Towards Practical Vectorized Analytical Query Engines

Orestis Polychroniou*
orestis@amazon.com
Amazon Web Services

Kenneth A. Ross†
kar@cs.columbia.edu
Columbia University

ABSTRACT

Query execution engines are adapting to the underlying hardware in order to maximize performance. Wider SIMD registers and more complex SIMD instruction sets are emerging in mainstream CPUs as well as new processor designs, such as the many-core platforms that rely on data parallelism via SIMD vectorization to pack a larger number of smaller cores per chip. In the database literature, using SIMD to optimize stand-alone operators with key–rid pairs is common, yet the state-of-the-art query engines rely on compilation of tightly coupled operators where hand-optimized individual operators become impractical. In this paper, we present VIP, an analytical query engine designed and built bottom-up from pre-compiled column-oriented data-parallel sub-operators and implemented entirely in SIMD. In our evaluation derived from the TPC-H workload, VIP outperforms query-specific hand-optimized scalar code.

1 INTRODUCTION

Hardware-conscious database design and implementation is a topic of continuous research due to the profound impact of modern hardware advances on query execution. Database systems diverged to focus on transactional, analytical, scientific, or other workloads. Storage and execution, now narrowed down to specific workloads, were redesigned by adapting to the new hardware dynamics. In analytical database systems, column-oriented storage is now a standard design choice, since queries typically access a small number of columns from a large number of tuples, in contrast to transactions that update a few tuples. However, analytical query engines are based on multiple designs, including column-oriented versus row-oriented execution, interpretation versus per-query code compilation, cache-conscious execution versus operator pipelining.

Efficient in-memory execution requires low interpretation cost, optimized memory access, and high CPU efficiency. Low interpretation cost is coupled with high instruction-level parallelism and is achieved by processing entire columns [27], batches of tuples per iterator call [5, 8, 9], or by compiling query-specific code at runtime [13, 20, 31]. Memory access can be optimized by pipelining operators to minimize materialization [12], or by partitioning to avoid cache and TLB misses [28]. Data parallelism is achieved via SIMD vectorization. Linear-access operators such as scans and compression [22, 38, 49, 50], are naturally data-parallel and easy to vectorize. The literature on applying ad-hoc SIMD optimizations to individual database operators [7, 14–16, 23, 35, 36, 41, 42, 44, 46] is rich. Recently, we introduced generic SIMD vectorization techniques for non-linear-access operators, such as partitioning, building and probing of hash tables, as well as probing Bloom filters [34, 37].

In this paper, we introduce VIP[1], an analytical query execution engine designed and built bottom-up from pre-compiled data-parallel sub-operators and implemented entirely in SIMD. VIP is the first query engine design implemented using advanced SIMD vectorization techniques [34] that supports realistic queries, with multiple columns and data types per operator, complex combinations of predicates in scans and joins, and multiple aggregates with expressions. To support these cases, VIP operators invoke pre-compiled sub-operators. Each sub-operator processes data almost exclusively in SIMD registers, from one column at a time [27], for one block of tuples at a time [5]. Query code generation and compilation is the state-of-the-art design for query engines [20, 31] and is employed by modern commercial analytical database systems [9, 13]. In our evaluation using queries derived from TPC-H, VIP outperforms hand-optimized query-specific scalar code designed to emulate the state-of-the-art, without including the per-query compilation overhead at runtime that is non-existent in VIP.

In Section 2, we present related work. In Section 3, we describe the VIP engine design. In Section 4, we present our evaluation, in Section 5 we discuss future work, and we conclude in Section 6.

2 RELATED WORK

Block-at-a-time[2] execution [5] and query-specific code generation and compilation [13, 20, 31] are the state-of-the-art designs for analytical query engines. Both eliminate the interpretation overhead but the latter incurs runtime compilation overhead to fuse pipelined operators and minimize intermediate result materialization. Block-at-a-time execution can use prefetching to mitigate the cost of cache misses [30]. Furthermore, the benefit of basic SIMD vectorization is also diminished when we are memory bound [18].

SIMD vectorization is typically applied to isolated database operators using key–rid pairs. Many join implementations exist [2–4, 19, 45], including for many-core platforms [6, 17]. SIMD implementations of stand-alone operators such as sorting [7, 14, 36, 43] are also common. Linear-access operators such as scans [21, 50] and compression [22, 25, 26, 38, 49], are data-parallel and easier to vectorize. Non-linear-access operators such as hash tables and partitioning require more advanced SIMD vectorization techniques

*Work completed while the first author was affiliated only with Columbia University.
†Supported by National Science Foundation grant IIS-1422488 and an Oracle gift.

DaMoN'19, July 1, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6801-8/19/07…$15.00
https://doi.org/10.1145/3329785.3329928

[1]Named after and primarily based on ***V**ectorization*, ***I**nterpretation*, and ***P**artitioning*.
[2]Block-at-a-time is a notion that was termed *vectorized* in related work; in this paper we use the term *vectorized* to denote SIMD-vectorized design and implementation.

[34, 37]. Fully vectorized implementations are not used in commercial systems due to the difficulty of supporting multiple columns, data types, and complex predicates effectively. The most principled approach is a recently proposed vector algebra [33] for both SIMD CPUs and SIMT GPUs. However, non-linear-access operators are basic loops over SIMD primitives such as joins using SIMD gathers for rid dereferencing out of the cache. Simple loops of SIMD loads and gathers and compiler auto-vectorization achieve marginal benefit [18] and are not comparable with the complexity of VIP.

3 DESIGN & IMPLEMENTATION

In VIP, we use column-oriented execution [27, 47], but never process an entire column at once. Instead, we process one column for the next block of tuples, retaining intermediate results in the cache to amortize the interpretation cost [5]. However, instead of using the Volcano model on blocks of tuples, we extend the column-at-a-time model [27]. In our experiments, we use static allocation of work units across threads due to the NUMA effect of our platform. Nevertheless, VIP can be extended to use dynamic allocation [24].

VIP operators are designed and built bottom-up from column-oriented data-parallel *sub-operators* and are implemented entirely in SIMD. During query execution, each operator invokes pre-compiled sub-operators to processes one column at a time from one block of tuples. The block size is chosen to amortize the cost of interpretation and retain the working set in the fast private cache of the CPU core. Sub-operators execute a basic operation on one column of a specific data type, thus allowing for extreme manual optimization.

To explain how sub-operators work across columns, consider a hash join on a composite key comprised of multiple columns. To hash the columns, we invoke the hash sub-operator for the 1st column based on its data type and store the hash values in the cache. Then, we call the hash sub-operator for the 2nd column and update the hash values. If the block size is small enough, the hashes remain in the cache. The VIP sub-operator to FNV hash [11] a 32-bit integer column is shown below. VIP is implemented in AVX-512 SIMD.

```
void hash_int32(int32_t* data, uint32_t* hash, size_t tuples) {
  const __m512i m_255 = _mm512_set1_epi32(255);  // mask to isolate lower byte
  const __m512i m_fnv = _mm512_set1_epi32(16777619);  // FNV prime constant
  for (size_t i = 0; i != tuples; i += 16) {  // 16 SIMD lanes
    __m512i h = _mm512_load_epi32(&hash[i]);  // load hash values from cache
    __m512i d = _mm512_load_epi32(&data[i]);  // load data values from column
    for (size_t j = 0; j != 4; ++j) {  // FNV hash processes 1 byte at a time
      h = _mm512_ternarylogic_epi32(h, d, m_255, 120);  xor byte with hash
      h = _mm512_mullo_epi32(h, m_fnv);  // multiply hash with the FNV prime
      d = _mm512_srli_epi32(d, 8); }  // shift right to process the next byte
    _mm512_store_epi32(&hash[i], h); }  // store back the updated hash values
  [...] }  // process the last (up to 15) values using scalar code
```

Every VIP sub-operator has to be implemented for each data type T. For instance, the hash sub-operator prototypes are of the form: `void hash_T(T* data, uint32_t* hash, size_t tuples)`.

3.1 Selection Scans

In modern analytical database systems, linear scans are preferred over indexes for selections [39, 40]. Code generating engines also prefer pre-compiled SIMD code for scans and decompression [22]. In the original column-at-a-time execution model [27], each predicate produces an rid list (i.e., array) denoting qualifying tuples. Conjunctions and disjunctions are computed via the union or intersection of rid lists. In VIP, we also use bitmaps to store the intermediate results across predicates, similarly to commercial systems [39].

On linear scans, performance is dictated by the number of accessed cache lines. SIMD naturally handles predicates on W contiguous values very efficiently, thus, we evaluate or skip in units of W tuples. The sub-operator to evaluate comparison predicates on 32-bit integer columns is shown below. The input bitmap denotes which tuples can be skipped and which tuples we still need to evaluate. The output bitmap is the evaluation result. Note that the output bitmaps do not have to be strict subsets of the input bitmaps. We express complex predicates as a tree and process one predicate at a time for the same block of tuples while traversing the tree.

```
void select_int32(const int32_t* data, size_t tuples,  // the input column
    const uint16_t* bitmap_in,  // bitmap denoting the yet undetermined tuples
    uint16_t* bitmap_out,  // bitmap denoting a superset of qualifying tuples
    int32_t constant, int op) {  // the comparison constant and the operator
  const __m512i m_con = _mm512_set1_epi32(constant);  // mask
  switch (op) {  // separate code per comparison operator
  case '>':  // code for the ">" operator (can be split as separate function)
    for (i = 0; i != tuples; i += 256) {  // scan over data column and bitmap
      __m256i bit = _mm256_load_si256(bitmap_in);  // load 256 bitmap bits
      if (!_mm256_testz_si256(b, b)) {  // skip 256 tuples if all bits are 0
        __m512i bit_E = _mm512_cvtepu16_epi32(bit);
        // find groups of 16 tuples (out of 256 tuples) that are not all 0
        uint64_t m = _mm512_test_epi32_mask(bit_E, bit_E);
        do {  // evaluate groups of 16 tuples at a time that are not all 0
          size_t j = _tzcnt_u64(m);  // load 16 values from the 32-bit column
          __m512i key = _mm512_load_epi32(&data[j << 4]));
          // evaluate the selective predicate (in this example greater than)
          bitmap_out[j] = _mm512_cmpgt_epi32_mask(key, m_con);
        } while (m = _blsr_u64(m)); }  // process the next 16 tuples
      bitmap_in += 16, bitmap_out += 16, data += 256; }
    break; [...] }}  // other comparison operators, e.g., "=", "<", etc.
```

For complex predicate expressions, exhaustively evaluating every predicate on every column and then merging the bitmaps is trivial. However, in some settings, short-circuiting predicates can be orders of magnitude faster, thus we use the output of earlier predicates to skip evaluating later predicates as well as skip loading columns. In conjunction nodes, the input bitmap represents the tuples that satisfy all previous predicates, i.e., siblings of the current node, and the remaining tuples can be skipped. In disjunction nodes, the input bitmap represents the tuples that failed all previous predicates. In the example of Figure 1, we show the input bitmap denoting the tuples we need to evaluate per predicate where we do not skip any tuples. To support `null` values, we can either use a separate bitmap denoting `null` merged with the other predicates, or use a special value per type evaluated as an additional predicate.

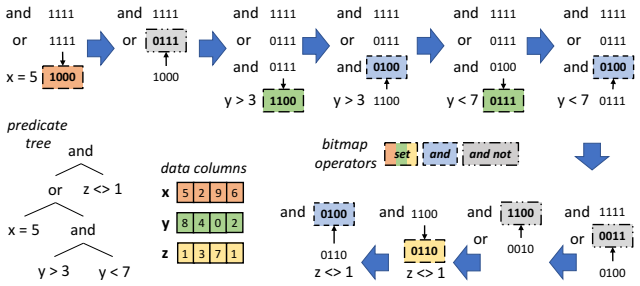

Figure 1: Selection scan example (without short-circuiting)

The output of the scan is a sorted list (i.e., array) of rids either used to access column values. In case of late materialization, we simply append rids. Materialization strategies are part of query optimization and depend on workload factors such as selectivity, cardinality, number of columns, and their data types, as well as the hardware factors such as memory bandwidth [1]. The VIP design is orthogonal to and can support multiple materialization strategies.

3.2 Compression

Modern analytical database systems evaluate selective predicates directly on compressed columns [39]. Dictionary encoding uses a sorted dictionary of distinct values per column and substitutes the column values with dictionary indexes using $\lceil \log n \rceil$ bits per index for n distinct values. There are many ways to store the bits of the indexes [38]; we use *horizontal bit packing* here [49]. In this scheme, we can use short-circuiting, although skipping complete cache lines is not guaranteed due to misalignment of the compressed data layout. The VIP sub-operator that unpacks the bits of dictionary indexes and evaluates the predicate without dereferencing the dictionary is shown below. We only need 5 SIMD instructions using 2 pre-computed permutation masks to unpack 16 32-bit indexes. The predicates are evaluated in SIMD registers directly after unpacking.

```
void select_int32_compressed(const void* data_in, [...] size_t dict_bits) {
  [...] // showing only the innermost loop for the ">" predicate on 16 tuples
  size_t j = _tzcnt_u64(m);  // compute the offset of 16 tuples out of 256
  __m512i x = _mm512_loadu_si512(data + dict_bits * 2 * j);  // load compressed
  // isolate the lower and upper 32-bit word per compressed index
  __m512i x1 = _mm512_permutevar_epi32(m_per_1, x);
  __m512i x2 = _mm512_permutevar_epi32(m_per_2, x);
  x1 = _mm512_srlv_epi32(x1, m_srl);  // align the lower 32-bit word
  x2 = _mm512_sllv_epi32(x2, m_sll);  // align the upper 32-bit word
  // merge the aligned lower and upper 32-bit word, and clear high-order bits
  x = _mm512_ternarylogic_epi32(x1, x2, m_max, 168);
  bitmap_out[j] = _mm512_cmpgt_epi32_mask(x, m_con); [...] }  // the predicate
```

For columns with many distinct values, dereferencing large dictionaries can be as expensive as a hash join. Thus, compression should be used when slow decompression is not required. Attributes that are compressed with the same dictionary can be joined without decompressing. On the other hand, compressing unique numeric values used in aggregate functions adds decompression overhead.

3.3 Hash Join

Hash joins are frequently optimized as a stand-alone operator using a key–rid pair. While we can achieve very good performance in such a setting, the results are misleading, primarily because they ignore the cost of late materialization [45]. Furthermore, if the join involves composite keys or the join key is not necessarily a 32-bit integer, many optimizations are either not applicable or are not nearly as efficient. In practice, we have to support multiple data types, composite keys, and additional non-equality predicates.

We outline how hash join operators work in the VIP design. First, we hash the columns that appear in equality predicates. We execute the hash join using the hash values instead of the actual columns. This allows us to map any data type into an integer including composite keys. After executing the hash join on the hash values, we generate rid lists for joined tuples pointing to the two inputs. We use the rids to evaluate the join predicates by accessing the actual columns, including non-equality predicates, and store back the rids of qualifiers. In the final phase, we can store the qualifying rids as the output of the join to dereference later, or eagerly materialize payload columns, similarly to materializing selection scan payloads.

When multiple SIMD lanes from the same scatter instruction write to the same memory location, conflicts occur. To avoid this hazard, we use special SIMD instructions that detect conflicts by performing all-to-all comparisons within the SIMD register. Without the special instructions, this step would require $O(W^2)$ scalar or $O(W)$ SIMD comparisons. Conflicts can also be detected using gathers and scatters [34], but the number of cache accesses increase.

The VIP sub-operator for building a hash table using the hash values of the hash join key columns is shown below. The bucket indexes are computed by masking the hash values, we do not rehash.

```
typedef struct { uint32_t hash; int32_t rid; } join_bucket_t;
void build_hashes(const uint32_t* hashes, size_t tuples,  // the hash values
    join_bucket_t* hash_table, size_t buckets) {  // the hash table
  const __m512i m_inc = _mm512_set_epi32(15,14,[...],0); [...]  // constants
  // scalar and SIMD registers holding the overall state of the function
  __m512i key, rid, loc;  size_t i = 0, j = 16;  __mmask16 k = 0xFFFF;
  while (i + j <= tuples) {  // process (up to) 16 hash values per iteration
    // replace finished SIMD lanes with new hash values from the input column
    key = _mm512_mask_expandloadu_epi32(key, k, &hashes[i]);
    __m512i inc = _mm512_add_epi32(m_inc, _mm512_set1_epi32(i));  i += j;
    rid = _mm512_mask_expand_epi32(rid, k, inc);  // generate inner rids
    loc = [...];  // compute the bucket location and gather hash table rids
    __m512i rid_H = _mm512_i32gather_epi32(loc, &hash_table[0].rid, 8);
    k = _mm512_cmplt_epi32_mask(rid_H, m_0);  // find empty hash table buckets
    __m512i con = _mm512_conflict_epi32(loc);  // detect conflicting lanes
    k = _mm512_mask_testn_epi32_mask(k, con, con);
    j = _mm_popcnt_u64(k);  // count lanes that can be reused
    // pack 32-bit keys and rids to 64-bit pairs and scatter to hash table
    __m512i buc_L = _mm512_permutex2var_epi32(key, m_pak_1, rid);
    __m512i buc_H = _mm512_permutex2var_epi32(key, m_pak_2, rid);
    _mm512_mask_i32loscatter_epi64(hash_table, k, loc, buc_L, 8);
    loc = _mm512_alignr_epi32(loc, loc, 8);
    _mm512_mask_i32loscatter_epi64(hash_table, k >> 8, loc, buc_H, 8); }
  [...] }  // build the last (15 or less) hash values using scalar code
```

The VIP sub-operator for probing the hash table is shown below. We cannot use hash table schemes that forbid key repetitions such as cuckoo hashing [32], even for foreign-key joins where unique inner keys are guaranteed, because inner keys may still be mapped to the same hash value that is used as the hash table key here. We can often guarantee no hash conflicts will occur by choosing the right hash function. For example, 32-bit FNV hash values never conflict if the total width of the input columns is 3 bytes or less.

```
size_t probe_hashes(const uint32_t* hashes, size_t tuples,  // the hash values
    const join_bucket_t* hash_table, size_t buckets,  // the hash table
    int32_t* inner_rids, int32_t* outer_rids) {  // inner and outer side rids
  [...] while (i + j <= tuples) {  // process (up to) 16 tuples per iteration
    // load new keys (hash values) from input while also reusing SIMD lanes
    key = _mm512_mask_expandloadu_epi32(key, k, &hashes[i]);
    __m512i inc = _mm512_add_epi32(m_inc, _mm512_set1_epi32(i));  i += j;
    rid = _mm512_mask_expand_epi32(rid, k, inc);  // generate outer rids
    loc = [...];  // gather hash table buckets of packed key-rid pairs
    __m512i buc_L = _mm512_i32logather_epi64(loc, hash_table, 8);
    loc = _mm512_alignr_epi32(loc, loc, 8);
    __m512i buc_H = _mm512_i32logather_epi64(loc, hash_table, 8);
    // unpack key-rid pairs to keys and (implicitly generated) rids
    __m512i key_H = _mm512_permutex2var_epi32(buc_L, m_unp_1, buc_H);
    __m512i rid_H = _mm512_permutex2var_epi32(buc_L, m_unp_2, buc_H);
    k = _mm512_cmpeq_epi32_mask(key, key_H);  // compare the keys
    _mm512_mask_compressstoreu_epi32(&outer_rids[o], k, rid);
    _mm512_mask_compressstoreu_epi32(&inner_rids[o], k, rid_H);
    o += _mm_popcnt_u64(k);  // append inner & outer rids
    k = _mm512_cmplt_epi32_mask(rid_H, m_0);  // detect empty bucket lanes
    j = _mm_popcnt_u64(k); }  // count empty bucket lanes to be replaced
  [...]  // process the last (15 or less) tuples using scalar code
  return o; }  // return the total number of matching tuples
```

In cases where we would rather probe the slower cache than partition the fact table [30], VIP can also support prefetching. In such a case, we would use a shared hash table built with atomics in scalar code; the cost of building is irrelevant. In other cases where we would prefer a partitioned hash join [29] to avoid cache misses, the hash tables are private per thread and no atomics are needed. After the hash probing, we evaluate the predicates using rid lists to dereference the columns. Hash collisions are rare because the hash join uses partitioning until the inner table fits in the cache. Also, the hash partitioning function is different from the hash function used for the join. For conjunctions of join predicates, a sub-operator evaluates each predicate and filters the rid lists. The hash collisions are resolved by evaluating the equality predicates. After evaluating all join predicates, we can materialize the rids or a set of payloads.

3.4 Partitioning

Partitioning can be used in hash join and group-by aggregation to avoid cache misses. In VIP, we store the partitioned output contiguously so we first compute a histogram. To compute the histogram, we load and hash the next block of tuples from the key columns, convert the hashes to partition ids based on the number of partitions, and increment the histogram counters. To avoid SIMD scatter conflicts, we replicate the histogram W times for W SIMD lanes [34]. The sub-operator to update the histograms is shown below.

```
void histogram(const uint32_t* hashes, int32_t* histogram_x16,
    uint8_t* pids, int bit_lo, int bit_hi) {  // bit range
  const __m512i m_1 = _mm512_set1_epi32(1); [...]
  for (size_t i = 0; i != tuples; i += 16) {  // 16 SIMD lanes
    __m512i p = _mm512_load_epi32(&hashes[i]);  // load hash values
    p = _mm512_srl_epi32(p, m_srl);  // get partition id from hash value
    p = _mm512_sll_epi32(p, m_sll);
    _mm_stream_si128(&pids[i], _mm512_cvtepi32_epi8(p));
    __m512i o = _mm512_add_epi32(p, m_rep);  // get offsets
    __m512i c = _mm512_i32gather_epi32(off, histogram_x16, 4);
    c = _mm512_add_epi32(c, m_1);  // increment the histogram counters
    _mm512_i32scatter_epi32(histogram_x16, o, c, 4); } [...] }
```

To pre-compute the boundaries of the partitioned output, we compute the prefix sum of the histograms across threads [43]. Then, we invoke the sub-operator shown below to compute the output offset of each tuple and store it in the cache. The output offsets will be reused by the sub-operators that shuffle each payload column and thus we avoid computing the output offsets multiple times.

```
void shuffle_core(const uint8_t* pids, size_t tuples,  // partition ids
    int32_t* partition_offset,  // the latest the output offset per partition
    int8_t* conflict_offset,  // compute the serialization offset per tuple
    int32_t* output_offset) { [...]  // compute the output offset per tuple
  for (size_t i = 0; i != tuples; i += 16) {  // 16 SIMD lanes
    // load partition ids and gather the output offset per partition
    __m512i p = _mm512_cvtepu8_epi32(_mm_load_si128(&pids[i]));
    __m512i o = _mm512_i32gather_epi32(p, partition_offset, 4);
    __m512i s = [...];  // serialize conflicts (9 instructions)
    _mm_store_si128(&conflict_offset[i],_mm512_cvtepi32_epi8(s));
    o = _mm512_add_epi32(o, s);  // update and store the offsets per tuple
    _mm512_store_epi32(&output_offset[i], off);  // output offset per tuple
    o = _mm512_add_epi32(o, m_1);  // update the per-partition output offsets
    _mm512_i32scatter_epi32(partition_offset, p, o, 4); } [...] }
```

Alongside the output offset, we store an offset to serialize scatter conflicts [34]. Specifically, if SIMD lanes i and j point to same partition p, the offset of lane i is o and the offset of lane j is $o + 1$ by adding the serialization offset of lane j which is 1. We use special conflict detection SIMD instructions to compute a bitmap of all-to-all lane conflicts, and then count the set bits per SIMD lane.

After computing the offsets, we invoke sub-operators to shuffle one column at a time for the next block of tuples. The output and serialization offsets are reloaded from the cache. Scattering the data to the output location directly results in too many cache conflicts [43]. To optimize the use of the cache during partitioning, we scatter the data in cache-resident buffers first and when the buffer of a partition is full, we flush data to output in a batch, while also using non-temporal stores to avoid polluting the cache [34, 36, 48].

The size of the buffer per partition is equal to two cache lines. Once the lower half of the buffer is full, we flush the lower cache line and shift the data from the upper. Depending on the column data type, we need a different number of values to fill the buffer of each partition. Since the sub-operators are specialized per data-type, this logic is hardcoded. In some cases, we optimize even within the same data type. For instance, long strings are horizontally shuffled one tuple per iteration using contiguous SIMD loads and stores, while short strings are vertically shuffled using gathers and scatters.

The 32-bit integer column shuffle sub-operator is shown below.

```
void shuffle_int32(const uint8_t* pids, size_t tuples,
    const int8_t* conflict_offsets, const int32_t* output_offsets,
    const int32_t* in, int32_t* buf, int32_t* out) { [...]
  for (size_t i = 0; i != tuples; i += 16) {  // 16 SIMD lanes
    __m512i o = _mm512_load_epi32(&output_offsets[i]);
    __m512i s = _mm512_cvtepu8_epi32(_mm_load_si128(&conflict_offsets[i]));
    o = _mm512_sub_epi32(o, s);  // remove serialization offset
    o = _mm512_and_epi32(o, m_15);  // determine buffer slot
    o = _mm512_add_epi32(o, s);  // add serialization offset
    __mmask16 k = _mm512_cmpeq_epi32_mask(o, m_15);  // partitions to flush
    __m512i p = _mm512_cvtepu8_epi32(_mm_load_si128(&pids[i]));
    o = _mm512_or_epi32(o, _mm512_slli_epi32(p, 5));  // offset in buffers
    __m512i v = _mm512_stream_load_si512(&in[i]);  // load data from column
    _mm512_i32scatter_epi32(buf, o, v, 4);  // store column data to buffers
    if (_mm512_kortestz(k, k)) continue;  // skip if no buffers are full
    uint64_t m = k;  // bitmask of lanes with full buffers to be flushed
    do {  // flush one full buffer at a time to memory using streaming stores
      size_t j = i + _tzcnt_u64(m);  // tuple location in input column
      size_t o = output_offsets[j];  // tuple location in partitioned output
      int32_t* b = &buf[pids[j] << 5];  // pick buffer to flush
      __m512i x1 = _mm512_load_si512(b);  // load buffer data
      __m512i x2 = _mm512_load_si512(b + 16);
      _mm512_stream_si512(&out[o - 15], x1);  // flush lower half of buffer
      _mm512_store_si512(b, x2);  // overwrite lower half with upper half
    } while (m = _blsr_u64(m)); }  // get next set bit of mask
  [...] }  // process the last (up to 15) tuples in scalar code
```

During a partitioned hash join, each thread partitions a portion of the input. If we need to split into many partitions that exceed the fanout that guarantees cache-conscious execution, we split into multiple passes like LSB radixsort [43]. Once we have more partitions than threads, we shuffle the data across threads, similar to a NUMA shuffling step [36]. Then, we can continue partitioning until the partitions of the inner table fits in the cache. In the final phase, each thread executes the final hash join for its local partitions. To find the boundaries of partition p out of 2^k partitions, we do not store the histograms explicitly. We use binary search over the join key columns and search for the first occurrence of hash p and $p + 1$ by recomputing the hash value on the fly and masking with $2^k - 1$.

3.5 Group-by Aggregation

In VIP, group-by aggregation has multiple steps, (i) estimate the number of groups, (ii) partition the input for cache-conscious execution, (iii) determine the group id (gid) per tuple, (iv) compute intermediate expressions, and (v) use the gids to update the partial aggregates. In contrast to joins where the smallest input size determines the number of partitions, the number of groups is estimated.

To estimate the number of groups, we use the PCSA algorithm [10] that can be split using fully vectorized VIP sub-operators. The core operation of PCSA is `b|=h&-h` where `b` is the PCSA bitmap and `h` is the hash value. To achieve cache-resident execution, we determine the number of partitions. If we do not partition the input, each thread processes the local portion of tuples and the partial aggregates are synchronously merged at the end. If we partition the input, we generate more partitions than threads and then assign each partition to be processed by the closest NUMA thread.

After the optional partitioning passes, the first step of the main group-by aggregation operator is to map the hash values of the group-by keys to gids. We use a hash table to store pairs of hash values and gids. Each group is initially identified by a unique hash value. We compute the hash value of the group-by columns for each input tuple and search two possible hash buckets in the cuckoo table for a matching hash value. If the hash value is not found, we create a new group, assign the next implicitly generated gid, and store the rid of the current tuple. Figure 2 shows an example.

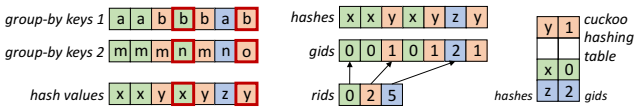

Figure 2: Example of mapping hashes to group ids (gids)

We show the VIP sub-operator that generates gids using the hash of the group-by columns. To mitigate complexity, we skip the part where we insert new groups into the table. Because inserting in a cuckoo hash table may inherently fail [32], or we may have underestimated the number of groups, we set a threshold that, if reached, causes the hash table to be resized and rebuilt from scratch.

```
typedef struct { uint32_t hash; int32_t gid; } aggr_bucket_t;
size_t hashes_to_gids(const uint32_t* hashes, int32_t* gids, size_t tuples,
    size_t groups, int32_t* rids, aggr_bucket_t* hash_table, int log_buckets) {
  const __m512i m_inc = _mm512_set_epi32(15,[...],1,0); [...]  // constants
  __m512i key, gid, loc;  __mmask16 k = 0xFFFF;  // initially use all lanes
  size_t i = 0, j = 16, g = 0, k = [...];  // hash table rebuild threshold
  while ((i = i + j) <= tuples) {  // 16 SIMD lanes
    if (--k == 0) { [...] }  // resize and rebuild the entire hash table
    key = _mm512_mask_loadu_epi32(key, k, &hashes[i - 16]);  // load hashes
    // compare the hash values across all lanes using conflict detection
    __m512i con = _mm512_conflict_epi32(key);
    __mmask16 k1 = _mm512_testn_epi32_mask(con, con);
    __m512i loc_1 = [...], loc_2 = [...];  // compute hash bucket locations
    // use alternative hash function for displaced tuples as per cuckoo hashing
    loc = _mm512_ternarylogic_epi32(loc, loc_1, loc_2, 150);
    loc = _mm512_mask_mov_epi32(loc, k1, loc_1);  // 1st hash bucket location
    // gather 16 hash buckets for lanes with unique hash
    __m512i buc_L, buc_H;  // load buckets from hash table
    buc_L = _mm512_mask_i32gather_epi64(buc_L, k3, hash_table, loc_1, 8);
    loc_1 = _mm512_alignr_epi32(loc_1, loc_1, 8);
    buc_H = _mm512_mask_i32gather_epi64(buc_H, k3 >> 8, hash_table, loc_1, 8);
    [...]  // unpack buckets to hashes (key_H) and gids (gid_H)
    // determine the lanes that need to gather the 2nd cuckoo hash bucket
    k2 = _mm512_mask_cmpge_epi32_mask(k1, gid_H, m_0);
    k2 = _mm512_mask_cmpneq_epi32_mask(k2, key, key_H);
    k2 = _mm512_kand(k2, k);  // 2nd hash bucket location
    loc = _mm512_mask_mov_epi32(loc, k2, loc_2);
    buc_L = _mm512_mask_i32logather_epi64(buc_L, k3, loc, hash_table, 8);
    loc = _mm512_alignr_epi32(loc, loc, 8);
    buc_H = _mm512_mask_i32logather_epi64(buc_H, k3 >> 8, loc, hash_table, 8);
    [...]  // re-unpack to hashes (key_H) and gids (gid_H)
    // find the leftmost lane with the same hash value in the SIMD register
    con = _mm512_and_epi32(con, _mm512_sub_epi32(m_0, con));
    con = _mm512_sub_epi32(m_31, _mm512_lzcnt_epi32(con));
    con = _mm512_mask_blend_epi32(k2, con, m_inc);
    // determine lanes with hashes that need to be scattered to the hash table
    k3 = _mm512_cmpge_epi32_mask(gid_H, m_0)
    k = _mm512_mask_cmpneq_epi32_mask(k3, key, key_H);
    k = _mm512_kor(_mm512_kand(k, k2), _mm512_kandn(k3, k2));
    if (!_mm512_kortestz(k, k)) {  // check if there are no new groups
      // copy gid from lestmost lane with same hash and store gids in order
      _mm512_storeu_epi32(&gids[i - 16], _mm512_permutevar_epi32(con, gid_H));
      j = 16, k = _mm512_kxnor(k, k);  // reuse all lanes
    } else { [...] }}  // create new gids, append rids, and update hash table
  [...] return g; }  // process the last tuples and return the number of groups
```

In contrast to joins where we cannot use cuckoo hashing, here we explicitly use cuckoo hashing in order to probe and map the hash values to gids in input order [34]. Mapping unique hashes to gids does not guarantee correctness due to collisions, even if collisions are rare after partitioning. To address collisions in joins, we re-evaluated the predicates, including the equalities. In aggregation, we use the rid of the first tuple per group to dereference the group-by columns and verify that the group-by columns within the same group match. If not, we create new groups. We scan each payload column once and compare the latest column value with the column value of the first tuple of the group. If the values do not match, we traverse a list of gid–rid pairs denoting distinct groups with the same hash and compare to other column values accessed via rid. If we find a match, we update the gid of the latest tuple. Otherwise, we append a new gid–rid pair to the list of groups with the same hash. In Figure 3, we show how to fix the collisions from Figure 2.

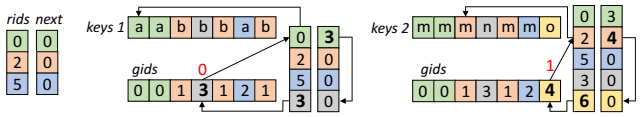

Figure 3: Example of resolving group-by hash collisions

We use type-specific sub-operators to resolve hash collisions of group by columns. In SIMD code, we load the group-by column and the gids sequentially. We gather the rids of the first tuple per group using the gids, and use the rid to gather the group-by column value determining the group. If the gathered value does not match the current value, we branch out and create a new group in scalar code.

To process the aggregate functions, we use the gids as direct mapping indexes. We compute functions across multiple columns block at a time. For example, for `sum(x*y)`, we compute `x*y` for the next block of tuples, store the intermediate result and then use it to update the partial sums. To avoid conflicts, we replicate the array of sums, since the sum is commutative, similar to histograms for partitioning. We show the sub-operator that computes the `min()` aggregate for an integer column below. If we have only one group or there is no group-by, we update the aggregate in registers.

```
void update_min_int32(const int32_t* data, const int32_t* gids,
    size_t tuples, float* min_x16, size_t groups) {
  const __m512i m_inc = _mm512_set_epi32(15,14,[...],1,0);
  for (size_t i = 0; i != tuples; i += 16) {  // 16 SIMD lanes
    __m512i val = _mm512_load_si512(&data[i]);  // load data from column
    __m512i gid = _mm512_loadu_si512(&gids[i]);  // load gids from cache
    // compute the offset of (replicated) partial aggregates from the gid
    __m512i loc = _mm512_or_epi32(_mm512_slli_epi32(gid, 4), m_inc);
    __m512i min = _mm512_i32gather_epi32(loc, min_x16, 4);  // load the min
    __mmask16 k = _mm512_cmplt_epi32_mask(val, min);  // store back if smaller
    _mm512_mask_i32scatter_epi32(min_x16, k, loc, val, 4); }}
```

If the input is partitioned, each thread processes distinct partitions. We keep the hash table of hashes to gids, the rids per group, and the partial aggregates in the cache, until we process all the tuples of the partition. Then, we merge the partial aggregates locally. If the input is not partitioned, we processes tuples block-at-a-time per thread and keep the partial aggregates in the cache. Then, we use a special-purpose hash table to link the partial aggregates across threads. Finally, we distribute and merge the partial aggregates.

4 EXPERIMENTAL EVALUATION

We perform all experiments on an Intel Xeon Phi 7210 Knights Landing CPU, based on the latest generation of many-core CPUs that rely on the most advanced AVX-512 SIMD instructions to maximize performance per core, allowing for a larger number of smaller cores per chip compared to mainstream CPUs. Our CPU has 64 cores running at 1.3 GHz with 4-way SMT and 16 GB of high-bandwidth on-chip MCDRAM memory with 295 GB/s load, 220 GB/s store, and 170 GB/s copy bandwidth. Our platform also has 192 GB of DDR4 DRAM with 70 GB/s load, 41 GB/s store, and 34 GB/s copy bandwidth. We compile using ICC 18 with -O3 optimization on a Linux 3.10 OS. In all experiments, we compare VIP against query-specific hand-optimized scalar code that emulates the state-of-the-art. The state-of-the-art achieves register-resident execution [31] by pipelining operators for one tuple at a time without function calls. This scalar register-resident row-at-a-time model of code-generating engines fully contrasts VIP, a SIMD interpreted engine processing tuples block-at-a-time. Compilation times are ignored, favoring the baseline, since VIP does not compile code at runtime.

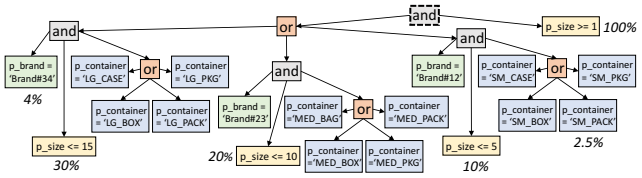

Figure 4: Selective predicate expression evaluation tree for selection on part table from TPC-H Q19 (0.24% selectivity)

To evaluate complex expressions, we pick the selection from TPC-H that combines the most predicates, the selection on table `part` from Q19. The expression tree in optimal evaluation order, based on selectivities, is shown in Figure 4 and is neither in CNF nor in DNF. In Figure 5, we show the selection throughput using both uncompressed and compressed data. The `p_brand` and `p_container` columns are `char(10)` with 25 and 40 distinct values respectively. The `p_size` column is a 32-bit integer with 50 distinct values. When compressed, we need 5, 6, and 5 bits respectively, reducing the footprint from 24 bytes to 2 bytes (16 bits) per tuple. The payload column is `p_partkey`, which is accessed for the qualifying 0.24% of tuples. We vary the scale factor (SF) of the dataset. For SF = 10000, the uncompressed data exceed the size of MCDRAM (HBW). VIP is 2.1–4.5X faster than the baseline and the best speedup is observed for scanning uncompressed data on MCDRAM. The large number of predicates highlights the efficiency of our design in handling complex expressions by handling bitmaps in the cache instead of compiling query-specific code. Since most Q19 predicates has high selectivity, short-circuiting predicates is crucial to performance, regardless of whether we process compressed data.

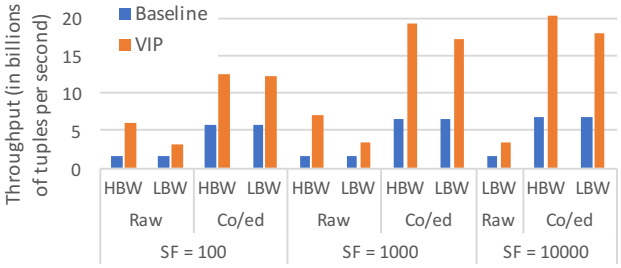

Figure 5: Selection on part table from TPC-H Q19

In Figure 6, we join the core tables of TPC-H. These queries, shown below, are at the core of most TPC-H queries with joins. The join payloads are the foreign keys used to join the core tables with the smaller dimension tables. On DRAM (LBW), we use fewer partitioning passes with larger fanout compared to MCDRAM (HBW).

```
select l_partkey, l_suppkey, o_custkey from lineitem, orders
 where l_orderkey = o_orderkey;
select l_orderkey, l_partkey, l_suppkey from lineitem, partsupp
 where l_partkey = ps_partkey and l_suppkey = ps_suppkey;
```

In the baseline, we materialize the payloads using a query-specific layout for the hash table buckets and we stop hash probing on the first match since the inner keys are unique. VIP supports partitioning until the inner table fits in the cache and execute the hash join in blocks of tuples to remain cache-resident. On fast memory (HBW), VIP is 1.8–2X faster for the join of table `lineitem` with `orders` and 1.6–1.7X faster for the join of `lineitem` with `partsupp`. The speedup is smaller on the join with `partsupp`, because in our design we evaluate each column of the composite key separately using

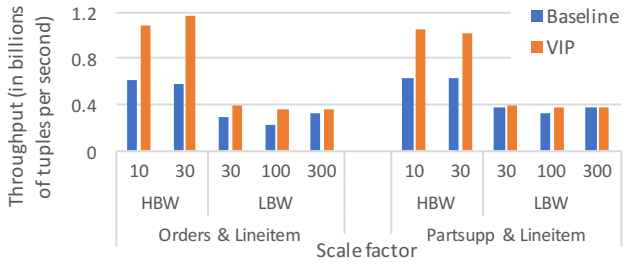

Figure 6: Hash joins using the largest tables from TPC-H

rids after joining on the hash values. On slow memory (LBW), the two methods are equivalent. The partitioned approach is memory-bound due to the number of passes while the baseline is bound by cache misses. The baseline is noticeably slower for SF = 100 because the hash table is slightly larger than the L2 cache and causes excessive cache conflicts. By increasing the hash table size by 4X, we can achieve the throughput that we show here for SF = 300.

To evaluate group-by aggregation in VIP, we use TPC-H Q1. For each column, we use the smallest data type that fits the value range. In the baseline, we process one tuple at a time and update private hash tables per thread. In VIP, we compute one expression at a time for the next block of tuples. In contrast to column-at-a-time execution [27] that materializes intermediate results after each operator on each column, each VIP operator never materializes intermediate results out of the cache. VIP can also reuse the results of common sub-expressions in the aggregate functions of Q1.

```
sum(l_extendedprice * (1 - l_discount)),
sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
```

As shown in Figure 7, VIP is 2.7–3.2X faster regardless of memory type. Estimating the groups is an order of magnitude faster here. Finally, we execute the same query without the group-by clause. Both methods now keep the partial aggregates in registers. In VIP, we compute each expression and aggregate separately for the next block of tuples. In the baseline, we compute all aggregates for one tuple a time at once. VIP is 3.6–4.3X faster on both memory types.

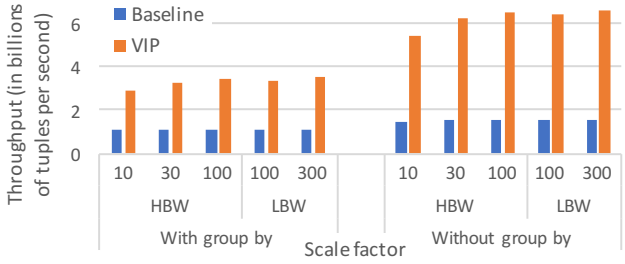

Figure 7: Group-by aggregation from TPC-H Q1

5 FUTURE WORK

Many-core CPUs, like the one used in out evaluation of VIP in this paper, relies on advanced SIMD instructions to achieve high performance. However, many-core CPUs and co-processors are not as common as mainstream CPUs and may not remain commercially viable in the future as a separate platform. We plan to evaluate the performance of VIP on the latest generation of mainstream CPUs that also support the AVX-512 SIMD instruction set and investigate adapting VIP to any additional features of new mainstream CPUs.

6 CONCLUSION

In this paper, we introduced VIP, a query engine designed and built bottom-up from pre-compiled data-parallel sub-operators and implemented entirely in SIMD. The VIP design can adapt to modern hardware features, such as utilizing the high bandwidth on-chip memory of many-core CPUs to facilitate cache-conscious execution. In contrast to earlier work that focuses on partial SIMD implementations of individual database operators such as sorting or hash joins, and assumes a favorable input setup such as key–rid pairs with a specific materialization strategy, the VIP engine supports all fundamental database operators, namely selections, hash joins, and group-by aggregations, with any number of columns, multiple data types, compression, and complex predicates or expressions.

Using the latest generation of many-core CPUs with the latest AVX-512 SIMD instructions, we show that VIP outperforms query-specific hand-optimized code emulating the state-of-the-art code-generating query engines, without including the runtime compilation overhead of the state-of-the-art that is non-existent in VIP. Overall, VIP is a step towards realistic query execution designed on top of advanced SIMD vectorization and taking advantage of additional advanced hardware features provided by modern CPUs.

REFERENCES

[1] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, pages 466–475, 2007.

[2] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multicore, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, Sept. 2013.

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

[4] S. Blanas, Y. Li, and J. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, pages 37–48, 2011.

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

[6] X. Cheng, B. He, X. Du, and C. T. Lau. A study of main-memory hash joins on many-core processor: A case with Intel Knights Landing architecture. In *CIKM*, pages 657–666, 2017.

[7] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *VLDB*, pages 1313–1324, 2008.

[8] A. Costea, A. Ionescu, B. Răducanu, M. Switakowski, C. Bârca, J. Sompolski, A. Luszczak, M. Szafrański, G. de Nijs, and P. Boncz. Vectorh: Taking SQL-on-Hadoop to the next level. In *SIGMOD*, pages 1105–1117, 2016.

[9] 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. In *SIGMOD*, pages 215–226, 2016.

[10] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, Sept. 1985.

[11] G. Fowler, L. C. Noll, K.-P. Vo, and D. Eastlake. The FNV non-cryptographic hash algorithm. Technical report, 2017. http://www.ietf.org/internet-drafts/draft-eastlake-fnv-13.txt.

[12] G. Graefe. Volcano: An extensible and parallel query evaluation system. *TKDE*, 6(1):120–135, Feb. 1994.

[13] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923, 2015.

[14] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, pages 189–198, 2007.

[15] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, Nov. 2014.

[16] H. Inoue and K. Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 8(11):1274–1285, July 2015.

[17] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *PVLDB*, 8(6):642–653, Feb. 2015.

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

[19] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, Aug. 2009.

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

[21] H. Lang, A. Kipf, L. Passing, P. Boncz, T. Neumann, and A. Kemper. Make the most out of your simd investments: Counter control flow divergence in compiled query pipelines. In *DaMoN*, 2018.

[22] 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 *SIGMOD*, pages 311–326, 2016.

[23] H. Lang, T. Neumann, A. Kemper, and P. Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *PVLDB*, 12(5):502–515, Jan. 2019.

[24] 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.

[25] Y. Li and J. M. Patel. BitWeaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[26] Y. Li and J. M. Patel. WideTable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, June 2014.

[27] S. Manegold, P. Boncz, and M. Kersten. Optimizing database architecture for the new bottleneck: memory access. *J. VLDB*, 9(3):231–246, 2000.

[28] S. Manegold, P. Boncz, and M. Kersten. What happens during a join? dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[29] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *TKDE*, 14(4):709–730, July 2002.

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

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

[32] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[33] 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, Oct. 2016.

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

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

[36] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014.

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

[40] K. A. Ross. Selection conditions in main memory. *TODS*, 29(1):132–161, Mar. 2004.

[41] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, 2007.

[42] P. Roy, J. Teubner, and G. Alonso. Efficient frequent item counting in multi-core hardware. In *KDD*, pages 1451–1459, 2012.

[43] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[44] B. Schlegel, T. Karnagel, T. Kiefer, and W. Lehner. Scalable frequent itemset mining on many-core processors. In *DaMoN*, 2013.

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

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

[47] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[48] J. Wassenberg and P. Sanders. Engineering a multi core radix sort. In *EuroPar*, pages 160–169, 2011.

[49] 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, Aug. 2009.

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