Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last

Prashanth Menon
Carnegie Mellon University
pmenon@cs.cmu.edu

Todd C. Mowry
Carnegie Mellon University
tcm@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

In-memory database management systems (DBMSs) are a key component of modern on-line analytic processing (OLAP) applications, since they provide low-latency access to large volumes of data. Because disk accesses are no longer the principle bottleneck in such systems, the focus in designing query execution engines has shifted to optimizing CPU performance. Recent systems have revived an older technique of using just-in-time (JIT) compilation to execute queries as native code instead of interpreting a plan. The state-of-the-art in query compilation is to fuse operators together in a query plan to minimize materialization overhead by passing tuples efficiently between operators. Our empirical analysis shows, however, that more tactful materialization yields better performance.

We present a query processing model called "relaxed operator fusion" that allows the DBMS to introduce staging points in the query plan where intermediate results are temporarily materialized. This allows the DBMS to take advantage of inter-tuple parallelism inherent in the plan using a combination of prefetching and SIMD vectorization to support faster query execution on data sets that exceed the size of CPU-level caches. Our evaluation shows that our approach reduces the execution time of OLAP queries by up to 2.2× and achieves up to 1.8× better performance compared to other in-memory DBMSs.

PVLDB Reference Format:
Prashanth Menon, Todd C. Mowry, Andrew Pavlo. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *PVLDB*, 11(1): 1 - 13, 2017.
DOI: https://doi.org/10.14778/3136610.3136611

1. INTRODUCTION

As DRAM becomes increasingly cost-effective, it enables greater numbers of DBMS applications to become memory-resident. Given this trend, we anticipate that most future OLAP applications will use in-memory DBMSs. Because in-memory DBMSs are designed such that the bulk of the working dataset fits in memory, disk accesses are no longer the main bottleneck in query execution. Instead, cache misses to memory and computational throughput are much more important factors in the performance of in-memory DBMSs.

Proceedings of the VLDB Endowment, Vol. 11, No. 1

DOI: https://doi.org/10.14778/3136610.3136611

Modern CPUs support instructions that help on both of these fronts: (1) software *prefetch* instructions can move blocks of data from memory into the CPU caches before they are needed, thereby hiding the latency of expensive cache misses [14, 22]; and (2) *SIMD* instructions can exploit vector-style data parallelism to boost computational throughput [33]. Although both software prefetching and SIMD have been studied in the past (in isolation) for specific DBMS operators, our focus in this paper is how to successfully *combine* both techniques across *entire* query plans.

A key challenge for both software prefetching and SIMD vectorization is that neither technique works well in a tuple-at-a-time model (i.e., Volcano [19]). In order to successfully hide cache miss latency with prefetching, the software must prefetch a number of tuples ahead (to overlap the cache miss with the processing of other tuples). To bundle together SIMD-width vectors for SIMD processing, the software needs to extract data parallelism across chunks of tuples at a time. While both software prefetching and SIMD vectorization require the ability to look across multiple tuples at a time, there are key differences and subtleties in how they interact with each other. For example, SIMD vector instructions require that data be packed together contiguously, whereas prefetching needs to generate a set of addresses (to be prefetched) ahead of time, but it does not require either those addresses (or the data blocks that they point to) to be arranged contiguously. In addition, since the relative sparsity of the data that is being processed tends to increase in the higher levels of a query plan tree, this also changes relative trade-offs between software prefetching and SIMD vectorization.

Until now, no DBMS has taken a holistic view on how to use all of the above techniques effectively together throughout the entire query plan. Most systems that support some form of query compilation do not use the vectorized query processing model or only compile a portion of the query plan (e.g., predicates). Likewise, no system employs data-level parallelism optimizations (i.e., SIMD) or explicit prefetching. Part of this reason is that the hardware advancements needed to support SIMD (e.g., wide registers) and software prefetching were introduced in the last five years.

Given this, our focus is on getting prefetching and vectorization to work together efficiently using a new processing model called **relaxed operator fusion** (ROF). ROF enables the DBMS to structure the generated query code such that portions are combined together to take advantage of SIMD vectorization and software-level prefetching. To evaluate our approach, we implemented our ROF model in the **Peloton** in-memory DBMS [5] and measured its efficacy with the TPC-H benchmark [40]. Our experimental results show that ROF improves the performance of the DBMS for OLAP queries by up to 2.2×. We also compare two other in-memory OLAP systems (HyPer [20], Actian Vector [1]) and show that ROF's combination

```
SELECT SUM(...) AS revenue
  FROM LineItem JOIN Part ON l_partkey = p_partkey
 WHERE (CLAUSE1) OR (CLAUSE2) OR (CLAUSE3)
```

Figure 1: TPC-H Q19 – An abbreviated version of TPC-H's Q19. The three clauses are a sequence of conjunctive predicates on attributes in both the `LineItem` and `Part` tables.

of query compilation with SIMD and prefetching achieves up to 1.8× the performance than the two techniques by themselves.

The remainder of this paper is structured as follows. Sect. 2 begins with a discussion of query plan compilation, vectorization, and prefetching on modern CPU architectures. Next, in Sect. 4, we describe our relaxed operator fusion technique for efficient query execution. In Sect. 5, we present our experimental evaluation. Lastly, we conclude with related work in Sect. 6.

2. BACKGROUND

We first provide an overview of the fundamental concepts of query compilation, vectorized query processing, and prefetching. It is only through an intelligent orchestration of these methods that make it possible to use them together. To aid in our discussion, we use the Q19 query from the TPC-H benchmark [40]. A simplified version of this query's SQL is shown in Fig. 1. We omit the query's `WHERE` clauses for simplicity; they are not important for our exposition.

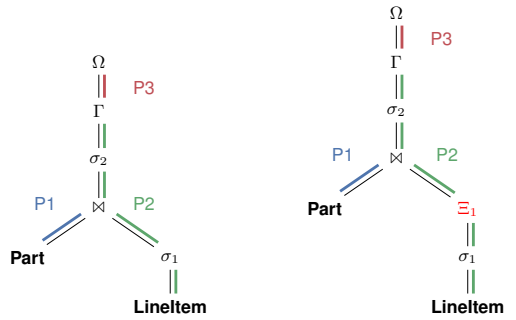
2.1 Query Compilation

There are multiple ways to compile queries in a DBMS. One method is to generate C/C++ code that is then compiled to native code using an external compiler (e.g., `gcc`). Another method is to generate an intermediate representation (IR) that is compiled into machine code by a runtime engine (e.g., `LLVM`). Lastly, staging-based approaches exist wherein a query engine is partially evaluated to automatically generate specialized C/C++ code that is compiled using an external compiler [21].

In addition to the variations in how the DBMS compiles a query into native code, there are several techniques for organizing this code [30, 41, 37, 31]. One naïve way is to create a separate routine per operator. Operators then invoke the routines of their children operators to pull up the next data (e.g., tuples) to process. Although this is faster than interpretation, the CPU can still incur expensive branch mispredictions because the code jumps between routines [30].

A better approach (used in HyPer [30] and MemSQL [32]) is to employ a push-based model that reduces the number of function calls and streamlines execution. To do this, the DBMS's optimizer first identifies the *pipeline breakers* in the query's plan. A pipeline breaker is any operator that explicitly spills any or all tuples out of CPU registers into memory. In a push-based model, child operators produce and *push* tuples to their parents, requesting them to consume the tuples. A tuple's attributes are loaded into CPU registers and flow along the pipeline's operators from the start of one breaker to the start of the next breaker, at which point it must be materialized. Any operator between two pipeline breakers operate on tuples whose contents are in CPU registers, thus improving data locality.

To illustrate this approach, consider the plan shown in Fig. 2a for the TPC-H query in Fig. 1 annotated with its three pipelines **P1**, **P2**, and **P3**. In this work, we use Ω to denote consumption of the plan's output. The DBMS's optimizer generates one loop for every pipeline. Each loop iteration begins by reading a tuple from either a base table or an intermediate data structure (e.g., a hash-table used for a join). The routine then processes the tuple through all operators in the pipeline, storing a (potentially modified) version of the tuple into the next materialized state for use in the next pipeline. Combining the execution of multiple operators within



(a) Query Plan with Pipelines **(b)** Query Plan using ROF

P1:
```
HashTable ht; // Join Hash-Table

// Scan Part table, P
for (auto &part : P.GetTuples()) {
  ht.Insert(part);
}
```

P2:
```
// Running Aggregate
Aggregator agg;

// Scan LineItem table, L
for (auto &lineitem : L.GetTuples()) {
  if (PassesPredicate1(lineitem)) {
    auto &part = ht.Probe(lineitem);
    if (PassesPredicate2(lineitem, part)) {
      agg.Aggregate(lineitem, part);
} } }
```

P3:
```
return agg.GetRevenue();
```

(c) Generated Code

Figure 2: Query Compilation Example – The plan for TPC-H Q19 (Fig. 2a) and its corresponding generated code (Fig. 2c). The plan shown in Fig. 2b uses our relaxed operator fusion model.

a single loop iteration is known as *operator fusion*. Fusing together pipeline operators in this manner obviates the need to materialize tuple data to memory, and can instead pass tuple attributes through CPU registers, or the stack which will likely be in the cache.

Returning to our example, Fig. 2c shows the colored blocks of code that correspond to the identically-colored pipeline in the query execution plan in Fig. 2a. The first block of code (**P1**) performs a scan on the `Part` table and the build-phase of the join. The second block (**P2**) scans the `LineItem` table, performs the join with `Part`, and computes the aggregate function. The code fragments demonstrate that pipelined operations execute entirely using CPU registers and only access memory to retrieve new tuples or to materialize results at the pipeline breakers.

2.2 Vectorized Processing

The code in Fig. 2c achieves better performance than interpreted query plans because it executes fewer CPU instructions, including costly branches and function calls. But it processes data in a tuple-at-a-time manner, which makes it difficult for the DBMS to employ optimizations that operate on multiple tuples at a time [6].

Generating multiple tuples per iteration has been explored in previous systems. MonetDB's bulk processing model materializes the entire output of each operator to reduce the number of function calls in the system. The drawback of this approach is that this is bad for cache locality unless the system stores tables in a columnar format and each query's predicates are selective [43].

Instead of materializing the entire output per operator, the X100 project [11] for MonetDB that formed the basis of VectorWise (now Actian Vector) generated a vector of results (typically 100–

P1:
```
HashTable ht; // Join Hash-Table

// Scan Part table, P, by vectors
for (auto &block : P.GetBlocks()) {
  ht.Insert(block);
}
```

P2:
```
// Running Aggregate
Aggregator agg;

// Scan LineItem table, L
for (auto &block : L.GetBlocks()) {
  auto &sel_1 = PassesPredicate1(block);
  auto &result = ht.Probe(block, sel_1);
  auto &part_block =
    Reconstruct(P, result.LeftMatches());
  auto &sel_2 =
    PassesPredicate2(block, part_block,
                     result);
  agg.Aggregate(block, part_block, sel_2);
}
```

P3:
```
return agg.GetRevenue();
```

Figure 3: Vectorization Example – An execution plan for TPC-H Q19 from Fig. 1 that uses the vectorized processing model.

10k tuples). This approach is also used in the Snowflake [17] and Tupleware [16] systems. Modern CPUs are inherently well-suited to vectorized processing. Since loops are tight and iterations are often independent, out-of-order CPUs can execute multiple iterations concurrently, fully leveraging its deep pipelines. A vectorized engine relies on the compiler to automatically detect loops that can be converted to SIMD, but modern compilers are only able to optimize simple loops involving computation over numeric columns. Only recently has there been work demonstrating how to manually optimize more complex operations with SIMD [33, 34].

Fig. 3 shows pseudo-code for TPC-H Q19 using the vectorized processing model. First, **P1** is almost identical to its counterpart in Fig. 2c, except tuples are read from `Part` in blocks. Moreover, since vectorized DBMSs employ late materialization, only the join-key attribute and the corresponding tuple ID is stored in the hash-table. **P2** reads blocks of tuples from `LineItem` and passes them through the predicate. In contrast to tuple-at-a-time-processing, the `PassesPredicate1` function is applied to all tuples in the block in one iteration All of the predicate filter functions produce an array of the positions of the tuples in the block that pass the predicate (i.e., *selection vector*). This vector is woven through each call to retain only the valid tuples. The system then probes the hash-table with the input block and the selection vector to find join match candidates. It uses this result to reconstruct the block of tuples from the `Part` table. The penultimate operation (filter) uses both blocks and the selection vector from the join to generate the final list of valid tuples.

Vectorized processing leverages both modern compilers and modern CPUs to achieve good performance. It also enables the use of explicit SIMD vectorization; however, most DBMSs do not employ SIMD vectorization throughout an entire query plan. Many systems instead only use SIMD in limited ways, such as for internal sub-tasks (e.g., checksums). Others have shown how to use SIMD for all of the operators in a relational DBMS but they make a major assumption that the data set is small enough to fit in the CPU caches [33], which is usually not possible for real applications. But this means that if the data set does not fit in the CPU caches, then the processor will stall because of memory accesses and then the benefit of vectorization will be minimal.

2.3 Prefetching

The idea behind prefetching is that the system anticipates DBMS memory accesses and pro-actively move data from memory into CPU caches before it is needed, thereby avoiding a cache miss when the data is subsequently accessed. Prefetches can be launched either by hardware predictors or through explicit software prefetch instructions. For the former, modern CPUs can detect and prefetch regular strided access patterns. But while such hardware prefetchers are useful when the DBMS sequentially scans a single column, they are unable to prefetch data under irregular access patterns (e.g., hash-table or index probing). Today's compilers can also automatically insert software prefetch instructions into code for array-based applications [29] and others have studied algorithms for prefetching in pointer-based applications [27]. Like hardware prefetchers, however, modern compilers are unable to detect and handle the frequent irregular patterns found in DBMS operations.

Using software prefetching to optimize DBMS operators has also been studied in the past [14, 22]. A common theme in these approaches is to transform the primary input-processing loop from one consisting of N dependent memory accesses per-tuple into a loop with N+1 code steps, where each step consumes data prefetched during the previous step and launches new prefetches for data required in the next step. *Group prefetching* (GP) [14] works by processing input tuples in groups of size G. Tuples in the same group proceed through code steps in lock-step, thereby ensuring at most G simultaneous independent in-flight memory accesses.*Software-pipelined prefetching* (SPP) works by processing different tuples in each code step, thereby creating a fluid pipeline. *Asynchronous memory-access chaining* (AMAC) [22] works by having a window of tuples in different steps of processing, swapping out tuples as soon as they complete to enable early stopping.

Software prefetch instructions were added to the ISAs of the major commercial CPUs starting roughly 20 years ago. While CPUs initially supported only a limited number of simultaneous in-flight prefetches, it has increased over time. It is still the case, however, that a DBMS needs to be careful not to launch too many prefetches in a short period of time, because the CPU is likely to drop them once the hardware buffer that tracks outstanding requests fills up.

3. PROBLEM OVERVIEW

In the previous section, we described how query compilation, vectorized processing, and prefetching can optimize query execution for in-memory DBMSs. A natural question to ask is whether it is possible to combine these techniques into a single query processing model that embodies the benefits of them all?

To the best of our knowledge, no DBMS has successfully combined all techniques into a single query processing engine. Part of the reason is that most systems that employ query compilation generate tuple-at-a-time code that avoids tuple materialization. As we discussed above, however, both vectorization and prefetching requires a vector of input tuples to successfully exploit data-level parallelism. Recent work has explored implementing SIMD scans in a query compiling DBMS, but it requires reverting to interpreted scans that feed its results into compiled code tuple-at-a-time [25] or is unable to stage data suitably to use prefetching [15]. DBMSs based on vectorized processing rely on out-of-order CPUs to exploit data-level parallelism, but this often is not possible for complex operators, such as hash-joins or index probes. Hence, we contend that what is needed is a hybrid model that is able to tactfully materialize state to support both tuple-at-a-time processing, vectorized processing, and prefetching.

To help demonstrate this point, we execute a microbenchmark that performs a hash-join between two tables, each containing a single 32-bit integer column (as in [33]). We implemented three different approaches: (1) a scalar tuple-at-a-time join, (2) a SIMD join using the vertical vectorization technique described in [33], and (3) a tuple-at-a-time join modified to use group-prefetching [14].

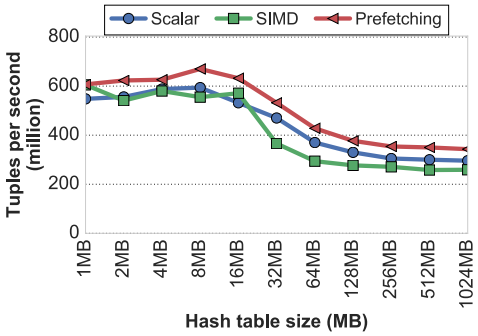


Figure 4: Microbenchmark – Effect of hash-table size on join performance

We measure the overall throughput in output tuples per-second as we vary the size of the join's hash-table from 1 MB to 1024 MB. We keep the size of the probe table (A) fixed to 100m tuples, totaling ∼382 MB of raw data. We vary the size of the build table (B) such that it produces a hash-table of the desired size in the experiment. The hash-table uses open-addressing with linear probing, a 50% fill-factor, and the 32-bit finalizer from MurmurHash3 [9] as the hash function. We choose this to simulate real-world implementations and because it requires minimal computation (three bit-shifts, two multiplications, and three XORs). Each hash-table bucket stores a 4-byte key and a 4-byte payload, and the hash-table is organized as an array-of-structs. The values in both tables are uniformly distributed, and each tuple in A matches with at most one tuple in B. The join produces a combined table with 100m tuples with a total size of ∼381 MB. We executed our test on 20 hardware threads (hyper-threading is disabled) with 25 MB of shared L3 CPU cache. We defer the full description of our environment until Sect. 5.

From the results shown in Fig. 4, we see that SIMD probes utilizing the vertical vectorization technique performs worse than tuple-at-a-time probes with prefetching, even when the hash-table is cache-resident. This is because vectorization requires recomputing hash values on hash and key collisions. The second observation is that both tuple-at-a-time techniques with and without prefetching outperform the SIMD version when the hash-table does not fit in cache. This is because the join shifts from a compute-bound operation to a memory-bound operation for which SIMD does not help. Lastly, we see that tuple-at-a-time processing with prefetching is consistently better than all approaches, often by up to 1.2×.

The main takeaway from the microbenchmark results is that tuple-at-a-time processing with prefetching outperforms SIMD for hash-joins regardless of hash-table sizes. Prefetching requires looking across a vector of tuples to exploit inter-tuple parallelism, but fully pipelined compiled plans avoid any materialization. Moreover, data becomes more sparse and memory accesses become more random as we move up the query plan tree, making prefetching that much more important. At the leaves of the tree, the DBMS can rely on the hardware prefetcher; this is not true higher up. Neither wholly vectorized nor wholly compiled query plans are optimal. What is needed is the ability for the DBMS to tactfully materialize tuples through prefetching at various points in the query plan — to enable vectorization and exploit inter-tuple parallelism — and otherwise fuse operators to ensure efficient pipelining.

4. RELAXED OPERATOR FUSION

The primary goal of operator fusion is to minimize materialization. We contend that strategic materialization can be advantageous as it can exploit inter-tuple parallelism inherent in query execution. Tuple-at-a-time processing by its nature exposes no inter-tuple parallelism. Thus, to facilitate strategic materialization, one could *relax* the requirement that operators within a pipeline be fused together. With this, the DBMS instead decomposes pipelines into *stages*. A stage is a partition of a pipeline in which all operators are fused together. Stages within a pipeline communicate solely through cache-resident vectors of tuple IDs. Tuples are processed sequentially through operators in any given stage one-at-a-time. If the tuple is valid in the stage, its ID is appended into the stage's output vector. Processing remains within a stage while the stage's output vector is not full. If and when this vector reaches capacity, processing shifts to the next stage in the pipeline, where the output vector of the previous stage serves as the input to the current. Since there is always exactly one active processing stage in ROF, we ensure both input and output vectors (when sufficiently small) will remain in the CPU's caches.

ROF is a hybrid between pipelined tuple-at-a-time processing and vectorized processing. There are two key distinguishing characteristics between ROF and traditional vectorized processing; with the exception of the last stage iteration, ROF stages always deliver a full vector of input to the next stage in the pipeline, unlike vectorized processing that may deliver input vectors restricted by a selection vector. Secondly, ROF enables vectorization across *multiple* sequential relational operators (i.e., a stage), whereas conventional vectorization operates on a single relational operator, and often times *within* relational operators (e.g., vectorized hash computation followed by a vectorized hash-table lookup).

To help illustrate ROF's staging, we first walk through an example. We then describe how to implement ROF in an in-memory DBMS.

4.1 Example

Returning again to our TPC-H Q19 example, Fig. 2b shows a modified query plan using our ROF technique to introduce a single stage boundary after the first predicate (σ_1). The Ξ operator denotes an output vector that represents the boundary between stages. The code generated for this modified query plan is shown in Fig. 5. In the first stage (lines 13–20), tuples are read from the `LineItem` table and passed through the filter to determine their validity in the query. If a tuple passes through the filter (σ_1), then its ID is appended to the stage's output vector (Ξ). When this vector reaches capacity, or when the scan operator has exhausted tuples in `LineItem`, the vector is delivered to the next stage.

The next stage in the pipeline (lines 22–30) uses this vector to read valid `LineItem` tuples for probing the hash-table and finding matches. If a match exists, both components are passed through the secondary predicate (σ_2) to again check the validity of the tuple in the query. If it passes this predicate, it is aggregated as part of the final aggregation operator.

We first note that one loop is still generated per-pipeline (lines 11–31). A pipeline loop contains the logic for all stages contained in the pipeline. To facilitate this, the DBMS splits pipeline loops into multiple inner-loops, one for each stage in the pipeline. In this example, lines 13–20 and 22–30 are for the first and second stages, respectively. The DBMS fuses together the code for the operators within a stage loop. This is seen in Fig. 5 as line 26 corresponds to the probe, line 27 to the second predicate, and line 28 to the final aggregation. In general, there are the same number of inner-loops per pipeline loop as there are stages, and the number of stage output vectors (Ξ) is equal to one less than the number of stages.

Lastly, the code maintains both a read and write position for each output vector. The write position tracks the number of tuples in the vector; the read position tracks how far into the vector a given stage has read. A stage has exhausted its input when either (1) the read position has surpassed the amount of data in the materialized state (i.e.,

```
1  #define VECTOR_SIZE 256
2
3  HashTable join_table;  // Join Operator Table
4  Aggregator aggregator; // Running Aggregator
5
6  int buf[VECTOR_SIZE] = {0}; // Stage Vector
7  int buf_idx = 0; // Stage Vector Offset
8  oid tid = 0; // Tuple ID
9
10 // Pipeline P2
11 while (tid < L.GetNumTuples()) {
12   // Stage #1: Scan LineItem, L
13   for (buf_idx = 0;
14        tid < L.GetNumTuples(); tid++) {
15     auto &lineitem = L.GetTuple(tid);
16     if (PassesPredicate1(lineitem)) {
17       buf[buf_idx++] = tid;
18       if (buf_idx == VECTOR_SIZE) break;
19     }
20   }
21   // Stage #2: Probe, filter, aggregate
22   for (int read_idx = 0;
23        read_idx < buf_idx; read_idx++) {
24     auto &lineitem =
25       L.GetTuple(buf[read_idx]));
26     auto &part = join_table.Probe(lineitem);
27     if (PassesPredicate2(lineitem, part)) {
28       aggregator.Aggregate(lineitem, part);
29 } } }
```

Figure 5: ROF Staged Pipeline Code Routine – The example of the pseudo-code generated for pipeline P2 in the query plan shown in Fig. 2b. In the first stage, the code fills the stage buffer with tuples from table LineItem that satisfy the scan predicate. Then in the second stage, the routine probes the join table, filters the results of the join, and aggregates the filtered results.

data table or hash-table) or (2) the read and write index are equal for the input vector. Therefore, a pipeline is complete only when all of its constituent stages are finished. If a stage accesses external data structures that are needed in subsequent stages, ROF requires a companion output vector that stores data positions/pointers that it is aligned with the primary TID output vector.

Our ROF technique is flexible enough to model both tuple-at-a-time processing and vectorized processing, and hence, subsumes both models. The former can be realized by creating exactly one stage per pipeline. Since a stage fuses all operators it contains and every pipeline has only one stage, pipeline loops contain no intermediate output vectors. Vectorized processing can be modeled by installing a stage boundary between pairs of operators in a pipeline.

Staging alone does not provide many benefits; however, it facilities two optimizations not possible otherwise: SIMD vectorization, and prefetching of non-cache-resident data.

4.2 Vectorization

SIMD processing is generally impossible when executing a query tuple-at-a-time. Our ROF technique enables operators to use SIMD by introducing a stage boundary on their input, thereby feeding them a vector of tuples. The question now becomes whether to also impose a stage boundary on the output of a SIMD operator. With a boundary, SIMD operators can efficiently issue selective stores of valid tuple IDs into their output vectors. With no boundary, the results of the operator are pipelined through all operators that follow in the stage. This can be done using one of two methods. Both methods assume the result of a SIMD operator resides in a SIMD register. In the first method, the operator breaks out of SIMD code to iterate over the results in the individual SIMD lanes one-at-a-time [12, 42]. Each result is pipelined to the next operator in the stage. In the second method, rather than iterate over individual lanes, the operator delivers its results in a SIMD register to the next operator in the stage. Both methods are not ideal. Breaking out of SIMD code unnecessarily ties up the registers for the duration of the

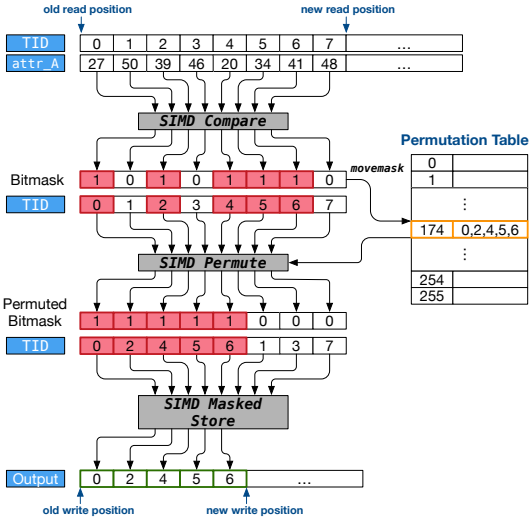


Figure 6: SIMD Predicate Evaluation Example – An illustration of how to use SIMD to evaluate the predicate for TPC-H Q19.

stage. Delivering the entire register risks under-utilization if not all input tuples pass the operator, resulting in unnecessary computation.

Given this, we greedily force a stage boundary on all SIMD operator outputs. The advantages of this are (1) SIMD operators always deliver a 100% full vector of only valid tuples, (2) it frees subsequent operators from performing validity checks, and (3) the DBMS can generate tight loops that are exclusively SIMD. We now describe how to implement a SIMD scan using these boundaries.

Implementation: In contrast to scalar selection where the result of applying a predicate against a tuple is a single boolean value indicating the validity of the tuple, the result of a SIMD application of a predicate is a bit-mask. Processing n elements in parallel produces a bit-mask stored in a SIMD register where all bits of each of the n elements are either 0 or 1 (to indicate the validity of the associated tuple). To determine which tuples are valid using the bit-mask, the DBMS could employ partial vectorization and iterate over the bits in the mask to extract one bit at a time.

ROF uses a different approach that is wholly in SIMD code. The technique leverages a precomputed, cache-resident index to lookup permutation masks that are used to shuffle SIMD elements into valid and invalid components. The illustration in Fig. 6 is performing a SIMD scan over a 4-byte integer column attr_A and evaluating the predicate attr_A $<$ 44. The DBMS first loads as many attribute values as possible (along with their tuple IDs) into the widest available SIMD register. Next, it applies the predicate to produce a bit-mask. In the example, the tuples with IDs (1, 3, 7) fail to pass the predicate. To correctly write out only the valid IDs, the DBMS shuffles the tuple ID SIMD register so that the valid and invalid tuple IDs are stored contiguously, effectively partitioning the register. To achieve this, the DBMS invokes the movemask instruction to convert the bit-mask into an integer number that it uses as an index into a *permutation table*. This is a precomputed table that maps a given bit-mask value to a 8-byte bit-mask that corresponds to the correct re-arrangement of elements in the SIMD register to partition it into valid and invalid parts. In the example, the bit-mask's value is 174, which corresponds to the bit-mask (0,2,4,5,6). Applying this permutation bit-mask moves elements in positions 0, 2, 4, 5, and 6 to the first five elements in the register. We apply this permutation to both the original bit-mask and the tuple ID counter. The DBMS then writes the modified tuple ID counter to

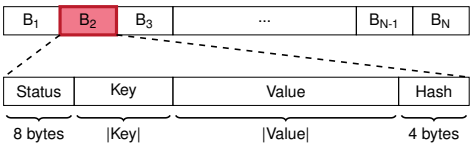


Figure 7: Hash-Table Data Structure – An overview of the DBMS's open-addressing hash-table used for joins (with B_N buckets).

the output vector at the current write position using a masked store with the modified bit-mask as the selection mask. The DBMS then increments the new write position by the number of valid tuples (using the `popcnt` instruction), loads a new vector of values, and increments the tuple ID vector by eight.

The permutation table stores an 8-byte value for each possible input bit-mask. A SIMD register storing n elements can produce 2^n masks. With AVX2 256-bit registers operating on eight 4-byte integers, this results in $2^8 = 256$ possible bit-masks. Thus, the size of the largest permutation table is at most $2^8 \times 8 = 2$ KB, small enough to fit in the CPU's L1 cache.

4.3 Prefetching

Aside from the regular patterns of sequential scans, the more complex memory accesses within a DBMS are beyond the scope of what today's hardware or commercial compilers can handle. Therefore, we propose new compiler passes for automatically inserting prefetch instructions that can handle the important irregular, data-dependent memory accesses of an OLAP DBMS.

The DBMS must prefetch sufficiently far in advance to hide the memory latency (overlapping it with useful computation), while at the same time avoiding the overhead of unnecessary prefetches [29]. Limiting the scope of prefetching to within the processing of a single tuple in a pipeline is inefficient because by the time the DBMS knows how far up the query plan the tuple will go, there is not sufficient computation to hide the memory latency. On the other hand, aggressively prefetching all of the data needed within a pipeline can also hurt performance due to cache pollution and wasted instructions. These challenges are exacerbated as the complexity within a pipeline increases, since it becomes increasingly difficult to predict data addresses in advance.

Our ROF model avoids all of these problems. The DBMS installs a *stage boundary* at the input to any operator that requires random access to data structures that are larger than the cache. This ensures that prefetch-enabled operators receive a full vector of input tuples, enabling it to overlap computation and memory accesses since these tuples can be processed independently. As we will show in Sect. 5, hash-joins and hash-based aggregations are two classes of important operators that yield significant improvements with prefetching.

Implementation: To help ground our discussion of how to implement prefetching in ROF, we briefly discuss the design of the hash-table used in both hash-joins and hash-based aggregations, found in Fig. 7. We use an open-addressing hash-table design with linear probing for hash and key-collisions. Previous work has shown this design to be robust and cache-friendly [35]. We use MurmurHash3 [9] as our primary hash function. This is differs from previous work [33] that prefers to use computationally simpler (and therefore faster) hash functions, such as `multiply-add-shift`. We want to use a general-purpose hash function that can (1) work on multiple different non-integer data types, (2) provide a diverse hash distribution, and (3) execute fast. MurmurHash3 satisfies these requirements and used in many popular systems [4, 2, 3].

Our hash-table, shown in Fig. 7, is laid out as a contiguous array of buckets. Buckets begin with an 8-byte `status` field that indicates (1) if this bucket is empty, (2) if it is occupied by a single key-value pair, or (3) if it is occupied and there are duplicate values for the key. Duplicate values are stored externally in a contiguous memory space, and the `status` field is re-purposed to store a pointer to this memory location. We store the `status` and key value near the beginning of the bucket to ensure we can read both with one memory-load; this is obviously not possible if the key exceeds the size of a cache-line (minus 8 bytes). This is important since the `status` field is read on *every* hash-table access for both insertions and probes, whereas the key is needed to resolve key-collisions. The hash value is used only during table resizing to avoid recomputation. Since resizing is far more infrequent than insertions and probes, storing the hash value at the end does not impact overall join or aggregation performance.

Our hash-table design is amenable to both software and hardware prefetching. Since joins and aggregations operate on tuple vectors, software prefeteching will speed up the initial hash-table probe. Secondly, the hardware prefetcher kicks in to accelerate the linear probing search sequence for hash collisions. By front-loading the `status` field and the key, the DBMS tries to ensure that at most one memory reference to a bucket is necessary to check both if the bucket is occupied and if the keys match.

Although we can employ any software prefetching technique with ROF, we decided to use GP for multiple reasons. Foremost is that generating GP code is simpler than with SPP and AMAC. GP also naturally provides synchronization boundaries between code stages for a group to resolve potential data races when inserting duplicate key-value pairs. Additionally, it was shown in [14] that SPP offered minimal performance improvement in comparison to GP while having a more complex code structure. Finally, using an open-addressing hash-table with linear probing means that all tuples have exactly one random access into the hash-table during probes and insertions (with the exception of duplicate-handling which requires two). Since all tuples in a group have the same number of random accesses even in the presence of skew, AMAC does not improve performance over GP.

4.4 Query Planning

A DBMS's optimizer has to make two decisions per query when generating code with the ROF model: (1) whether to enable SIMD predicate evaluation and (2) whether to enable prefetching.

During optimization, Peloton's planner takes a greedy approach and installs a boundary after every scan operator if the scan has a SIMD-able predicate. Determining whether a given predicate can be implemented using SIMD instructions is a straightforward process that uses data-type and operation information already encoded in the expression tree. As we will show in Sect. 5, using SIMD when evaluating predicates during a scan never degrades performance.

The planner can also employ prefetching optimizations using two methods. In the first method, the query planner relies on database- and query-level statistics to estimate the sizes of all intermediate materialized data structures required by the query. For operators that require random access to data structures whose size exceeds the cache size, the planner will install a stage boundary at the operator's input to facilitate prefetching. This heuristic can backfire if the collected statistics are inaccurate (see Sect. 5.3) and result in a minor performance degradation. An alternative approach is for the query planner to *always* install a stage boundary at the input to any operator that performs random memory accesses, but generate two code paths: one path that does prefetching and one that does not. The query compiler generates statistics collection code to track the size of intermediate data structures, and then uses this information to guide the program through either code path at runtime. In this way, the decision to prefetch is independent of query planning. We note that this approach will result in a code explosion as each branch

Figure 8: TPC-H Query Plans with Pipelines – The high-level query plan for the subset of the TPC-H queries that we evaluate in our experiments, and do deep-dive analysis on. Each plan is annotated with their pipelines [30].

requires a duplication of the remaining query logic; this process can repeat for each prefetching operator. ROF remedies this by installing a stage boundary at the operator's output, thereby duplicating only the operator's logic rather than the entire query plan.

5. EXPERIMENTAL EVALUATION

We now present an analysis of our ROF query processing model. For this evaluation, we implemented ROF in the Peloton in-memory DBMS [5]. Peloton is an HTAP DBMS that used interpretation-based execution engine for queries. We modified the system's query planner to support JIT compilation using LLVM (v3.7). We then extended the planner to also generate compiled query plans using our proposed ROF optimizations.

We performed our evaluation on a machine with a dual-socket 10-core Intel Xeon E5-2630v4 CPU with 25 MB of L3 cache and 128 GB of DRAM. This is a Broadwell-class CPU that supports AVX2 256-bit SIMD registers and ten outstanding memory prefetch requests (i.e., ten line-fill buffer (LFB) slots) per physical core. We also tested our ROF approach with an older Haswell CPU and did not notice any changes in performance trends.

In this section, we first describe the workload that we use in our evaluation. We then present a high-level comparison of the performance of our ROF query processing model versus a baseline implementation that only uses query compilation. Next, we provide a detailed breakdown of the query plans to explain where the optimizations of the ROF model have their greatest impact. We then select the optimal query plan for each TPC-H query and measure the sensitivity of the results to two important compiler parameters for ROF (i.e., vector size and prefetching distance). To prevent cache coherence traffic from interfering with our measurements, we limit the DBMS to only use a single thread per query and do not execute multiple queries at the same time for these initial experiments. We then evaluate the DBMS's performance with ROF when using multiple threads and finish with a comparison of the absolute performance of Peloton with ROF against two state-of-the-art OLAP DBMSs.

We ensure that the DBMS loads the entire database into the same NUMA region using `numactl`. We run each experiment ten times and report the average measured execution time over all trials.

5.1 Workload

We use a subset the TPC-H benchmark in this evaluation [40]. TPC-H is a decision support system workload that simulates an OLAP environment where there is little to prior knowledge of the queries. It contains eight tables in 3NF schema. We use a scale factor of 10 in each experiment (∼10 GB). Peloton is still early in development; we plan to run larger scale experiments are it matures.

Although the TPC-H workload contains 22 queries, we select eight queries that cover all TPC-H choke-point query classifications [10] that vary from compute- to memory/join-intensive queries.

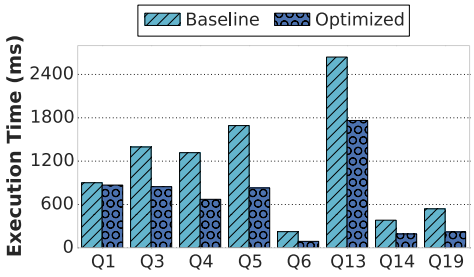


Figure 9: Baseline Comparison – Query execution time when using regular query compilation (baseline) and when using ROF (optimized).

Thus, we expect our results to generalize and extend to the remaining TPC-H queries. An illustration of the pipelined plans for these queries is shown in Fig. 8; the plan for Q19 is shown in Fig. 2a.

5.2 Baseline Comparison

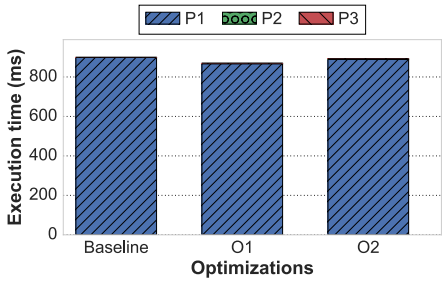
For this first experiment, we execute the TPC-H queries using the data-centric approach used in HyPer [30] and other DBMSs that support query compilation. We deem this as the baseline approach. We then execute the queries with our ROF model. This demonstrates the improvements that are achievable with the prefetching and vectorization optimizations that we describe in Sect. 4.

Fig. 9 shows the performance of our execution engine (which implements a data-centric query compilation engine [30]) both with and without our ROF technique enabled. As we see in Fig. 9, our ROF technique yields performance gains ranging from 1.7× to 2.5× for seven of eight queries (i.e., all but Q1, which we will discuss in detail below).

5.3 Optimization Breakdown

To better understand how ROF impacts performance, we now present case studies for a subset of the TPC-H queries we evaluated in Fig. 9. For the sake of space, we only discuss in detail five of the eight TPC-H queries we evaluate. The results we draw extend to the remaining queries. Figs. 10 to 14 show the execution time for each query broken down by the time spent in each pipeline in the original query plans from Fig. 8 as we incrementally apply ROF to additional operators within the tree. The leftmost bar is the baseline execution (without ROF), the next bar applies ROF to one operator, the bar after that includes both plus applying ROF to any additional operators. These optimizations are cumulative and each one is orthogonal to the previous. We describe the details of these optimizations in a table below each graph.

Q1 Case Study: ROF yields only a marginal improvement (1.04×) over the baseline for this query. The bulk of Q1's execution time is spent in **P1**, which performs a selection and an aggregation. **P2** and **P3** are not easily visible in Fig. 10: the time spent materializing



O1	**Modification: P1** ⇒ (`LineItem` → σ → Ξ → Γ) **Description:** Apply SIMD to predicate σ.
O2	**Modification:** — **Description:** Use Ξ to prefetch buckets on Γ build.

Figure 10: Q1 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q1 query plan shown in Fig. 8a.

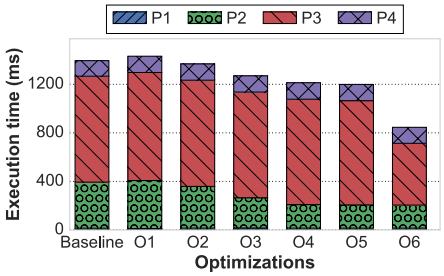
aggregated data into a memory heap (in preparation for sorting) and the sorting itself accounts for less than 0.3% of the execution time.

For our first optimization (**O1**), the planner converts the predicate on the `LineItem` table into a SIMD scan. It adds a stage boundary after σ in **P1**, and converts the scalar predicate evaluation into a SIMD check. But because this predicate matches almost the entire table (i.e., 98% selectivity). At such high-selectivity, the benefit data-level parallelism (i.e., reduced CPI) from using SIMD predicate evaluation is offset by the overhead of (redundant) data copying of valid IDs into an output vector. Thus, in this case SIMD yields only a marginal reduction in latency. Moreover, the scan portion of **P1** accounts for only 4% of the overall time; the remaining 96% of the time is in the aggregation (Γ). This means that applying SIMD to the scan (assuming a maximum theoretical speedup of 8× using AVX2 256-bit registers) can only achieve a speedup of at most 1.036×.

The second optimization (**O2**) uses the output of the SIMD scan stored in σ's output stage vector (Ξ) to prefetch hash-table entries in the build phase of the aggregation (Γ). But Fig. 10 shows prefetching makes the query slower. Γ's hash-table is sufficiently small enough (just four entries) to fit in the CPU's L1 cache, obviating the need for prefetching. The overhead of the DBMS invoking the prefetch instructions is non-negligible and thus degrades performance. **O2** highlights the importance of accurate query statistics. Inaccurate statistics may lead the planner to incorrectly install an ROF stage boundary to enable prefetching resulting in reduced performance.

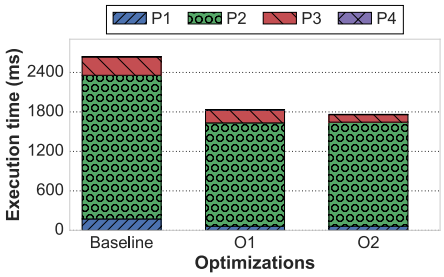
Q3 Case Study: This next query is the most complex one from the TPC-H workload that we evaluate. As such, it presents the largest number of opportunities to apply our ROF technique. The first observation from the measurements in Fig. 11 is that the optimizations that apply SIMD predicate evaluation (i.e., **O1**, **O2**, and **O5**) all offer marginal performance improvements. This is because the majority of time spent in pipelines **P1**, **P2**, and **P3** are not scanning table data, but rather in the joins $\bowtie_1$ and $\bowtie_2$. Roughly 1% of time in **P1** is spent filtering `Customer` tuples, 3.7% of **P2** is on scanning and filtering the `Orders` table, and 7% of **P3** is on scanning the `LineItem` table. Although using SIMD with these operators provides some benefit, it does not address the main bottlenecks.

Instead, the more complex, memory-intensive operators will produce greater improvements. These are the optimizations that implement prefetching of hash-table buckets for either the build- or probe-phase of joins $\bowtie_1$ or $\bowtie_2$ (i.e., **O3**, **O4**, and **O6**). **O3** uses the staging vector written to by the SIMD predicate on σ_2 to prefetch hash-table buckets for the probe of $\bowtie_1$. This speeds up **P2** by 1.4×. **O4** improves upon this by introducing a second stage boundary after the join $\bowtie_1$. This second stage prefetches hash-table buckets during



O1	**Modification: P1** ⇒ (`Customer` → σ_1 → Ξ_1 →$\bowtie_1$) **Description:** Apply SIMD to predicate σ_1 (`Cutomer`).
O2	**Modification: P2** ⇒ (`Orders` → σ_2 → Ξ_2 →$\bowtie_1$→$\bowtie_2$) **Description:** Apply SIMD to predicate σ_2 (`Orders`).
O3	**Modification:** — **Description:** Use Ξ_2 to prefetch buckets during $\bowtie_1$ probe.
O4	**Modification: P2** ⇒ (`Orders` → σ_2 → Ξ_2 →$\bowtie_1$→ Ξ_3 →$\bowtie_2$) **Description:** Use Ξ_3 to prefetch buckets during build of $\bowtie_2$.
O5	**Modification: P3** ⇒ (`LineItem` → σ_3 → Ξ_4 →$\bowtie_2$→ $Sort$) **Description:** Apply SIMD to predicate σ_3.
O6	**Modification:** — **Description:** Use Ξ_4 to prefetch buckets for $\bowtie_2$ probe.

Figure 11: Q3 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q3 query plan shown in Fig. 8b.



O1	**Modification: P2** ⇒ (`Orders` → σ → Ξ_1 →$\bowtie$→ Γ) **Description:** Use Ξ_1 to prefetch buckets for build and probe of $\bowtie$.
O2	**Modification: P2** ⇒ (`Orders` → σ → Ξ_1 →$\bowtie$→ Ξ_2 → Γ) **Description:** Use Ξ_2 to prefetch buckets during build of Γ.

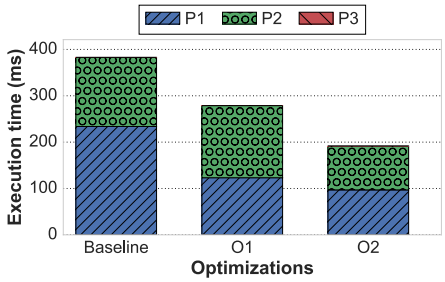
Figure 12: Q13 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q13 query plan shown in Fig. 8c.

the build phase of the join $\bowtie_2$. **O4**'s prefetching improves **P2**'s execution time by 1.26× and for the overall query by 1.14×.

The last and most important optimization for Q3 is **O6** because this highlights the advantage of the ROF approach. While using SIMD for the predicate evaluation on `LineItem` (which is the largest table in the query) only increases performance by 1.02×, using the resulting output vector to perform prefetching on the probe of $\bowtie_2$ results in an improvement of 1.38×. In general, we find that using ROF optimizations across this query plan improves by up to 1.61×, with much of this resulting from prefetching.

Q13 Case Study: This query presents another interesting data point because it demonstrates that ROF can still improve performance when only using prefetching without SIMD. Q13 contains a predicate on `Orders` that cannot be implemented using SIMD since it contains a non-trivial `LIKE` clause on a string column. However, by installing a stage boundary after the predicate (σ), Fig. 12 shows that ROF still improves performance over the baseline.

The first optimization (**O1**) prefetches hash-table buckets for both the build- and probe-phases of the group-join ($\bowtie$). We implement



O1	**Modification: P1** ⇒ (`LineItem` → σ → Ξ →⋈) **Description:** Apply SIMD to predicate σ (`LineItem`).
O2	**Modification:** — **Description:** Use Ξ and `Part` to prefetch buckets during join ⋈.

Figure 13: Q14 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q14 query plan shown in Fig. 8d.

the group-join operator from [28]. As described previously, the DBMS installs a stage boundary (Ξ_1) after the predicate σ. The scan of `Orders` is entirely scalar, and its output is written to the stage's output vector. The DBMS is then able to prefetch on the build input since it is already being read from materialized state (i.e., the `Customer` table). **O1** results in a performance improvement of 1.34× over the baseline.

The second optimization (**O2**) prefetches hash-table buckets needed during the build phase of the aggregation (Γ). Once again the DBMS installs a stage boundary (Ξ_2) after the group-join (⋈). This optimization further improves performance over **O1** by 1.04×, resulting in an overall improvement of 1.5× compared to the baseline.

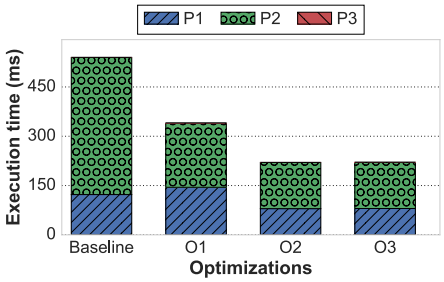
Q14 Case Study: The predicate on the `LineItem` table has only a 2% selectivity and, hence, it is an ideal candidate for being converted into a SIMD predicate. To do so, in **O1** the planner installs a stage boundary (Ξ) after the predicate over `LineItem` (σ). This improves **P1**'s execution time by 1.89× and Q14's overall time by 1.37×.

Next, the planner enables prefetching for both the build- and probe-phases of the join in **O2**. For **P1**, this is facilitated by re-using **O1**'s output vector (Ξ) for the SIMD predicate (σ). **P2** does not need an additional stage since the scan of the `Part` table is directly from the table. **O2** further improves performance by 1.45× from the previous optimization and by almost 2× over the baseline.

We do not apply any optimizations on the aggregation operator (Γ) because it is static (i.e., it always generates a single output tuple). In generated code, the aggregation is implemented as a simple counter. Hence, **P3** requires no computation and contributes effectively nothing to the query's overall execution time of the query.

Q19 Case Study: Like Q14, the predicate on the `LineItem` table in this query is selective; less than 4% of tuples make it through the filter. This predicate is extracted from a much larger disjunctive predicate that is applied to the results of the join (⋈). Again, this means that the predicate is a good candidate for SIMD evaluation. Thus, **O1** installs a stage boundary and output vector (Ξ_1), after the predicate on `LineItem` (σ_1). The results in Fig. 14 show that **O1** improves the overall performance of the query by almost 1.6×.

The second optimization uses **O1**'s output vector to issue prefetch instructions for hash-table buckets during the hash-join probe. Similarly, **O2** also modifies **P1** to prefetch hash-table buckets to build ⋈. Together these two optimizations further improve Q19's performance by almost 1.6× and almost 2.5× over the baseline implementation. We note that the contribution of **P3** is effectively zero because it is a static aggregation that performs no computation.



O1	**Modification: P2** ⇒ (`LineItem` → σ_1 → Ξ_1 →⋈→ σ_2 → Γ) **Description:** Apply SIMD to predicate σ_1 (`LineItem`).
O2	**Modification:** — **Description:** Use Ξ_1 to prefetch buckets during probe of ⋈.
O3	**Modification: P2** ⇒ (`LineItem` → σ_1 → Ξ_1 →⋈→ Ξ_2 → σ_2 → Ξ_3 → Γ) **Description:** Insert staging points between every pair of operators.

Figure 14: Q19 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q19 query plan shown in Fig. 2a.

5.4 Sensitivity to Vector Width

In the previous experiments, we executed the queries using the optimal vector sizes and prefetch group size. We now analyze the sensitivity of the ROF model to these two configuration parameters. Our evaluation shows that these parameters are independent to each other and thus we will examine them separately. We begin with measuring the effect of the stage output vector size to overall query performance. We select the optimal staged plan for the eight TPC-H queries and fix the prefetch group size to 16. We then vary the size of all the output vectors in each plan from 64 to 256k tuples.

The results in Fig. 15 show that all but one of the queries (Q13) are insensitive to the size of the stages' output vectors. This is notable for Q14 and Q19 because we showed in the previous experiment that they both benefited greatly from SIMD vectorization. This is because their scans are highly selective (2% for Q14 on the `Orders` table and 4% for Q19 on the `LineItem` table), and so using SIMD shifts the main bottleneck to the next stage in their respective pipelines. For Q14, this is mainly the probe-side of the join, whereas it is the build-side of the join in Q19. The extra latency incurred due to accessing larger-than-cache hash-tables constitutes the largest time component of their execution plan, even though it is ameliorated through prefetching.

Q1 is also insensitive to the vector size, but for a slightly different reason. In this query, more than 98% of the tuples in the `LineItem` table qualify the predicate. As such, the primary bottleneck in the first pipeline is not in the SIMD scan, but in the aggregation. This aggregation does not benefit from larger vector sizes, which is why Q1 is not impacted by varying this configuration parameter.

For Q13, the primary bottleneck in the query plan is the evaluation of the `LIKE` clause on the `o_comment` field of the `Orders` table. Since the DBMS cannot execute this predicate using SIMD, increasing the size of the predicate's output vector does not help. It instead uses this vector to prefetch hash-table buckets as part of the subsequent probe. But since the query plan uses a group size of 16, the system is already able to saturate the memory parallelism in the hardware and thus larger vector sizes do not help.

The only query whose performance gets better as the vector width increases is Q3. The results show that this query benefits slightly with larger vector sizes, but does not improve further when vector size exceeds 16k tuples. This is because the SIMD scan on the `LineItem` table remains in the SIMD code stage over a larger range of tuples. The predicate is roughly 54% selective, and so using

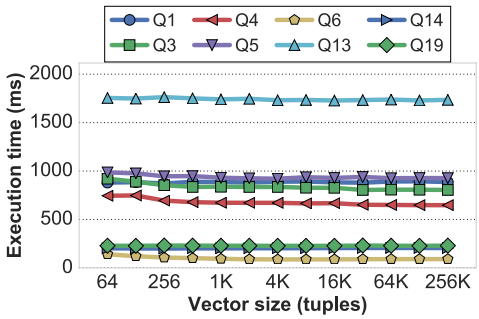


Figure 15: Sensitivity to Vector Width – The average execution time of the TPC-H queries when varying the maximum number of tuples stored in the ROF's stage output vectors.

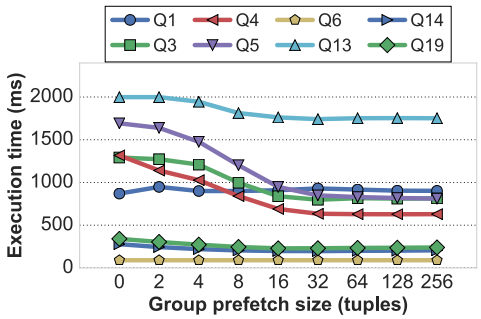


Figure 16: Sensitivity to Prefetching Distance – The execution time of the TPC-H queries when varying the ROF model's group prefetch size.

larger vectors reduces the number of outer-most loop iterations. In general, larger vectors reduces the total number of outer-most loop iterations. This is helpful for scan queries with low-selectivity predicates. However, larger vectors don't improve performance for queries with joins. This is because modern CPUs support a limited number of outstanding memory references, making the query become memory-bound quicker than CPU-bound.

5.5 Sensitivity to Prefetching Distance

We next analyze the performance of the DBMS when varying the size of the ROF model's prefetch groups. We again use the best staged plans that we generated in Sect. 5.2 for each query. This time we fix the output vector size constant to use the optimal configuration for each query as determined in Sect. 5.4. We then vary the group sizes from zero (disabling prefetching) to 256 tuples.

The results in Fig. 16 show that prefetch group size has a strong influence the performance of all queries, with the exception of Q1 and Q6. Q6's performance remains constant across prefetch groups because it contains only a sequential scan. In the previous section, we showed that Q1 is insensitive to the output vector size. But we now also see that Q1 is not affected by this other parameter as well. We attribute this to the fact that the only data structure that is prefetched in Q1 (i.e., aggregate hash-table) fits in the CPU's L1 cache. In fact, Fig. 16 shows that Q1's lowest execution time is when there is no prefetching at all. This indicates that the combination of the weak predicate and a small hash-table makes Q1 ill-suited for ROF. We note, however, that ROF does not degrade the performance of Q1 (as seen in Figs. 9 and 10) since the predicate, though weak, can still use SIMD vectorization.

Our second observation is that all of the queries get faster with increasing group sizes up until 16 tuples. The CPU used in our evaluation supports a maximum of 10 outstanding L1 cache references (per core), and thus one would expect the optimal group size to be 10 since this should saturate the CPU's memory-level parallelism. These results, however, show that the optimal group size is 16. This is because the GP technique that we implemented is also limited by instruction count. Using larger group sizes enable fewer iterations of the outer-most loop, which reduces overall instruction count. Hence, groups larger than 16 do not improve performance because at that point the DBMS saturates the CPU.

5.6 Multi-threaded Execution

We now evaluate the performance of Peloton with ROF when executing queries using a multiple threads. We implemented a simplified version of the multi-threaded execution strategy employed in HyPer [26]. Each pipeline in the query plan is executed using multiple threads that each modify only thread-local state. At pipeline-breaking operators, a single coordinator thread coalesces data produced by each execution thread.

We ran the eight TPC-H queries from Sect. 5.2, using each query's best staged plan. We vary the number of execution threads from one to the number of physical cores in the benchmark machine. We report averages over ten runs using a TPC-H SF10 database.

The results in Fig. 17 demonstrate that using ROF with vectorization and prefetching complements multi-threaded query execution. We first note that the jump in execution time when moving from one thread to two threads for all the queries is due to the non-negligible bookkeeping and synchronization overhead that is necessary to support multi-threaded execution. This is independent of the ROF model. As described earlier, hash-joins end at a synchronization barrier on the build-side as execution threads wait for the coordinator thread to construct a global hash-table. For low thread counts, this overhead outweighs the benefit of multiple threads. But this cost is eliminated with the addition of more execution threads.

Fig. 17a shows that ROF does not improve Q1 since it is a CPU-bound query with a high-selectivity predicate. This corroborates our previous results in Sects. 5.3 to 5.5. Executing Q1 with multiple threads yields a consistent increase up to 20 cores at which point all CPUs are fully utilized and have saturated the memory bandwidth.

The other seven TPC-H queries exhibit similar speed-up with increasing thread counts. Although each execution thread constructs a small thread-local hash-table, the size of the coordinator thread's global hash-table will always exceed the CPU cache size. Since the join's probe-side is usually an order-of-magnitude larger than the build-side, Figs. 17b and 17f to 17h shows that using ROF with prefetching improves performance by 1.5–1.61× over the baseline.

One final observation is the slight variance in execution times in Q3 and Q13 with more than 10 threads. This jitter is due to NUMA effects on our two CPU machine (10 cores per socket). Both Q3 and Q13 execute using a group hash-join, meaning that the DBMS uses the materialized hash-table during building and probing. Concurrent updates to the table are serialized using 64-bit compare-and-swap instructions. Hence, CPUs in different NUMA regions experience different latency when accessing these counters stored in the global hash-table. Q5 exhibits this effect as it requires two global hash-tables probes (i.e., four random memory accesses). Despite this, ROF is still improves performance by 1.5×.

5.7 System Comparison

Lastly, we compare Peloton with ROF against two state-of-the-art in-memory databases: Vector [1] and HyPer [20]. The former is a columnar DBMS based on MonetDB/x100. It uses a vectorized execution engine that supports both compressed tables and SIMD instructions when available. The latter is also a columnar DBMS, but uses LLVM (like Peloton) to generate compiled tuple-at-a-time

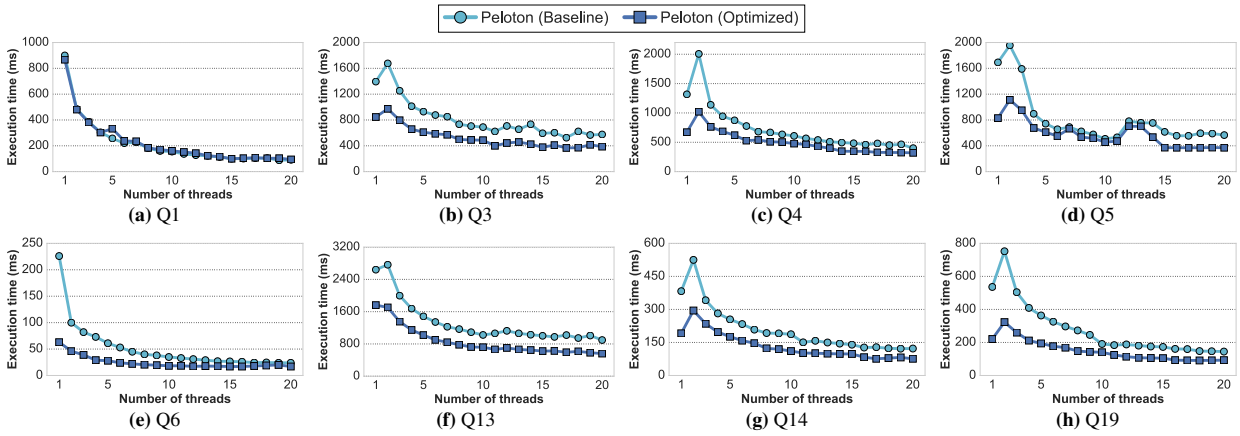


Figure 17: Multi-threaded Execution – The performance of Peloton when using multiple threads to execute queries with and without the ROF model.

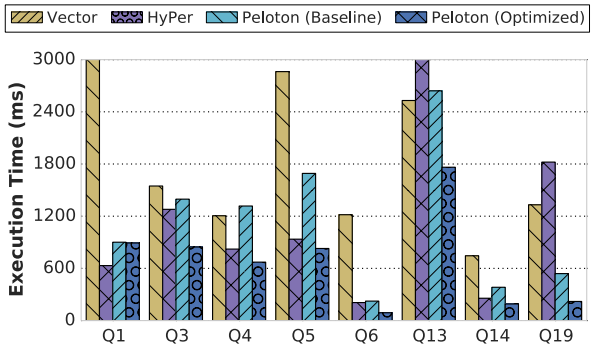


Figure 18: System Comparison – Performance evaluation of the TPC-H benchmark in Vector, HyPer, and Peloton with and without our ROF model.

query plans. For Peloton, we execute the queries both with and without our ROF model enabled; this corresponds to the "baseline" and "optimized" configurations from Sect. 5.2.

We deployed all of the DBMS using the same hardware and database as described in Sect. 5.1. To ensure a fair comparison, we disabled multi-threaded execution in all of the systems and made a good faith effort to tune each one for TPC-H. We note, however, that both Vector and HyPer include additional optimizations that may not exist across all three DBMSs. Thus, we tried to ensure the query plans generated in each system are equivalent or at least do not differ too greatly. We warm up each DBMS first by executing all of the TPC-H queries without taking measurements.

The results of this experiment are shown in Fig. 18. We now provide an analysis of the eight TPC-H queries:

Q1: HyPer performs the best in Q1, completing almost 1.38× faster than Peloton, and more than 6.5× faster than Vector. This is because HyPer uses fixed-point decimal arithmetic rather than more computationally expensive floating point arithmetic.

Q3: Peloton with our ROF technique outperforms Vector and HyPer by 1.8× and 1.5×, respectively. Additionally, we see that Peloton without our ROF technique is comparable to HyPer since they both use the same push-based compiled queries. Q3 contains two joins, both of which access a hash-table that does not fit in cache if the join is not partitioned using early materialization. This means that every access to the hash-table is a cache-miss. Our ROF technique hides this cache-miss latency by overlapping computation and memory accesses of different tuples. While (radix) partitioning-based joins is another strategy to solve this problem, previous work has shown that the subsequent overhead of gathering attributes necessary for operators following the join negates the benefits of gained by in-cache joins [36]. Second, Vector executes the `LineItem` predicate using a SIMD scan. This version of HyPer does not implement SIMD scans, though this is addressed in later work [25].

Q4: Peloton with our ROF technique outperforms both Vector and HyPer by 1.8× and 1.2×, respectively. Peloton without ROF has performance comparable to Vector, but is outperformed by HyPer. The primary reason for this is because HyPer uses CRC32 for hashing, implemented using SIMD (SSE4), whereas Peloton uses the more computationally intensive MurmurHash3. Thus, the cache benefits afforded by prefetching with ROF are slightly offset by the higher instruction overhead (in comparison to HyPer) due to a more complex hash function. In general, Peloton with ROF improves performance over the baseline by more than 2×.

Q5: Peloton with ROF is roughly 3.4× faster than Vector and 1.1× faster than HyPer. Q5 stresses join performance as it contains a five-way join between the largest tables in the benchmark. Hence, prefetching plays an important role as materialized join-tables will exceed the size of cache. Peloton with ROF (and prefetching) improves baseline performance by over 2×, but only offers a small improvement over HyPer. As in Q4, this is due to the simpler CRC32 hash function employed by HyPer. The scan over `LineItem` contains no restrictions before probing the hash-table on `Orders`, hence the majority of time is spent performing hash computations in Peloton. This higher instruction count in comparison to HyPer offsets the benefits of prefetching.

Q6: Peloton with ROF outperforms Vector by 5.4× and HyPer by 2.3×. Q6 is a sequential scan with a highly selective predicate (1.2%). Hence, leveraging SIMD predicate evaluation yields significant performance improvements, more than 2× in Peloton. The version of HyPer we use does not include the SIMD optimizations [25], but we believe it will also enjoy similar benefits of SIMD.

Q13: In Q13, we see that Peloton without ROF performs worse than Vector, but when we enable ROF to remove the cache miss penalties incurred during the join it performs roughly 1.4× faster than Vector. We note here that the majority of time spent in executing this query is in the scan of the `Orders` table. This is because the scan involves a `LIKE` clause and thus the query's performance hinges on the performance this evaluation. We note that Peloton uses a simple comparison for the `LIKE` function that assumes clean input data. The slower results for Vector and HyPer suggest to us that they

are likely using more sophisticated implementations that are able to handle problematic data better (e.g., broken UTF encodings).

Q14: This query contains a highly selective scan on `LineItem` that benefits from a SIMD implementation. Peloton and Vector are able to take advantage of this optimization, but HyPer (in this version) and Peloton without ROF must execute a scalar scan. We note that both Peloton and Vector implement Q14 with a hash-join, whereas HyPer uses an index nested-loop join. This is the reason why Peloton without ROF is slower than HyPer since the probe phase of the join will incur the additional overhead of duplicate chain traversal. But with the addition of SIMD scan and prefetching over the build- and probe-phase of the join, Peloton with ROF performs 3.9× and 1.35× faster than Vector and HyPer, respectively.

Q19: Similar to Q14, Q19 also contains a highly selective scan on `LineItem`. But Peloton uses dictionary-encoding for the filtered attributes because their cardinalities are sufficiently small. With dictionary-encoding enabled, Peloton with ROF converts the scalar scan over `LineItem` into vectorized scan using SIMD. Vector also automatically compresses strings. With the addition of prefetching and staging, Peloton with ROF executes this query 6× faster than Vector and 8× faster than HyPer.

6. RELATED WORK

A primitive form of code generation was developed for IBM System R in 1970s [13]. System R directly compiled SQL statements into assembly code by selecting pre-defined code templates for each operator. The IBM researchers later remarked that though compiling repetitive queries had obvious benefits by avoiding the cost of parsing and optimization, the benefits of compiling ad-hoc queries were less clear. But IBM abandoned this approach in the early 1980s because of the high cost of external function calls, poor portability across operating systems, and software engineering complications.

Query compilation was not considered in any major DBMS in the 1980s and 1990s (with a few minor exceptions). Instead, it was supplanted by the Volcano query processing model [19]. The Volcano abstraction allows a DBMS's query planner to compose plans from operators. It is also easier to implement and manage in practice and offers comparable performance as query compilation when operating in a disk-oriented DBMS. This is because the dominant factor in query evaluation is the time required to retrieve data from disk and not the overhead of interpretation.

More recently, query compilation has been used in modern in-memory DBMSs. One of the first systems to revive the technique was Microsoft's Hekaton [18], an in-memory OLTP storage manager for the SQL Server DBMS. Hekaton compiles queries by transforming a conventional query plan into a single C-code function that implements a Volcano-style iterator.

Cloudera's Impala [23] is a distributed OLAP DBMS with a mixed C++/LLVM vectorized execution engine. Impala uses LLVM to compile query-specific versions of frequently executed functions, including functions to parse tuples, compute tuple hashes, and evaluate predicates. Impala also compiles and inlines UDFs into the query's execution plan.

HyPer [20] pioneered the data-centric (push-based) query execution model [30]. HyPer translates a given query plan into LLVM IR, but relies on precompiled C++ code for the more complex query-agnostic database logic. The push-based engine fuses together all operators in a pipeline, obviating the need to materialize data between operators, instead allowing them to access tuple attributes directly in CPU registers. This produces compact loops that improve code locality and overall execution time.

MemSQL [4] also uses the LLVM to perform whole query compilation. Query parameters are stripped out from queries to avoid the need for recompilation when the query is run with new input values.

LegoBase [21] uses generative-programming (i.e., staging) to partially evaluate a Volcano-style interpretation engine and produce highly customized C query code. Optimizations to convert to push-based dataflow, row or columnar formats, or vectorized or tuple-at-a-time processing are applied during this transformation.

DBToaster [8, 7] is a stream processing engine designed for efficient view maintenance. In existing DBMSs, incremental updates to materialized views are treated like an update to a regular table. DBToaster instead analyzes these relationships to construct an optimized delta query that often obviates the need for subsequent scans of base tables. It then translates this delta query into C++ code and compiles it into machine code using a standard compiler.

Tupleware [15] is a distributed DBMS that automatically compiles workflows composed of UDFs into LLVM IR. Workflows are introspected to find vectorizable and non-vectorizable portions that drives the code generation process. Tupleware also presents a new hybrid predicate evaluation technique that separates predicate checking with output copying using a heuristic model.

In [38] and [37], the authors compared the performance of a vectorized and compiled query engine across three different simple query types: projections, selections and hash joins. They conclude that neither technique is always optimal, but that a combination of the two techniques is required to achieve the best performance.

The HIQUE [24] system uses query compilation without the Volcano iterator model. Instead, HIQUE translates the algebraic query plan into C++ using code templates for each operator. These templates form the structure of the operator, but low-level record access methods and predicate evaluation logic is customized per query. Unlike our ROF model, each operator in HIQUE always materializes its results, which prevents operator pipelining.

MapD [39] is a GPU-accelerated DBMS designed to handle read-only queries. It implements a mixed C++/LLVM execution engine. All query-specific routines and predicate expressions are compiled into LLVM IR, then JIT compiled into native GPU code through Nvidia's intermediary NVVM IR. Like MemSQL, MapD extracts constants to avoid recompilation when a query is re-executed with different parameters. It does this by using the generated IR as a key into a hash-table that maps IR to JITed query code.

7. CONCLUSION

We presented the relaxed operator fusion query processing model for in-memory OLAP DBMSs. With ROF, the DBMS introduces staging points in a query plan where intermediate results are temporarily materialized to cache-resident buffers. Such buffers enables the DBMS to employ various optimizations to exploit inter-tuple parallelism using a combination of vectorization and software prefetching. This allows a DBMS to support faster OLAP query execution and to support vectorization optimizations that were previously not possible when data sets exceed the size of CPU-level caches. We implemented our ROF model in the Peloton in-memory DBMS and showed that it reduces the execution time of OLAP queries by up to 2.2×. We also compared Peloton with ROF against two other in-memory DBMSs (HyPer and Actian Vector) and showed that it achieves 1.8× lower execution times.

Acknowledgements: This work was supported (in part) by the National Science Foundation (CCF-1438955, IIS-1718582), and the Intel Science and Technology Center for Visual Cloud Systems. We also would like to thank Tim Kraska for his feedback.

8. REFERENCES

[1] Actian Vector. http://esd.actian.com/product/Vector.

[2] Apache Cassandra. http://cassandra.apache.org/.

[3] Apache Spark. http://spark.apache.org/.

[4] MemSQL. http://www.memsql.com.

[5] Peloton Database Management System. http://pelotondb.io.

[6] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980, 2008.

[7] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[8] Y. Ahmad and C. Koch. Dbtoaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.

[9] A. Appleby. MurMur3 Hash. https://github.com/aappleby/smhasher.

[10] P. Boncz, T. Neumann, and O. Erling. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*. 2014.

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

[12] D. Broneske, A. Meister, and G. Saake. Hardware-sensitive scan operator variants for compiled selection pipelines. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 403–412, 2017.

[13] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of system r. *Commun. ACM*, 24:632–646, October 1981.

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116–127, 2004.

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

[16] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: "big" data, big analytics, small clusters. In *CIDR*, 2015.

[17] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. SIGMOD '16, pages 215–226, 2016.

[18] C. Freedman, E. Ismert, and P.-A. Larson. Compilation in the microsoft SQL server hekaton engine. *IEEE Data Eng. Bull.*, 2014.

[19] G. Graefe. Volcano- an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6:120–135, 1994.

[20] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[21] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[22] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. *PVLDB*, 9(4):252–263, 2015.

[23] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research*, 2015.

[24] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 613–624. IEEE, 2010.

[25] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 311–326, 2016.

[26] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, 2014.

[27] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 222–233, 1996.

[28] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11):843–851, 2011.

[29] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 62–73, 1992.

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

[31] S. Pantela and S. Idreos. One loop does not fit all. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 2073–2074, 2015.

[32] D. Paroski. Code Generation: The Inner Sanctum of Database Performance. http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html, September 2016.

[33] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. SIGMOD, pages 1493–1508, 2015.

[34] O. Polychroniou and K. A. Ross. Vectorized bloom filters for advanced simd processors. DaMoN '14, pages 6:1–6:6, 2014.

[35] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.

[36] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1961–1976, 2016.

[37] J. Sompolski. Just-in-time Compilation in Vectorized Query Execution. Master's thesis, University of Warsaw, Aug 2011.

[38] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, 2011.

[39] A. Suhan and T. Mostak. MapD: Massive Throughput Database Queries with LLVM on GPUs. http://devblogs.nvidia.com/parallelforall/mapd, June 2015.

[40] The Transaction Processing Council. TPC-H Benchmark (Revision 2.16.0). http://www.tpc.org/tpch/, June 2013.

[41] S. D. Viglas. Just-in-time compilation for sql query processing. *PVLDB*, 6(11):1190–1191, 2013.

[42] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[43] M. Zukowski, N. Nes, and P. Boncz. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. DaMoN '08, pages 47–54, 2008.