.77
Q2.1	Typer	39	0.8	30	1.3	0.12	0.17	18.35
Q2.1	TW	30	1.5	44	1.6	0.13	0.23	7.63
Q3.1	Typer	55	0.7	40	1.1	0.20	0.24	27.95
Q3.1	TW	53	1.3	71	1.7	0.23	0.41	15.68
Q4.1	Typer	78	0.5	39	1.8	0.31	0.38	45.91
Q4.1	TW	59	1.0	61	2.5	0.32	0.63	19.48

These results are quite similar to TPC-H Q3 and Q9 and show once more that Tectorwise requires more instructions but has an advantage for join heavy queries due to better hidden memory stalls. In general, we find that TPC-H subsumes SSB for our purposes and in the name of conciseness, we present our findings using TPC-H in the rest of this paper.

4.5 Tectorwise/Typer vs. VectorWise/HyPer

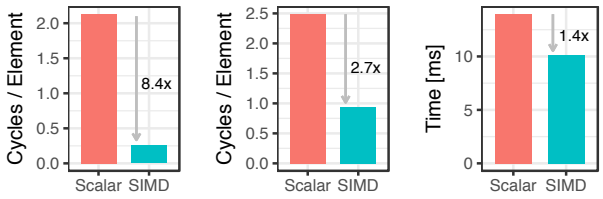
Table 2: **Production Systems**

	HyPer	VW	Typer	TW
Q1	53	71	**44**	85
Q6	**10**	21	15	15
Q3	48	50	47	**44**
Q9	124	154	126	**111**
Q18	224	159	**90**	154

Let us close this section by comparing Actian Vector 5.0 (the current marketing name for VectorWise) and the research (TUM) version of HyPer (a related system is now with Tableau). The results are shown in Table 2 use one thread and TPC-H scale factor 1. The first observation is that HyPer performs similarly to Typer, and Tectorwise's performance is similar to VectorWise. Second, except for Q6[5], either Typer or Tectorwise are slightly faster than both production-grade systems. It is unsurprising given that these must handle complex issues like overflow checking (that our prototype ignores).

5. DATA-PARALLEL EXECUTION (SIMD)

Let us now turn our attention to data-parallel execution using SIMD operations. There has been extensive research investigating SIMD for database operations [51, 50, 36, 37, 38, 35, 46, 44]. It is not surprising that this research generally assumes a vectorized execution model. The primitives of vectorized engines consist of simple tight loops that can be translated to data-parallel code. Though there has been research on utilizing SIMD in data-centric code [34, 26], this is more challenging since the generated code is more complex. We will therefore use Tectorwise as the platform for evaluating how large the impact of SIMD on in-memory OLAP workloads is. In contrast to most research on SIMD, we use TPC-H and not micro-benchmarks.



(a) Dense Input (b) Sparse Input (c) TPC-H Q6

Figure 6: **Scalar vs. SIMD Selection in Tectorwise** – *(a) 40% selectivity. (b) Secondary selection: Input selection vector selects 40% and selection selects 40%. (c) Runtime of TPC-H Q6, SF=1*

The Skylake X CPU we use for this paper supports the new AVX-512 instruction set and can execute two 512-bit SIMD operations per cycle—doubling register widths and throughput in comparison with prior microarchitectures. In other words, using AVX-512 one can process 32 values of 32-bit per cycle, while scalar code is limited to 4 values per cycle. Furthermore, in comparison with prior SIMD instruction sets like AVX2, AVX-512 is more powerful (almost all operations support masking and there are new instructions like compress and expand) and orthogonal (almost all operations are available in 8, 16, 32, and 64-bit variants). One would therefore expect significant benefits from using SIMD. In the following, we focus on selection and hash table probing, which are both common and important operations.

5.1 Data-Parallel Selection

A vectorized selection primitive produces a selection vector containing the indexes of all matching tuples. Using AVX-512 this can be implemented using SIMD quite easily[6]. The comparison instruction generates a mask that we then pass to a compress store (`COMPRESSSTORE`) instruction. This operation works across SIMD lanes and writes out the positions selected by the mask to memory.

We performed a micro-benchmark for selection, comparing a branch-free scalar x86 implementation with a SIMD variant. In the benchmark, we select all elements from an 8192 element integer array which are smaller than a constant. Results for a best-case scenario, in which all consumed data are 32-bit integers, are present in the L1 cache, and the input is a contiguous vector, are shown in Figure 6a. The observed performance gain for this micro-benchmark is 8.4×. However, as Figure 6c shows, in a realistic query with multiple expensive selections like Q6, we only observe a speedup of 1.4×—even though almost 90% of the processing time is spent in SIMD primitives. Our experiments revealed two effects that account for this discrepancy: *sparse data loading* due to selection vectors and *cache misses* due to varying stride. The remainder of this section discusses these effects.

Sparse data loading occurs in all selection primitives except for the first one. From the second selection primitive on, all primitives receive a selection vector that determines the elements to consider for comparison. These elements must be gathered from non-contiguous memory locations. A comparison of selection primitives with selection vectors (40% selectivity) is shown in Figure 6b. Performance gains in this case range only up to a 2.7× (again for

[5] HyPer is faster on Q6 than the other systems because it evaluates selections using SIMD instructions directly on compressed columns [19].

[6] With AVX2, the selection primitive is non-trivial and either requires a lookup table [35, 19, 26] or complex permutation logic based on BMI2 [1].

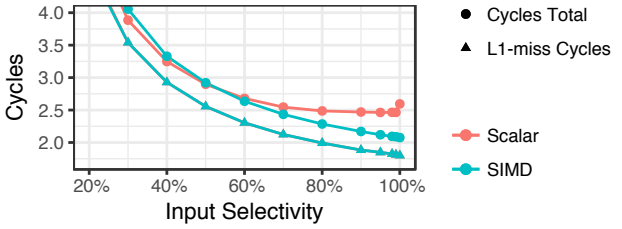


Figure 7: **Sparse Selection** – *Cost of selection operation with selection vector on input, depending on element size and implementation variant. Output selectivity 40%, data set size 4 GB*

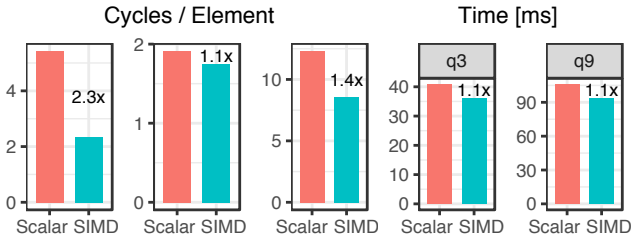


Figure 8: **Scalar vs. SIMD Join Probing** – *in microbenchmarks and full queries.*

32-bit types). Considering that the selections in Q6 consist of one initial selection without input selection vector and four subsequent selections that have to consider a selection vector, we can expect the overall speedup to be closer to 3× than to 8×.

The previous benchmarks only considered data sets which reside in L1 cache. For larger data sets, accessing memory can become a limiting factor. Figure 7 shows the interplay of selection performance and input sparsity on a 4 GB data set. Note that the performance drops for selectivities below 100%, while the scalar and SIMD variants are nearly equal when the is selectivity are below 50%. We also show an estimate of how many cycles on average are spent resolving cache misses. We observe that most of the time is spent waiting for data. Thus the memory subsystem becomes the bottleneck of the selection operation and the positive effect of utilizing SIMD instructions disappears. In the selection cascade of Q6, only the first selection primitive benefits from SIMD and selects 43% of the tuples. This leaves all subsequent selections to operate in a selectivity area where the scalar variant is just as fast.

5.2 Data-Parallel Hash Table Probing

We next examine hash join probing, where most of the query processing time is spent in TPC-H. There are two opportunities to apply SIMD: computing the hash values, and the actual lookups in the hash table. For hashing we use Murmur2, which consists of

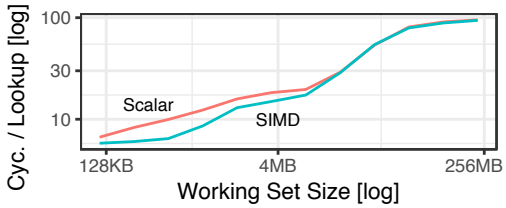


Figure 9: **Join Probe** – *Interaction of working set size and cost per tuple during Tectorwise hash table lookup*

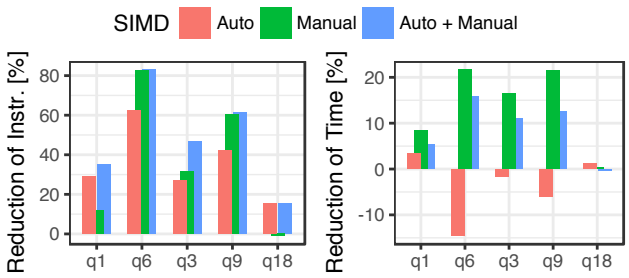


Figure 10: **Compiler Auto-Vectorization** – *(ICC 18)*

arithmetic operations like integer shifts and multiplications that are available in AVX-512. We can also apply SIMD to lookups into hash tables by using gather, compress store, and masking.

A performance breakdown of components necessary for hash joins is shown in Figure 8. Figure 8(a) shows that for hashing alone a gain of 2.3× is possible. For gather instructions, shown in Figure 8(b), we observe an improvement of 1.1× (in the best case). This is because the memory system of the test machine can perform at most two load operations per cycle—regardless of whether SIMD gather or scalar loads are used. Figure 8(c) shows that when employing gather and other SIMD instructions to the Tectorwise probe primitive, a best-case performance gain of 1.4× can be achieved.

With a SIMD speedup of 2.3× for hashing and 1.4× for probing, one may expect an overall speedup in between. However, as is shown in Figure 8(d) the performance gains almost vanish for TPC-H join queries. This happens even though the majority of the time (55% and 65%) is spent in SIMD-optimized primitives. The reason for this behavior can be found in Figure 9. With a growing working set, gains from SIMD diminish and the execution costs are dominated by memory latency. SIMD is only beneficial when all data fits into the cache. We are not the first to observe this phenomenon: Polychroniou et al. [35] found this effect in their study of application of SIMD to database operators.

5.3 Compiler Auto-Vectorization

We manually rewrote Tectorwise primitives using SIMD intrinsics. Given that the code of most primitives is quite simple, one may reasonably ask whether compilers can do this job automatically. We tested the GCC 7.2, Clang 5.0, and ICC 18 compilers. Of these, only ICC was able to auto-vectorize a fair amount of primitives (and only with AVX-512). Figure 10 shows how successful ICC was in relevant paths for query processing. Its vectorized variant reduces the observed number of instructions executed per tuple by between 20% to 60%. By inspecting traces of the executed code, we confirmed that automatic vectorization was applied to hashing, selection, and projection primitives. Hash table probing and aggregation, however, were not transformed. We also show a variant with automatic and manual SIMD application combined, which has a benefit for Q3 and Q9.

Unfortunately, these automatic SIMD optimizations do not yield any significant improvements in query runtime. Automatic vectorization alone hardly creates any gains but even introduces cases where the optimized code becomes slower. This means that even though primitives can be auto-vectorized, this is not yet a fire-and-forget solution.

5.4 Summary

We found with AVX-512 it is often straightforward to translate scalar code to data-parallel code, and observed performance gains of up to 8.4× in micro-benchmarks. However, for the more

Table 3: **Multi-Threaded Execution** – *TPC-H SF=100 on Skylake (10 cores, 20 hyper-threads)*

	Thr.	Typer ms	Typer speedup	TW ms	TW speedup	Ratio
Q1	1	4426	1.0	7871	1.0	0.56
Q1	10	496	8.9	867	9.1	0.57
Q1	20	466	9.5	708	11.1	0.66
Q6	1	1511	1.0	1443	1.0	1.05
Q6	10	243	6.2	213	6.8	1.14
Q6	20	236	6.4	196	7.4	1.20
Q3	1	9754	1.0	7627	1.0	1.28
Q3	10	1119	8.7	913	8.4	1.23
Q3	20	842	11.6	743	10.3	1.13
Q9	1	28086	1.0	20371	1.0	1.38
Q9	10	3047	9.2	2394	8.5	1.27
Q9	20	2525	11.1	2083	9.8	1.21
Q18	1	13620	1.0	18072	1.0	0.75
Q18	10	2099	6.5	2432	7.4	0.86
Q18	20	1955	7.0	2026	8.9	0.97

complicated TPC-H queries, the performance gains are quite small (around 10% for join queries). Fundamentally, this is because most OLAP queries are bound by data access, which does not (yet) benefit much from SIMD, and not by computation, which is the strength of SIMD. Coming back to the comparison between data-centric compilation and vectorization, we therefore argue that SIMD does not shift the balance in favor of vectorization much[7].

6. INTRA-QUERY PARALLELIZATION

Given the decade-long trends of stagnating single-threaded performance and growing number of CPU cores—Intel is selling 28 cores (56 hyper-threads) on a single Xeon chip—any modern query engine must make good use of all available cores. In the following, we discuss how to incorporate parallelism into the two query processing models.

6.1 Exchange vs. Morsel-Driven Parallelism

The original implementations of VectorWise and HyPer use different approaches. VectorWise uses exchange operators [3]. This classic approach [13] keeps its query processing operators like aggregation and join largely unaware of parallelism. HyPer, on the other hand, uses morsel-driven parallelism, in which joins and aggregations use shared hash-tables and are explicitly aware of parallelism. This allows HyPer to achieve better locality, load-balancing, and thus scalability, than VectorWise [22]. Using the 20 hyper-threads on our 10-core CPU, we measured an average speedup on the five TPC-H queries of 11.7× in HyPer, but only 7.2× in VectorWise. The parallelization framework is, however, orthogonal to the query processing model and we implemented morsel-driven parallelization in both Tectorwise and Typer, as it has been shown to scale better than exchange operators [22].

Morsel-driven parallelism was developed for HyPer [22] and can therefore be implemented quite straightforwardly in Typer: The table scan loop is replaced with a parallel loop and shared data structures like hash tables are appropriately synchronized similar to HyPer's implementation [22, 23].

For Tectorwise, it is less obvious how to use morsel-driven parallelism. The runtime system of Tectorwise creates an operator tree and exclusive resources for every worker. To achieve that the workers can work together on one query, every operator can have shared state. For each operator, a single instance of shared state is created. All workers have access to it and use it to communicate. For example, the shared state for a hash join contains the hash-table for the build side and all workers insert tuples into it. In general, the shared state of each operator is used to share results and coordinate work distribution. Additionally, pipeline breaking operators use a barrier to enforce a global order of sub-tasks. The hash join operator uses this barrier to enforce that first all workers consume the build side and insert results into a shared hash table. Only after that, the probe phase of the join can start. With shared state and a barrier, the Tectorwise implementation exhibits the same workload balancing parallelization behavior as Typer.

[7] We note that the benefit of SIMD can be larger when data is compressed [19] and on vector-oriented CPUs like Xeon Phi (see Section 7).

Table 4: **Hardware Platforms** – *used in experiments.*

	Intel Skylake	AMD T.ripper	Intel KNL
model	i9-7900X	1950X	Phi 7210
cores (SMT)	10 (x2)	16 (x2)	64 (x4)
issue width	4	4	2
SIMD [bit]	2×512	2×128	2×512
clock rate [GHz]	3.4-4.5	3.4-4.0	1.2-1.5
L1 cache	32 KB	32 KB	64 KB
L2 cache	1 MB	1 MB	1 MB
LLC	14 MB	32 MB	(16 GB)
list price [$]	989	1000	1881
launch	Q2'17	Q3'17	Q4'16
mem BW [GB/s]	58	56	68

6.2 Multi-Threaded Execution

We executed our TPC-H workload on scale factor 100 (ca. 100 GB of data). Table 3 shows runtimes and speedups in comparison with single-threaded execution. Using the 10 physical cores of our Skylake CPU, we see speedups between 8× and 9× for Q1, Q3, and Q9 in both systems. Given that modern CPUs reduce clock rates significantly when multiple threads are used these results are close to perfect scalability. For the scan query Q6 the speedup is limited by the available read memory bandwidth, and the large-scale aggregation of Q18 approaches the write bandwidth.

We also conducted these experiments on AWS EC2 machines and found that both systems scale equally well. However, we observe that when we use a larger EC2 instance to speed up query execution, the price per query rises. For example, the geometric mean over our TPC-H queries for a m5.2xlarge instance with 8 vCPUs is 0.0002$ per query (2027 ms runtime). On an instance with 48 cores it is 0.00034$ per query (534 ms runtime). So in this case, running queries 4× faster costs 1.7× more.

Tectorwise and Typer have similar scaling behavior. Nevertheless, the "Ratio" column of Table 3, which is the quotient of the runtimes of both systems, reveals an interesting effect: For all but one query, the performance gap between the two systems becomes smaller when all 20 hyper-threads are used[8]. For the join queries Q3 and Q9, the performance benefit of Tectorwise is cut in half, and Tectorwise comes closer to Typer for Q1 and Q18. This indicates that hyper-threading is effective at hiding some of the downsides of microarchitecturally sub-optimal code.

[8] Q6 is memory bound and is the only exception. Typer's branch-free selection implementation consumes more memory bandwidth resulting in 20% lower performance at high thread counts.

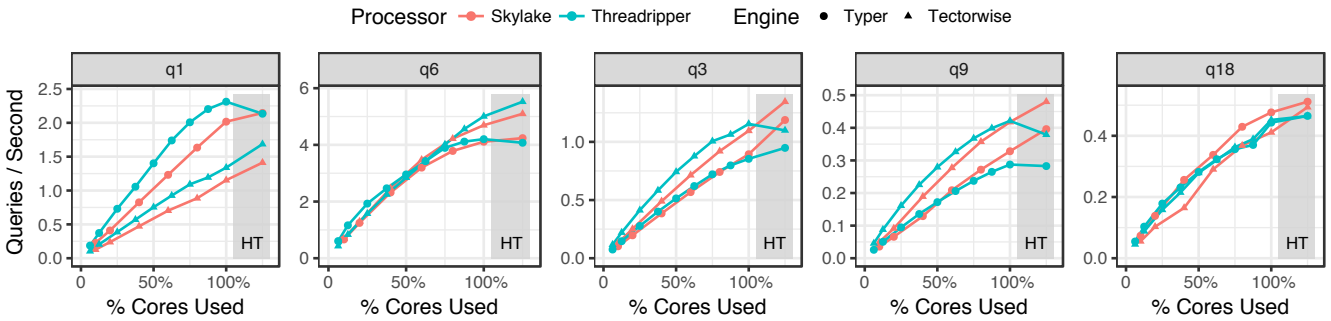


Figure 11: **Skylake vs. Threadripper.** – *SF=100*

6.3 Out-Of-Memory Experiments

Table 5: **SSD Results**

	Typer ms	TW ms	Ratio
Q1	923	1184	0.78
Q6	808	773	1.05
Q3	1405	1313	1.07
Q9	3268	2827	1.16
Q18	2747	2795	0.98

To compare Tectorwise and Typer at maximum speed, all measurements so far were in-memory (i.e., all table data was present in main memory). Large OLAP databases often exceed main memory capacity, which is why we also measured the impact of fetching the data from secondary storage. To do this, we stored the table data in a RAID 5 array of 3 SATA SSDs providing 1.4 GB/s read bandwidth instead of main memory, which has a bandwidth of 55 GB/s. Table 5 shows the runtimes with 20 threads on scale factor 100 when data is read from secondary storage. Comparing these with the in-memory results (c.f. Table 3), we can observe that the performance differences between the two query engines are slightly smaller but still noticeable ("Ratio" moves closer to one). Furthermore, as expected, the performance of the scan-dominated Q1 and Q6 are more affected by the slower bandwidth than the performance of the join and aggregation queries. Overall, we find that our in-memory analysis applies to out-of-memory settings with modern I/O devices.

7. HARDWARE

In previous experiments, we solely measured on Intel's latest microarchitecture Skylake. To find out whether our results also hold for other hardware platforms, we now also look at AMD with its recent Zen microarchitecture and Intel's Phi product line.

7.1 Intel Skylake X vs. AMD Threadripper

Table 4 shows the technical specifications for our Intel and AMD CPUs. Intel Skylake and AMD Threadripper cost almost the same, which directly allows comparing performance per dollar. Both systems also posses an almost equal memory bandwidth. However, the AMD Threadripper features 16 compute cores, clocked at maximally 4.0 GHz, while the Intel Skylake clocks at a higher rate of 4.5 GHz but contains only 10 cores. The differences between these processors is not coincidental but rather represents the design choices of the overall CPU product palettes of AMD and Intel. AMD offers more cores per dollar, but has only a quarter of computational SIMD throughput.

In terms of query processing performance our experiments show that both CPU models are roughly on par in absolute performance. Figure 11 shows the performance (in queries / second) for our experimental queries and systems both on Skylake and on Threadripper. As both processors have a different core counts the graphs are normalized on the x-axis to show which percentage of the available cores was used to achieve the runtime. Notably, the performance of both CPUs is very similar for Q6 and Q18 and the remaining queries are still quite similar (Q1 $< 20\%$, Q3 $< 25\%$, Q9 $< 40\%$). Also the relative performance of Typer and Tectorwise are quite similar. The join queries Q3 and Q9 and the selective scan in Q6 are processed faster by Tectorwise. Typer has an advantage on the computational query Q1. Overall, the performance characteristics two platforms are quite similar and the relative performance between the two hardware platforms is almost the same on both CPUs.

The only significant difference between the two platforms is that, although both platforms offer 2-way Simultaneous Multi-Threading (SMT), Intel's hyper-threading implementation seems to be much better. On the Skylake, we see a performance boost from hyper-threading for all queries. On the AMD system, the benefit of SMT is either very small, and for some queries the use of hyper-threads results in a performance degradation.

7.2 Knights Landing (Xeon Phi)

Despite being from two different hardware manufacturers, Skylake and Threadripper are quite similar. This cannot be said of the second generation of Intel's Xeon Phi product line. This microarchitecture is also called *Knights Landing* and is designed as a processor for high-performance computing (HPC). It is an integrated many-core architecture: There are 64 to 72 cores on each chip, but every core is relatively slow compared to Xeon cores. On the plus side, each core is equipped with two 512-bit vector processing units with an aggregate capacity of multiple TFLOPs. That makes it attractive for HPC applications.

From a database systems perspective, Knights Landing seems promising. Main memory can directly be accessed using six DDR4 memory channels (in contrast to GPUs data does not have to copied through PCIe). Each core features a 64 KB L1 cache and a 1 MB L2 cache that is shared with one neighboring core. Additionally, 16 GB of high-bandwidth memory, with a bandwidth of around 300 GB per second, is available. The available memory, number of cores, and the fact that many SIMD resources are available make the Knights Landing processor seem like a perfect OLAP machine.

Naturally, we want to explore this machine's qualities and see how Tectorwise and Typer perform in this scenario.

For our experiments on Knights Landing we configured the high-bandwidth memory as a hardware-managed L3 cache and expose all CPUs as one NUMA node (Quadrant Mode). A comparison of query processing performance of Knights Landing against Skylake is shown in Figure 12. Without any changes to the code we observe about the join queries Q3 and Q9 that Knights Landing's execution performance is from 0 to 25% higher than Skylake's. The relative performance of Tectorwise and Typer is similar on both CPUs.

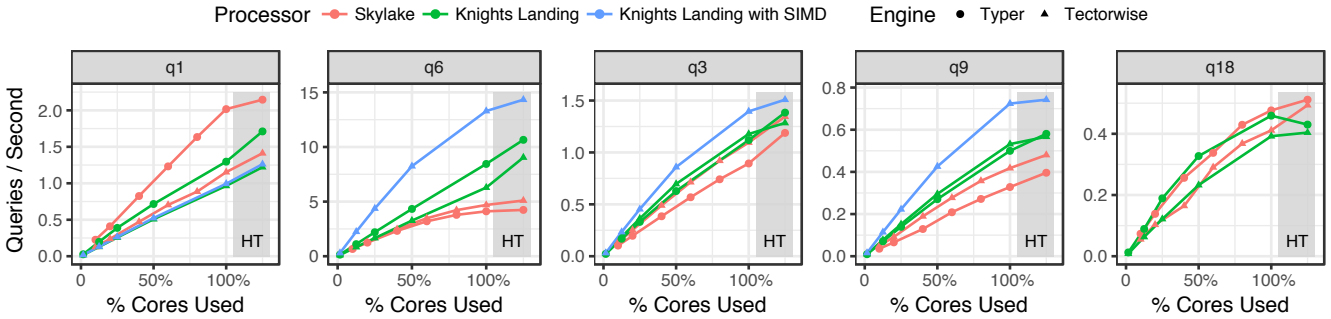


Figure 12: **Skylake vs. Knights Landing** – *SF=100.*

For query Q18 Knights Landing's performance is about 20% lower. On query Q1 it is about 30% lower. Finally, on Q6 Knights Landing is up to 2× faster. Recall that on Skylake query Q6 is bandwidth bound. Thus the extra 2 DDR4 channels of Knights Landing combined with the high-bandwidth memory as cache provide the required resources to get ahead of the Skylake processor. This measurement, however, must be seen in perspective: Each of our measurements is executed repeatedly. The cache of 16 GB, which can hold the entire working set of query Q6, boosts the performance unrealistically. In a real workload the cache would be shared with other queries which would likely evict much of query Q6's data. As a frame of reference one may use our measurement of query Q6 with the hardware configured not to have an L3 cache. In that case Knights Landing's Q6 performance is only 10% higher than the respective performance on Skylake. In a mixed workload one can expect the difference to be between these two. As a summary up to this point, with some queries being slightly faster and others slightly slower than Skylake, Knights Landings's performance seems not that great.

To be fair, this platform is designed for heavy use of SIMD instructions. Therefore, we need to take the measurements with manual SIMD optimizations into account. We observe that Knights Landing is able to execute a join query up to 50% faster than Skylake. On the selection query Q6 even a factor of almost 3× is achieved (although the same remark as for the scalar variant of Q6 applies). However, when taking a step back and looking at the whole performance picture, we also need to take the cost of each processor into account. Unfortunately, Knights Landing comes at almost twice the price of our Intel and AMD processors. Thus when the performance is broken down to execution speed per dollar, the commodity CPUs come out on top.

8. OTHER FACTORS

So far, we have focused on OLAP workloads and found only moderate performance differences between the two model—in particular, when properly parallelized. The performance differences are not large enough to make a general recommendation whether to use vectorization or compilation. Therefore, as a practical matter, other factors like OLTP performance or implementation effort, which we discuss in this section, may be of greater importance.

8.1 OLTP and Multi-Language Support

The vectorized execution model achieves efficiency when many vectors of values are processed, which is almost always the case in OLAP, but not in OLTP, where a query might only touch a single tuple. For OLTP workloads, vectorization has little benefit over traditional Volcano-style iteration. With compilation, in contrast, it is possible to compile all queries of a stored procedure into a single, efficient machine code fragment. This is a major benefit of compilation for OLTP and HTAP systems. Despite already having a modern vectorized engine (Apollo [21, 20]), the Microsoft SQL Server team felt compelled to additionally integrate the compilation-based engine Hekaton [12].

Compilation can also be highly beneficial for integrating user-defined functions and multiple languages into the same execution environment [11, 32, 31, 42].

8.2 Compilation Time

A disadvantage of code generation is the risk of compilation time dominating execution time [48]. This can be an issue in OLTP queries, though in transactional workloads it can be countered by relying on stored procedures, in which case code-generation can be done ahead of time. However, compilation time can also become large if the generated code is large because (optimizing) LLVM compile time is often super-linear to code size. OLAP queries that consist of many operators will generate large amounts of code, but also a small SQL query such as `SELECT * FROM T` can produce a lot of code if table `T` has thousands of columns, as each column leads to some code generation. Real-world data-centric compilation systems take mitigating measures against this. HyPer switches off certain LLVM optimization passes such as register allocation and replaces them by its own more scalable register allocation algorithm, and even contains a LLVM IR interpreter that is used to execute the first morsels of data; if that is enough to answer the query, full LLVM compilation is omitted [17]. This largely obviates this downside of compilation—but comes at the cost of additional system complexity. Spark falls back to interpreted tuple-at-a-time execution if a pipeline generates more than 8 KB Java byte code.

8.3 Profiling and Debuggability

A practical advantage of vectorized execution is that detailed profiling is possible without slowing down queries, since getting clock cycle counts for each primitive adds only marginal overhead, as each call to the function works on a thousand values. For data-centric compilation, it is hard to separate the contribution of the individual relational operators to the final execution time of a pipeline, though it could be done using sample-based code profiling, if the system can map back generated code lines to the relational operator in the query plan responsible for it. For this reason it is currently not possible in Spark SQL to know the individual contributions to execution time of relational operators, since the system can only measure performance on a per-pipeline basis.

8.4 Adaptivity

Adaptive query execution, for instance to re-order the evaluation order of conjunctive filter predicates or even joins is a technique

for improving robustness that can compensate for (inevitable) estimation errors in query optimization. Integrating adaptivity in compiled execution is very hard; the idea of adaptive execution works best in systems that interpret a query—in adaptive systems they can change the way they interpret it during runtime. Vectorized execution is interpreted, and thus amenable for adaptivity. The combination of fine-grained profiling and adaptivity allows VectorWise to make various *micro-adaptive* decisions [39].

We saw that VectorWise was faster than Tectorwise on TPC-H Q1 (see Table 1); this is due to an adaptive optimization in the former, similar to [15], that it not present in the latter. During aggregation, the system partitions the input tuples in multiple selection vectors; one for each group-by key. This task only succeeds if there are few groups in the current vector; if it fails the system exponentially backs off from trying this optimization in further vectors. If it succeeds, by iterating over all elements in a selection vector, i.e. all tuples of one group in the vector, hash-based aggregation is turned into ordered aggregation. Ordered aggregation then performs partial aggregate calculation, keeping e.g. the sum in a register which strongly reduces memory traffic, since updating aggregate values in a hash table for each tuple is no longer required. Rather, the aggregates are just updated once per vector.

8.5 Implementation Issues

Both models are non-trivial to implement. In vectorized execution the challenge is to separate the functionality of relational operators in control logic and primitives such that the primitives are responsible for the great majority of computation time (see Section 2.1). In compiled query execution, the database system is a compiler and consists of code that generates code, thus it is harder to comprehend and debug; especially if the generated code is quite low-level, such as LLVM IR. To make code generation maintainable and extensible, modern compilation systems introduce abstraction layers that simplify code generation [42] and make it portable to multiple backends [32, 31]. For HyPer, some of these abstractions have been discussed in the literature [29], while others have yet to be published.

It is worth mentioning that vectorized execution is at some disadvantage when sort-keys in order by or window functions are composite (consist of multiple columns). Such order-aware operations depend on comparison primitives, but primitives can only be specialized for a single type (in order to avoid code explosion). Therefore, such comparisons must be decomposed in multiple primitives, which requires a (boolean) vector as interface to these multiple primitives. This extra materialization costs performance. Compiled query execution can generate a full sort algorithm specifically specialized to the record format and sort keys at hand.

8.6 Summary

As a consequence of their architecture and code structure compilation and vectorization have distinct qualities that are not directly related to OLAP performance:

	OLTP	Lang. Support	Comp. Time	Profiling	Adaptivity	Implementation
Comp.	✓	✓	(✓)			Indirection
Vect.			✓	✓	✓	Constraints

On the one hand, compiled queries allow for fast OLTP stored procedures and seamlessly integrating different programming languages. Vectorization, on the other hand, offers very low query compile times, as primitives are precompiled: As a result of this structure, parts of a vectorized query can be swapped adaptively during runtime and profiling is easier. Finally, both systems have their own implementation challenges: Implementing operators with code generation introduces an additional indirection, whereas vectorization comes with a set of constraints on the code, which can be complicated to handle.

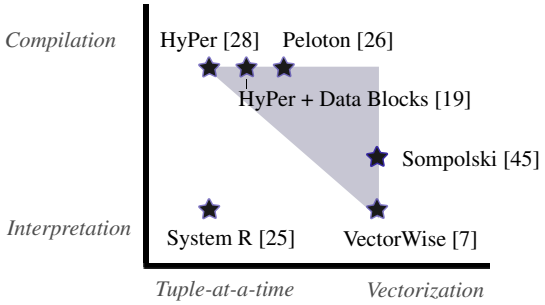


Figure 13: **Design space between vectorization and compilation** – *hybrid models integrate the advantages of the other approach.*

9. BEYOND BASIC VECTORIZATION AND DATA-CENTRIC CODE GENERATION

9.1 Hybrid Models

Vectorization and data-centric code generation are fundamentally different query processing models and the applied techniques are mostly orthogonal. That means that a design space, as visualized in Figure 13, exists between them. Many systems combine ideas from both paradigms in order to achieve the "best of both worlds". We first describe how vectorization can be used to improve the performance of the compilation-based systems HyPer and Peloton, before discussing how compilation can help vectorization.

In contrast to other operators in HyPer, scans of the compressed, columnar *Data Block* format [19] are implemented in template-heavy C++ using vectorization and without generating any code at runtime. Each attribute chunk (e.g., 2^{16} values) of a Data Block may use a different compression format (based on the data in that block). Using the basic data-centric compilation model, a scan would therefore have to generate code for all combinations of accessed attributes and compression formats—yielding exponential code size growth [19]. Besides compilation time, a second benefit of using vectorization-style processing in scans is that it allows utilizing SIMD instructions where it is most beneficial (Section 5). Since, Data Blocks is the default storage data format, HyPer (in contrast to Typer) may be considered a hybrid system that uses vectorization for base table selections and decompression, and data-centric code generation for all other operators.

By default, data-centric code generation fuses all operators of the same pipeline into a single code fragment. This is often beneficial for performance, as it avoids writing intermediate results to cache/memory by keeping the attributes of the current row in CPU registers as much as possible. However, there are also cases where it would be better to explicitly break a single pipeline into multiple fragments—for example, in order to better utilize out-of-order execution and prefetching during a hash join. This is the key insight behind Peloton's *relaxed operator fusion* [26] model, which selectively introduces explicit materialization boundaries in the generated code. By batching multiple tuples, Peloton can easily introduce SIMD and software prefetching instructions [26]. Consequently, Peloton's pipelines are shorter and their structure resembles vectorized code (see Figure 13). If the query optimizer's decision about whether to break up a pipeline is correct (which is non-trivial [24]), Peloton can be faster than both standard models.

Table 6: **Query Processing Models** – *and pioneering systems.*

System	Pipelining	Execution	Year
System R [25]	pull	interpretation	1974
PushPull [27]	push	interpretation	2001
MonetDB [9]	n/a	vectorization	1996
VectorWise [7]	pull	vectorization	2005
Virtuoso [8]	push	vectorization	2013
Hique [18]	n/a	compilation	2010
HyPer [28]	push	compilation	2011
Hekaton [12]	pull	compilation	2014

The two previous approaches use vectorization to improve an engine that is principally based on compilation. Conversely, compilation can also improve the performance of vectorized systems. Sompolski et al. [45], for example, observed that it is sometimes beneficial to fuse the loops of two (or more) VectorWise-style primitives into a single loop—saving materialization steps. This fusion step would require JIT compilation and result in a hybrid approach, thus moving it towards compilation-based systems in Figure 13. However, to the best of our knowledge, this idea has not (yet) been integrated into any system.

Tupleware is a data management system focused on UDFs, specifically a hybrid between data-centric and vectorized execution, and uses a cost model and UDF-analysis techniques to choose the execution method best suited to the task [11].

Apache Impala uses a form of compiled execution, which, in a different way, is also a hybrid with vectorized execution [49]. Rather than fusing relational operators together, they are kept apart, and interface with each other using vectors representing batches of tuples. The Impala query operators are C++ templates, parameterized by tuple-specific functions (data movement, record access, comparison expressions) in, for example, a join. Impala has default slow ADT implementations for these functions. During compilation, the generic ADT function calls are replaced with generated LLVM IR. The advantages of this approach is that (unit) testing, debugging and profiling can be integrated easily—whereas the disadvantage is that by lack of fusing operators into pipelines makes the Impala code less efficient.

9.2 Other Query Processing Models

Vectorization and data-centric compilation are the two state-of-the-art query processing paradigms used by most modern systems, and have largely superseded the traditional pull-based iterator model, which effectively is an interpreter. Nevertheless, there are also other (more or less common) models. Before discussing the strengths and weaknesses of these alternative approaches, we taxonomize them in Table 6. We classify query processing paradigms regarding (1) how/whether pipelining is implemented, and (2) how execution is performed. Pipelining can be implemented either using the pull (*next*) interface, the push (*produce/consume*) interface, or not at all (i.e., full materialization after each operator). Orthogonally to the pipelining dimension, we use the execution method (interpreted, vectorized, or compilation-based) as the second classification criterion. Thus, in total there are 9 configurations, and, as Table 6 shows, 8 of these have actually been used/proposed.

Since System R, most database systems avoided materializing intermediate results using pull-based iteration. The push model became prominent as a model for compilation, but has also been used in vectorized and interpreted engines. One advantage of the push model is that it enables DAG-structured query plans (as opposed to trees), i.e., an operator may push its output to more than one consumer [27]. Push-based execution also has advantages in distributed query processing with Exchange operators, which is one of the reasons it has been adopted by Virtuoso [8]. One downside of the push model is that it is slightly less flexible in terms of control flow: A merge-sort, for example, has to fully materialize one input relation. Some systems, mostly notably MonetDB [10], do not implement pipelining at all—and fully materialize intermediate results. This simplifies the implementation, but comes at the price of increased main memory bandwidth consumption.

In the last decade, compilation emerged as a viable alternative to interpretation and vectorization. As Table 6 shows, although compilation can be combined with all 3 pipelining approaches, the push model is most widespread as it tends to result in more efficient code [47]. One exception is Hekaton [12], which uses pull-based compilation. An advantage of pull-based compilation is that it automatically avoids exponential code size growth for operators that call consume more than once. With push-based compilation, an operator like full outer join that produces result tuples from two different places in the source code, must avoid inlining the consumer code twice by moving it into a separate function that is called twice.

10. SUMMARY

To our surprise, the performance of vectorized and data-centric compiled query execution is quite similar in OLAP workloads. In the following, we summarize some of our main findings:

< *Computation:* Data-centric compiled code is better at computationally-intensive queries, as it is able to keep data in registers and thus needs to execute fewer instructions.

> *Parallel data access:* Vectorized execution is slightly better in generating parallel cache misses, and thus has some advantage in memory-bound queries that access large hash-tables for aggregation or join.

= *SIMD* has lately been one of the prime mechanisms employed by hardware architects to increase CPU performance. In theory, vectorized query execution is in a better position to profit from that trend. In practice, we find that the benefits are small as most operations are dominated by memory access cost.

= *Parallelization:* With find that with morsel-driven parallelism both vectorized and compilation based-engines can scale very well on multi-core CPUs.

= *Hardware platforms:* We performed all experiments on Intel Skylake, Intel Knights Landing, and AMD Ryzen. The effects listed above occur on all of the machines and neither vectorization nor data-centric compilation dominates on any hardware platform.

Besides OLAP performance, other factors also play an important role. Compilation-based engines have advantages in

< *OLTP* as they can create fast stored procedures and

< *language support* as they can seamlessly integrate code written in different languages.

Vectorized engines have advantages in terms of

> *compile time* as primitives are pre-compiled,

> *profiling* as runtime can be attributed to primitives, and

> *adaptivity* as execution primitives can be swapped mid-flight.

Acknowledgements. This work was partially funded by the DFG grant KE401/22-1. We would like to thank Orri Erling for explaining Virtuoso's vectorized push model and the anonymous reviewers for their valuable feedback.

11. REFERENCES

[1] https://stackoverflow.com/questions/36932240/avx2-what-is-the-most-efficient-way-to-pack-left-based-on-a-mask, 2016.

[2] S. Agarwal, D. Liu, and R. Xin. Apache Spark as a compiler: Joining a billion rows per second on a laptop. "https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html", 2016.

[3] K. Anikiej. Multi-core parallelization of vectorized query execution. Master's thesis, University of Warsaw and VU University Amsterdam, 2010. http://homepages.cwi.nl/~boncz/msc/2010-KamilAnikijej.pdf.

[4] R. Appuswamy, A. Anadiotis, D. Porobic, M. Iman, and A. Ailamaki. Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. *PVLDB*, 11(2):121–134, 2017.

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[6] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, 2013.

[7] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[8] P. A. Boncz, O. Erling, and M. Pham. Advances in large-scale RDF data management. In *Linked Open Data - Creating Knowledge Out of Interlinked Data - Results of the LOD2 Project*, pages 21–44. Springer, 2014.

[9] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[10] P. A. Boncz, W. Quak, and M. L. Kersten. Monet and its geographic extensions: A novel approach to high performance GIS processing. In *EDBT*, pages 147–166, 1996.

[11] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik. An architecture for compiling UDF-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[12] C. Freedman, E. Ismert, and P. Larson. Compilation in the Microsoft SQL Server Hekaton engine. *IEEE Data Eng. Bull.*, 37(1):22–30, 2014.

[13] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[14] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[15] T. Gubner and P. Boncz. Exploring query compilation strategies for JIT, vectorization and SIMD. In *IMDM*, 2017.

[16] T. Kersten. https://github.com/TimoKersten/db-engine-paradigms, 2018.

[17] A. Kohn, V. Leis, and T. Neumann. Adaptive execution of compiled queries. In *ICDE*, 2018.

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

[19] H. Lang, T. Mühlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD*, pages 311–326, 2016.

[20] P. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL Server column stores. In *SIGMOD*, pages 1159–1168, 2013.

[21] P. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server column store indexes. In *SIGMOD*, pages 1177–1184, 2011.

[22] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[23] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann. Efficient processing of window functions in analytical SQL queries. *PVLDB*, 8(10):1058–1069, 2015.

[24] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *VLDB Journal*, 2018.

[25] R. A. Lorie. XRM - an extended (n-ary) relational memory. *IBM Research Report*, G320-2096, 1974.

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

[27] T. Neumann. *Efficient generation and execution of DAG-structured query graphs*. PhD thesis, University of Mannheim, 2005.

[28] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

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

[30] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, pages 567–574, 2001.

[31] S. Palkar, J. J. Thomas, D. Narayanan, A. Shanbhag, R. Palamuttam, H. Pirk, M. Schwarzkopf, S. P. Amarasinghe, S. Madden, and M. Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017.

[32] S. Palkar, J. J. Thomas, A. Shanbhag, M. Schwarzkopt, S. P. Amarasinghe, and M. Zaharia. A common runtime for high performance data analysis. In *CIDR*, 2017.

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

[34] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - A vector algebra for portable database performance on modern hardware. *PVLDB*, 9(14):1707–1718, 2016.

[35] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2005.

[36] O. Polychroniou and K. A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*, 2013.

[37] O. Polychroniou and K. A. Ross. Vectorized bloom filters for advanced SIMD processors. In *DaMoN*, 2014.

[38] O. Polychroniou and K. A. Ross. Efficient lightweight compression alongside fast scans. In *DaMoN*, 2015.

[39] B. Raducanu, P. Boncz, and M. Zukowski. Micro adaptivity in Vectorwise. In *SIGMOD*, pages 1231–1242, 2013.

[40] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[41] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976, 2016.

[42] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *SIGMOD*, pages 1907–1922, 2016.

[43] U. Sirin, P. Tözün, D. Porobic, and A. Ailamaki. Micro-architectural analysis of in-memory OLTP. In *SIGMOD*, pages 387–402, 2016.

[44] E. A. Sitaridi, O. Polychroniou, and K. A. Ross. SIMD-accelerated regular expression matching. In *DaMoN*, 2016.

[45] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, pages 33–40, 2011.

[46] D. Song and S. Chen. Exploiting SIMD for complex numerical predicates. In *ICDE*, pages 143–149, 2016.

[47] R. Y. Tahboub, G. M. Essertel, and T. Rompf. How to architect a query compiler, revisited. In *SIGMOD*, pages 307–322, 2018.

[48] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. In *DBTest*, 2018.

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

[50] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[51] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[52] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, University of Amsterdam, 2009.

[53] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.