Exploiting Data Skew for Improved Query Performance

Wangda Zhang and Kenneth A. Ross

Abstract—Analytic queries enable sophisticated large-scale data analysis within many commercial, scientific and medical domains today. Data skew is a ubiquitous feature of these real-world domains. In a retail database, some products are typically much more popular than others. In a text database, word frequencies follow a Zipf distribution with a small number of very common words, and a long tail of infrequent words. In a geographic database, some regions have much higher populations (and therefore data measurements) than others. Current systems do not make the most of caches for exploiting skew. In particular, a whole cache line may remain cache resident even though only a small part of the cache line corresponds to a popular data item. In this article, we propose a novel index structure for repositioning data items to concentrate popular items into the same cache lines. The net result is better spatial locality, and better utilization of limited cache resources. We develop a theoretical model for analyzing the cache utilization, and implement database operators that are efficient in the presence of skew. Our experimental evaluation on real and synthetic data shows that exploiting skew can significantly improve in-memory query performance. In some cases, our techniques can speed up queries by over an order of magnitude.

Index Terms—Data skew, query processing, permutation index, cache optimization, SIMD

1 Introduction

In online analytic processing (OLAP) a user executes a collection of complex queries over large data sets, in order to understand the data at hand and to obtain actionable knowledge. With the increasing main-memory capacity of contemporary hardware, query execution can occur entirely in RAM. Analytical query workloads that are typically read-only need no disk access after the initial load. In response to this trend, several commercial and research database management systems have been designed (or re-designed) for memory-resident data [1]. Examples of recent systems include H-Store/VoltDB [2], Hekaton [3], HyPer [4], IBM BLINK [5], DB2 BLU [6], SAP HANA [7], Vectorwise [8], Oracle TimesTen [9], MonetDB [10], HYRISE [11], HIQUE [12], LegoBase [13], Peloton [14], and Quickstep [15]. Most analytic database systems use some variant of columnar storage, since only the columns needed to answer the query need to be read [16].

Skew is a common feature of many real-world domains. Power-law distributions apply to many types of data, including word-usage in text databases, protein interactions, internet routing node-degree, phone-call data, city populations, email contact-lists, surname frequencies, and paper citations [17]. Several other kinds of skewed distributions can also be found [18]. Fig. 1a shows the cumulative distribution of book reviews in the publicly available Amazon reviews dataset, where books are ranked by the number of reviews. The distribution is likely to be a representative proxy of the actual sales data. As shown, the 100,000 most popular book titles ($< 5\%$) account for roughly 50 percent of the entire data, and 75 percent of the sales concentrate on the top 500,000 titles.

In the context of a data warehouse, skew is likely to affect query performance. Consider a large fact table corresponding to the sales of a bookseller. One of the columns of this fact table is the id of a book, represented as an integer foreign key `bid` into a dimension table of detailed book information. Skew as shown in Fig. 1a in the `bid` column would be expected due to different popularity. The following example queries utilize `bid` in an important way.

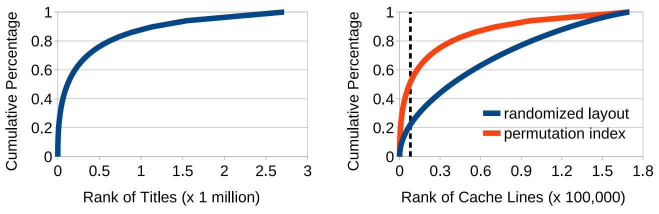
```
-- Q1: Consult a column of the dimension table
Select B.price
From   Sales S, Books B
Where  S.bid = B.bid
```

Query `Q1` executes a foreign key join to read or materialize a column from the dimension table. The obtained prices can then be used to calculate revenues. To answer this query, the database system can scan the fact table `Sales`, and look up the `price` column of table `Books` using `S.bid` as an array offset. If the number of products is such that the `price` column is larger than the cache in size, and `S.bid`s are uniformly distributed in the dimension domain, then this lookup could incur expensive cache misses due to unclustered memory accesses, which can often be the dominant cost [19]. With skew in `S.bid`, the situation is somewhat better because the most frequently occurring items are likely to reside in the cache. Nevertheless, the cache is still underutilized because a single cache-resident cache line will typically hold a small number of popular items and many unpopular items.

```
-- Q2: Filter the fact table based on B.price
Select S.bid
From   Sales S, Books B
Where  S.bid = B.bid and B.price < 100
```

• The authors are with Columbia University, New York, NY 10027 USA. E-mail: {zwd, kar}@cs.columbia.edu.

Manuscript received 24 Oct. 2019; revised 14 May 2020; accepted 25 June 2020. Date of publication 1 July 2020; date of current version 1 Apr. 2022. (Corresponding author: Wangda Zhang.) Recommended for acceptance by A. Bouguettaya. Digital Object Identifier no. 10.1109/TKDE.2020.3006446



(a) Distribution of book titles (b) Access to cache lines (`Q1`)

Fig. 1. An example of data skew in a real dataset.

To process this selective filtering, the database system can preprocess the dimension table to determine which products meet the `B.price<100` condition. It can then create a bitmap indexed by `bid` that can be consulted using `S.bid` as an offset. `Q2` has similar cache miss issues to `Q1`, except that the data being consulted is one bit per `bid` rather than several bytes. As a result, the product cardinality thresholds for cache-residence will be larger under the same cache capacity.

```
--Q3: Aggregate grouped by book ids
Select  bid, count(*)
From    Sales
Group by bid
```

For this query, the database system can create a table of counts, indexed by `bid`, and update the count for each corresponding fact table record in turn. The memory access pattern (and cache behavior) is similar to `Q1`, with the added observation that because cache lines are updated, cache-line replacement triggers some additional memory-write traffic.

```
--Q3a: Heavy hitter counts
Select  bid, count(*)
From    Sales
Group by bid
Order by count(*) desc
Limit   4000
```

Query `Q3a` is similar to `Q3` except that we are only interested in the counts of heavy hitters, which for this query means the `bid`s with a count among the top 4,000 book titles. Because the cache footprint of the heavy hitters is much smaller than the entire `bid` domain, there are opportunities for further performance enhancement if the candidate heavy hitters can be identified in advance.

For these core database operations, skew-aware data management is not adequately considered in existing main memory database architectures. In this paper, we propose a novel index structure called a permutation index for reordering data items by their access frequency (Section 3). Under skewed data distribution, popular data items are concentrated into common cache lines using permutation indexes, leading to improved locality for query processing. By carefully organizing data at the cache line level, we can exploit the data skew for better performance, which has not be utilized by previous research.

The permutation index method is simple, yet very effective at improving the cache utilization. Fig. 1b illustrates the cumulative frequency of cache line accesses for the example book dataset during query `Q1`. Under a randomized data layout, the most popular book titles are sparsely distributed, so the distribution of cache line accesses is smoothed out. Even with significant data skew, the most frequent 8,000 cache lines (about the size of L1 cache using 4-byte `bid`s) only correspond to 22 percent of the fact table data (see the dashed line). With the permutation index, frequent data are concentrated and the percentage increases to 52 percent.

Built on permutation indexes, we develop efficient database operators in the presence of skew. Permutation indexes provide frequency information about data items, allowing for threshold-based algorithms that execute different code paths for items with different degrees of skew. To take full advantage of modern architectures, our implementation makes use of single-instruction multiple-data (SIMD) instruction sets, multithreaded execution, and software prefetching (Section 4).

Finally, we present a detailed experimental evaluation of our techniques in Section 5. We conduct experiments on Intel Skylake and Xeon Phi processors, using both synthetic microbenchmarks and real data sets. Our results show that exploiting skew and reordering data can significantly improve performance, making queries using permutation indexes up to an order of magnitude faster.

2 Background

2.1 Data Skew and Representation

As introduced in Section 1, a wide variety of real-world phenomena approximately follow skewed distributions [17], [18]. While the techniques described in this paper apply to any skewed distribution, for ease of exposition, we focus on Zipf distributions in our microbenchmark study because they are common and allow one to model the degree of skew with a single parameter z. In a Zipf distribution, the frequency of the data item having rank r is proportional to r^{-z}. $z = 0$ corresponds to a uniform distribution, while many real-world skewed data sets can be modeled by Zipf distributions with $z \approx 1$ [17], [20].

We assume a columnar format as is common in analytic databases. An individual column is represented as a dense array of integers. The compact integer representations can be used as data values and/or as foreign key offsets into dimension tables stored as a collection of column arrays.

2.2 Architectural Issues

Single Instruction Multiple Data. Modern processors support single-instruction multiple-data (SIMD) instruction sets, which process many data items at a time, enhancing the data-parallelism of algorithms that can be written in a SIMD fashion. Further, SIMD instructions convert control dependencies to data dependencies, helping to eliminate branch misprediction latencies [21]. Nowadays mainstream CPUs (e.g., Skylake [22]) and Xeon Phi processors (Knights Landing [23]) support the 512-bit extensions (AVX-512).

Cache and TLB Performance. Modern architectures provide multiple levels of data and instruction caches. These small but fast memories improve performance when algorithms display spatial or temporal locality. Conversely, algorithms that ignore the caches (e.g., by randomly accessing data structures exceeding the cache size) incur many cache misses

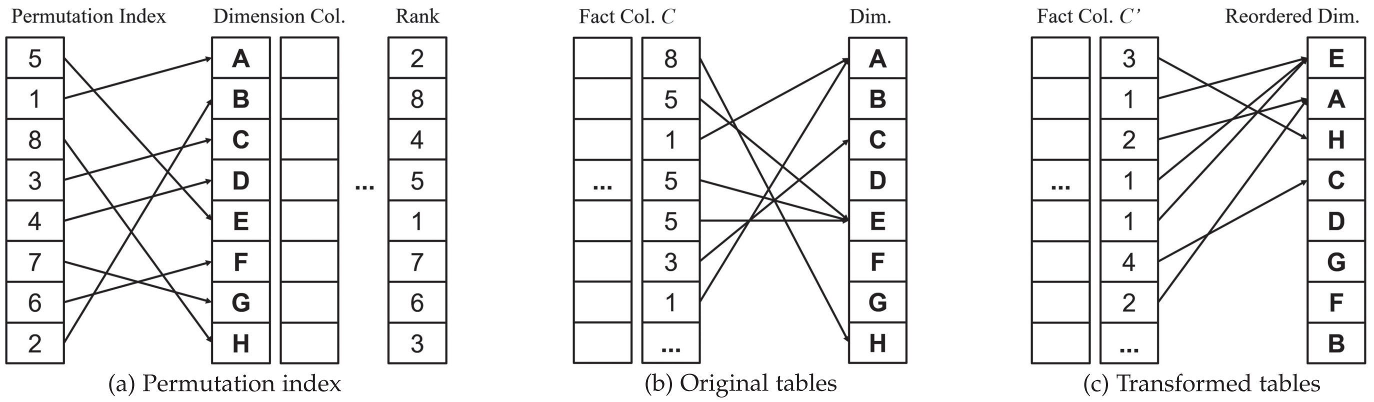


Fig. 2. (a) Representations of a permutation index, (b) a foreign key column C before applying the permutation index transformation, and (c) the same column C' with new identifiers after the transformation.

and/or TLB (i.e., translation lookaside buffer) misses that can degrade performance by an order of magnitude. Even when data fits into cache, TLB thrashing can still occur if the working sets are fragmented. Therefore, addressing cache and TLB performance is a necessity for a database management system given that most data structures for representing the underlying data will be much larger than the data caches. When access patterns are deterministic, prefetching can hide some of the memory access latency. For common patterns such as sequential access, the hardware can automatically prefetch data that is soon needed.

2.3 Cache Performance Optimizations

For the example queries of Section 1, some existing techniques could be used to improve performance by optimizing the cache behavior. One approach is to measure the footprint of the accessed data column to see whether it is larger than the performance-critical cache. If so, a range-partitioning pass over the input `bid` references (and their fact table payloads for `Q1` and `Q2`) can redistribute the data into fragments. With a sufficient partitioning factor, each fragment will reference a subset of `bid`s that fits into the cache. As long as the partitioning pass is done efficiently with mostly-sequential data accesses and few cache misses [24], the overhead of partitioning may be smaller than the gain from avoiding cache misses. On the other hand, multiple passes through the very large sales table would be needed, meaning that the overhead is nontrivial even if there is a performance improvement.

An alternative approach is to use software prefetching to overlap the latencies of multiple cache misses [25]. A prefetching distance d is determined empirically, and the `bid` value for the record d steps ahead of the current record is prefetched into the cache using machine-specific prefetch instructions. The hope is that the cache miss latency is paid while the processor is doing useful work on other items. Modern CPUs can have as many as 10 outstanding memory requests per core, allowing many prefetches to be in flight at once.

While prefetching helps by overlapping latencies, it does not completely eliminate cache miss effects in workloads that are memory-bandwidth bound. In the queries of Section 1, there is limited work that can be done while waiting for the miss to resolve. Therefore, there remains an opportunity for methods that reduce the total volume of data that needs to be brought into the cache from slow memory.

3 Permutation Indexes

To address the cache utilization problem as outlined in Section 1, we propose a novel index structure that we call a *permutation index*. We start by identifying a fact table column C with an integer data type. For simplicity we assume that the integer column is an offset into a dimension table as in the example queries. Let $|C|$ denote the number of distinct values appearing in column C.

The permutation index is an array of size $|C|$ containing the values appearing in column C in decreasing frequency order. The left part of Fig. 2a shows an example in which 5 is the most frequent value in the referencing column (Fig. 2b), then $1, 8, 3$, etc. The arrows in Fig. 2a show how these integers may be interpreted as offsets into a dimension table (starting from 1). The rightmost column in Fig. 2a shows an equivalent representation of the same information in which the popularity rank of each distinct data item in C is stored as an additional column of the dimension table. If there are dimension table offsets that do not appear in C, then the offset representation of Fig. 2a (left) will be slightly more compact than rank representation (right) because no information needs to be stored for the missing values.

Note that the array representation of a permutation index using $|C| \log |C|$ bits is already within $O(|C|)$ bits of the information theoretic bound of $\log(|C|!)$ [26]. To support efficient inverse operations, we can store a permutation index using compressed data structures for permutations, so that both lookups and reverse-lookups take sublinear time [26], [27]. Alternatively, we can store both the offset and rank representations, so that lookups in either direction take only constant time.

3.1 Index Building and Maintenance

To achieve a better access pattern, we must recode the references in column C into a new version with the permuted references. For the data of Fig. 2b, references to dimension table row i are replaced by references to row $Rank(i)$ according to the rank representation of the permutation index in Fig. 2a, to obtain column C' in Fig. 2c. For example, 8 (i.e., the first item) in C is replaced by 3 in C', since

$Rank(8) = 3$. Similarly, $Rank(5) = 1$, so all 5s in column C are replaced by 1s in the transformed C'.

Building the permutation index can be implemented as an aggregation query (Q3) to obtain the frequency of distinct dimension values, taking linear time, followed by sorting the values by their frequencies. The sorting step takes super-linear time but it executes over the much smaller dimension domain. Then, the identifier replacement in a fact table can be implemented as a materialization query (Q1) using the permutation index we just built, again taking linear time. After this transformation, the old column C (Fig. 2b) is dropped, and replaced with a new column C' with the new identifiers $Rank(i)$.

We assume that this replacement in the fact table has occurred prior to the query execution, typically when the database administrator (or automated physical design optimizer) decided to build the permutation index. Replacement of one integer value with another yields no net change in space for column C. The space cost of building the permutation index is one integer value per dimension table row, and is independent of the size of column C of the (presumably large) referencing column in the fact table.

Updates to the permutation index can be handled without major reorganization by simply ignoring the effect of fact table insertions/deletions on the frequencies of existing domain values. When a new dimension table row is inserted, it is appended to the end of the dimension table, and its identifier is appended to the end of the permutation index as the new least-frequent item. A small number of updates is unlikely to dramatically change the relative ordering of popular value frequencies from a large fact table, so these simple choices will preserve most of the performance advantages of the structure even after a few updates. After many updates (e.g., a batch update in a data warehouse) the index should be rebuilt.

3.2 Overview of Query Processing

The permutation index functions as a reordering template for preprocessing the dimension table during the initial phases of query processing. We create a copy of the needed dimension table columns for a query, permuted according to the corresponding permutation index. For Fig. 2a, assuming we needed the first dimension column, we would create an intermediate result in the order E,A,H,C,D,G,F,B, as shown in Fig. 2c. Because the cost of the query is likely to be dominated by the large fact table containing the source column C, preprocessing the dimension table in this way will be relatively fast. Note that a dimension table may have several permutation indexes that refer to it from different source columns, each with different orders. If the dimension table is referenced just from a single source, then its reordering can be done entirely ahead of query processing.

Having reordered the data, popular items are adjacent and therefore are likely to share cache lines with other popular items (Fig. 2c). Therefore, during query processing if we access the data via offsets in the reordered table, we will get much better utilization of all cache levels, particularly when there is skew in an otherwise large domain. As we shall discuss in detail in Section 4, several basic database operations can make use of a permutation index and the reordered data to improve efficiency (in different ways). Since a permutation index simply replaces a fact table column and adds a copy of the dimension column, there is minimal impact on other database queries.

Using permutation indexes over skewed data essentially reorders data to change the memory access pattern during query execution. Physically reordering the dimension table by access frequency would eliminate the need for an explicit permutation index. However, we can impose only one ordering on the dimension table and there may be many competing demands (e.g., ordering by a domain value to support lexicographic comparisions via identifiers, or additional references from fact columns having different skew properties). A similar observation holds for the fact table, where reordering rows might help locality in the ordered columns, but there can be only one order unless data is stored redundantly. In light of these observations, one can think of permutation indexes as a way to influence physical database design. For some skewed data columns, permutation indexes eliminate the need for physical ordering of the fact table by those columns in order to get good cache performance. Physical organization can then focus on the remaining columns that may have more serious nonlocality.

3.3 Cache Behavior Analysis

To decide whether to build a permutation index or not, we need to estimate the improvement in cache utilization. A sophisticated query optimizer also requires knowledge of the cache behavior to compute the cost of query execution, together with hardware characteristics. For these purposes, we now study a model for estimating the cache hit rates under different data layouts.

Prior work has developed analytical cache models based on stack distances [28], [29]. The stack distance is the number of distinct cache lines referred between two references to the same line. For an LRU cache with S lines, accesses with stack distance less than S will be cache hits, while others are misses. To apply this model in our setting, we need an estimation of access frequency to the cache lines, and the stack distance distribution for each line.

Conventional databases store statistics about column distributions to help estimate the selectivities of query conditions. It is also possible to sample the data and fit the samples to known distributions [30]. Most power-law distributions can be modeled with a few parameters, such as the slope and intercept in a log-log plot. Given the value frequency distribution, we can then map it to the cache line distribution for a particular layout. For example, using permutation indexes over 4-byte values, the access frequency of the most frequent 64-byte cache line would be the sum of the 16 largest frequencies in the value distribution. For a randomized mapping from value distribution to cache line distribution, the computed frequency would be an approximation.

Let f_i denote the access frequency to cache line L_i, where $0 \le f_i \le 1$ and $\sum_i f_i = 1$. For line L_i, we model its next reference as a geometric distribution with probability $p = f_i$. Suppose after k trials, we see cache line L_i. According to the stack distance model, if there are fewer than S distinct lines occurred within the k trials, then the reference will be a cache hit. Due to data skew (espectially the very frequent lines), there are potentially repeated occurrences of the same cache lines within the k trials. For every line L_j where

$j \neq i$, its expected number of occurrences is

$$n_j = k * f_j/(1 - f_i).$$

If $n_j > 1$, then there are $(n_j - 1)$ repeated occurrences of L_j. Thus, an estimation of the number of distinct lines is

$$d = k - \sum_{j \neq i} \max(0, n_j - 1).$$

As k increases, d also increases (at a lower rate).

For cache line L_i, we want to find a threshold K so that after K trials, we see L_i and there are estimated $d = S$ distinct lines $L_{j \neq i}$ seen within the K trials. Then we know for any $k < K$, we have $d < S$ so the access will be a cache hit. To compute the threshold K, a binary search can be used where K is at least S. Modeled as a geometric distribution, the cumulative distribution (CDF) of $(k < K)$ for L_i is

$$cdf_i = 1 - (1 - f_i)^K.$$

Therefore, the overall estimated cache hit rate is $\sum_i (f_i \cdot cdf_i)$.

Empirically we find this model is accurate with errors less than 5 percent on our microbenchmarks given that we have a good estimation of the cache line frequencies. In practice, we can also sample the cache line distribution directly to verify the accuracy of the model and adjust accordingly if there is partial clustering, which we plan to address as future work.

4 Skew-Aware Operator Implementation

We now discuss how to implement efficient database operators that can take advantage of permutation indexes and the reordered data. As introduced in Section 1, we focus on three types of basic database operations: materialization (Q1), selection (Q2), and aggregation (Q3 and Q3a).

The benefits of the proposed permutation index approach are twofold. First, reordering data using permutation indexes improves cache utilization during execution of important database operations (Section 3). Second, the transformed identifiers in the fact table (Fig. 2c) provide valuable information about data frequency, which the operators can exploit to perform threshold-based processing (Section 4.2). In this way, the operators are "aware" of the degree of skew, and are able to take appropriate code paths for different actions.

We find that even a straightforward scalar implementation of these operators can take advantage of the improved cache utilization (Section 5.3). To further enhance performance, we study the use of a variety of techniques including SIMD vectorization (Sections 4.1 & 4.2), multithreaded execution (Section 4.3), and software prefetching (Section 5.1).

4.1 Data-Parallel Execution

Data-parallel execution using SIMD instructions has been successful at speeding up various database operations [21], [31], [32], [33], [34], especially when the data is cache resident. For data-parallel read and write, we use AVX-512 gather and scatter instructions extensively. Most AVX-512 instructions provide variants for other data types including 8, 16, and 64-bit integers as well. For simplicity of presentation, we assume a database column is simply an array of 32-bit integers, and provide pseudocode using corresponding instrinsics in this section.

```
void materialize(const size_t num_tuples, const __m512i* in,
                 const uint32_t* data, __m512i* out) {
  for (size_t i = 0; i < num_tuples / 16; ++i) {
    #ifdef ENABLE_PREFETCH
    _mm512_prefetch_i32gather_ps(in[i + PREFETCH_DISTANCE],
                                 data, 4, PREFETCH_HINT);
    #endif
    out[i] = _mm512_i32gather_epi32(in[i], data, 4);
  }
}
void select(const size_t num_tuples, const __m512i* in,
            const uint32_t* bitmap, uint32_t* out) {
  uint32_t k = 0;
  for (uint32_t i = 0; i < num_tuples / 16; ++i) {
    // Compute ((bitmap[pos/32] >> (pos%32)) & 1).
    __m512i pos = in[i];
    __m512i bidx = _mm512_srli_epi32(pos, 5);
    __m512i g = _mm512_i32gather_epi32(bidx, bitmap, 4);
    __m512i bit_shifts = _mm512_and_si512(pos, ALL_31);
    __m512i shifted = _mm512_srlv_epi32(g, bit_shifts);
    __m512i bits = _mm512_and_si512(shifted, ALL_1);
    __mmask16 mask = _mm512_cmpneq_epi32_mask(bits, ALL_0);
    _mm512_mask_compressstoreu_epi32(&out[k], mask, pos);
    k += _mm_popcnt_u32(mask);
  }
}
void aggregate(const size_t input_size, const uint32_t* in,
               uint32_t* out) {
  __m512i pos;
  __mmask16 mask = -1;
  for (size_t i = 0; i < input_size; ) {
    __m512i new_pos = _mm512_loadu_si512(&in[i]);
    pos = _mm512_mask_compress_epi32(new_pos, ~mask, pos);
    __m512i cnt_in = _mm512_i32gather_epi32(pos, out, 4)};
    __m512i conflicts = _mm512_conflict_epi32(pos);
    mask = _mm512_testn_epi32_mask(conflicts, ALL_NEG_1);
    __m512i cnt_out{_mm512_add_epi32(cnt_in, ALL_1)};
    _mm512_mask_i32scatter_epi32(out, mask, pos, cnt_out, 4);
    i += _mm_popcnt_u32(mask);
  }
  ... // Handle remaining data in scalar code.
}
```

Materialization. The operator uses the fact table values as indexes to perform gather instructions from the dimension data array. Given 32-bit integer values, an AVX-512 gather instruction retrieves 16 dimension values at once. The latency of this gather operation depends directly on the number of cache misses occurred (Section 4.2.1).

Selection. This operation produces an array of qualifying fact table offsets. After preprocessing the dimension table using the selection condition to obtain a bitmap, the operator checks each row of the fact table against the bitmap using the referencing identifier as the index, writing out the row offset in the fact table if the bitmap testing succeeds (the scalar implementation needs to be branch-free to avoid the branch misprediction penalty). Whenever the operator needs to test a random bit, reads from the bitmap incur random memory accesses, similar to materialization. For SIMD, we use gather and shift instructions to compute addresses in the bitmap and to extract the bits into a mask. Using the AVX-512 compressed store instruction `vpcompressd`, we can contiguously store the selected fact table offsets (those with their respective bits set in the mask) into an output array.

Aggregation. This operation generates an array of numeric types to compute an aggregate of some fact table column, grouped by the dimension table offset. The implementation basically scatters into an output array after gathering old aggregates and performing arithmetic computations. In a scalar implementation, the memory access pattern is similar to materialization with additional writes.

SIMD aggregation, however, needs an additional step to check for conflicts before scattering, since different

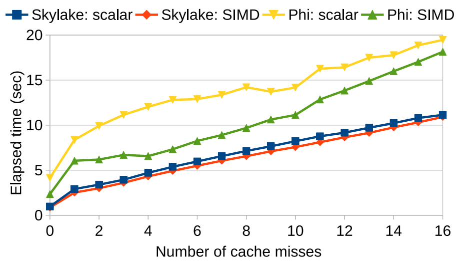


Fig. 3. Materialization performance with varying cache misses per 16 elements.

SIMD lanes may write to the same memory location. When internal conflicts occur, we identify a subset of conflict-free SIMD lanes using an AVX-512 conflict detection instruction `vpconflictd`, and allow only writes from this subset to succeed using the masked scatter instruction. The other SIMD lanes are retained for processing during the next iteration, alongside new data. If there are many conflicts (e.g., when fact table data is highly skewed), then performance deteriorates severely due to this conflict resolution step. In the worst case where all SIMD lanes attempt to update the same aggregate value, only one (instead of 16 assuming 4-byte integers) can proceed. We shall discuss this problem further in Section 4.2.2. Note that these SIMD implementations do not rely on permutation indexes. Data-parallel optimizations and cache locality optimizations are orthogonal, working together to further boost performance.

4.2 Threshold-Based Processing

An important insight in the use of permutation indexes is that the transformed identifier in the fact table (see Fig. 2c) can be used as a proxy for the value frequency in the table. Given an estimate of the function mapping the numeric identifier to the value frequency, the system can estimate the likely cache behavior at each level using the model in Section 3.3. In general, the most frequently accessed data are likely cached under common cache replacement policies. Given the available cache capacity, it is possible to derive a threshold t such that accesses to data items with identifiers smaller than t are likely cache hits, while identifiers larger than t lead to misses. We can therefore choose different code paths for items based on their anticipated cache residence using a simple comparison between the identifiers and the threshold t.

4.2.1 Materialization

One situation where such an optimization pays off is when most, but not all data references are cache hits. Consider a materialization query like `Q1` in which roughly one in 16 accesses is a cache miss, and the other 15 accesses are cache hits. A profile of the performance of materialization, with data constructed to achieve specific cache miss rates, is shown in Fig. 3. For this profiling, we use 128 million dimension keys and 1 billion fact table tuples (the default microbenchmark setting in Section 5.1). Using single-thread implementations of both scalar and SIMD versions described in Section 4.1, we measured the query latency on a Skylake processor and a Xeon Phi processor.

```
void materialize_split(const size_t num_tuples,
    const __m512i* in, const uint32_t* data, __m512i* out) {
  __m512i out_pos = _mm512_set_epi32(15, 14, ... , 0);
  size_t cnt = 0;
  for (size_t i = 0; i < num_tuples / 16; ++i) {
    __m512i idx = in[i];
    // Compute mask for out-of-cache accesses.
    __mmask16 mask = _mm512_cmpge_epu32_mask(
        idx, _mm512_set1_epi32(THRESHOLD));
    // Resolve cache hits immediately.
    out[i] = _mm512_mask_i32gather_epi32(ALL_0,
        _mm512_knot(mask), idx, data, 4);
    // Store extra info for unresolved data in buffers.
    _mm512_mask_compressstoreu_epi32(
        &idx_buf[cnt], mask, idx);  // unresolved indexes
    _mm512_mask_compressstoreu_epi32(
        &pos_buf[cnt], mask, out_pos);  // output positions
    // Update the count of out-of-cache accesses.
    cnt += _mm_popcnt_u32(mask);
    // Update output positions.
    out_pos = _mm512_add_epi32(out_pos, ALL_16);
  }
  // Resolve out-of-cache accesses.
  for (size_t i = 0; i < cnt / 16; ++i) {
    __m512i g = _mm512_i32gather_epi32(idx_buf[i], data, 4);
    _mm512_i32scatter_epi32(out, pos_buf[i], g, 4);
  }
  ... // Handle remaining data in scalar code.
}
```

On both platforms, there is a big jump from 0 misses to 1 miss per 16 references, and then a less steep increase with additional misses. When there are no misses, the code can run at full capacity with essentially no memory traffic. At one miss, there is a stall on average once for each SIMD instruction. Additional misses have a less dramatic impact because the system can have multiple outstanding misses at any given time.

Another observation from Fig. 3 is that the SIMD version has about 2x better performance over the scalar version when data is cache resident for the Phi. When there are many cache misses and the performance is bounded by the memory bandwidth, scalar and SIMD implementations have similar performances. In other words, SIMD optimization is most effective for accessing cache-resident data.

Given the apparent performance difference between in- and out-of-cache accesses, it would therefore be advantageous to split a data stream with a low cache miss rate into two pieces: The first (and largest) fragment always hits the cache and runs at cache speeds. The smaller fragment contains the likely misses, but when misses occur they occur together and their latencies can be overlapped.

The implementation for such a splitting method can be done in a data-parallel way. Based on an estimated threshold, a SIMD comparison operator determines which items are likely hits, and their references are resolved immediately. We use masked gather instructions in AVX-512 to directly write materialized data into the output array. For items failing the comparison, the item and its address are written to the tail of a buffer. A second pass then iterates through the buffer to resolve the remaining references. In Section 5.1, we demonstrate that for a materialization operation, the splitting method performs better than the baseline when there are one or two cache misses per 16 accesses.

4.2.2 Aggregation

The frequency information implicit in transformed identifiers can also be used to improve the performance of aggregation. As we discussed in Section 4.1, SIMD implementations of aggregation perform poorly when the degree of skew is high in data. The reason for this performance degradation is that the most popular data items become so common that conflicts within the SIMD scatter step are frequent. In other words, two or more different SIMD lanes try to update the aggregate value (e.g., count) for the same group. Conflicts take several steps to resolve, and during this resolution process additional conflicting values can be read into the SIMD register, exacerbating the problem.

To mitigate this behavior, one can avoid conflicts on common items by re-mapping accesses to the most frequent items to distinct copies, one per SIMD lane. Identifying whether an item is among the most common is simple: it has an identifier that is no more than some threshold t. Because of data skew, remapping just a few of the most frequent items can already be quite effective in reducing conflicts. For example, in our microbenchmark experiments (Section 5.1), remapping the 40 most common 4-byte values was a good choice. In the modified aggregation algorithm, we keep 16 copies of each of the top 40 values. Accesses to these common values never conflict because the copy used is determined by the SIMD lane. The remaining items have only one copy, as before. The copies can be stored immediately before the original data array, preserving data contiguity. For SIMD lane i (where $i = 0 \ldots 15$), if the identifier loaded for this lane is not greater than t, then the copy at offset $(-i * t)$ is updated.

Using SIMD instructions, the remapping can be simply implemented as masked arithmetic computations based on the comparison with the threshold t. The top-t locations in the original array now store only the aggregates from a single SIMD lane. When $t = 40$, the cache footprint is bigger by $40 * 15 * 4 = 2400$ bytes, which is small relative to typical cache sizes and thus of minor impact. In the end, combining the copies for all SIMD lanes produces the final aggregates. In our experimental evaluations, we find this copying method effectively reduces conflicts in case of high skew, drastically improving performance.

4.2.3 Heavy Hitter Aggregation

The permutation index has direct benefits for queries like Q3a that compute counts for just the most frequent elements (i.e., heavy hitters). The most common 4,000 values for Q3a are simply the transformed identifiers from 1 to 4,000, assuming that the permutation index is up-to-date. We can directly aggregate these values and ignore the rest (using the limit value as a cutoff threshold), unlike conventional methods that would need to compute the exact counts for all bid values. The cache footprint of this approach is much smaller. For 4-byte integers, the output array is entirely L1-cache resident. Since there are almost no out-of-cache memory accesses, the performance of heavy hitter aggregation is significantly better than a full aggregation.

The implementation of this technique inside a database system does not require code changes to the relational operators. Instead, if the database knows from its catalog that the query aggregates over a transformed fact table column, then its query optimizer can simply rewrite the query to add a WHERE clause with a range filter using the LIMIT value as the predicate. For example, the Q3a query can be rewritten to Q3b below if bids have been transformed using a permutation index.

```
void aggregate_copy(const size_t input_size,
                    const uint32_t* in, uint32_t* out) {
  // Add replicated bins in front of regular bins.
  const uint32_t frequent_bins_size{15 * THRESHOLD};
  // Offsets for the replicated bins in output.
  const __m512i offsets = _mm512_set_epi32(0, -THRESHOLD,
      -2 * THRESHOLD, ... , -15 * THRESHOLD);
  __m512i idx;
  __mmask16 mask = -1;
  for (size_t i = 0; i < input_size;) {
    // Load sixteen elements including previous conflicts.
    __m512i new_idx = _mm512_loadu_si512(&in[i]);
    idx = _mm512_mask_compress_epi32(new_idx, ~mask, idx);
    // Compare with the threshold.
    __mmask16 freq = _mm512_cmp_epu32_mask(
        idx, _mm512_set1_epi32(THRESHOLD), _MM_CMPINT_LT);
    // Compute new positions using replicated bins.
    __m512i pos = _mm512_add_epi32(
        _mm512_mask_add_epi32(idx, freq, idx, offsets),
        _mm512_set1_epi32(frequent_bins_size));
    // Gather current counts.
    __m512i cnt_int = _mm512_i32gather_epi32(pos, out, 4);
    // Conflict detection.
    __m512i conflicts = _mm512_conflict_epi32(pos);
    mask = _mm512_testn_epi32_mask(conflicts, ALL_NEG_1);
    // Masked scatter the updated counts.
    __m512i cnt_out = _mm512_add_epi32(cnt_int, ALL_1);
    _mm512_mask_i32scatter_epi32(out, mask, pos, cnt_out, 4);
    // Update the input pointer.
    i += _mm_popcnt_u32(mask);
  }
  ... // Sum up the counts for frequent bins.
  ... // Handle remaining data in scalar code.
}
```

```
 --Q3b: Heavy hitters using permutation index
Select   bid, count(*)
From     Sales
Where    bid < 4000
Group by bid order by bid
```

4.3 Intra-Query Parallelization

A modern CPU contains many cores, each capable of independent work. It is therefore essential that this available parallelism is exploited by implementing multi-threaded versions of query processing algorithms. Fortunately, materialization (Q1) and selection (Q2) operations are relatively easy to parallelize. We can partition the input fact data into non-overlapping chunks so that different threads can work on different chunks independently. Shared reads (e.g., from a common dimension array) typically do not cause performance issue since they are cached across cores.

Aggregation (Q3) is trickier because independent threads may try to update the count for the same bid. Similar to the SIMD conflict detection, implementations must guarantee the atomicity of potentially conflicting updates to the shared result array, which may induce a performance overhead. Data skew potentially exacerbates the problem by increasing the probability that conflicting updates occur.

01wOn the other hand, to fully avoid conflicts and the overhead, each thread can have its own independent copy of the result array, so that threads do not concurrently update the same memory location. The obvious disadvantage is memory consumption, which is multiplied by the number of threads used. Another cost is the overhead of the final aggregation that combines partial results from all threads, which gradually becomes non-negligible with more threads used.

In case of data skew, we are able to get the best of both approaches. We use a hybrid approach to allow different

TABLE 1
Hardware Specifications

Microarchitecture	Skylake	Knights Landing
Model Number	8175M	Phi 7210
Clock Frequency	2.5 GHz	1.3 GHz
Cores x SMT	24 x 2	64 x 4
L1 Size / Core	32 KB	32 KB
L2 Size / Core	1 MB	512 KB
L3 Size	33 MB	-
Memory Bandwidth	60 GB/s	55 GB/s

TABLE 2
Summary of Algorithms in Experiments

Label	Algorithm
rand	Baseline method using randomized data layout
freq	Reordered layout using permutation indexes
freq,split	Threshold-based materialization in Section 4.2.1
freq,copy	Threshold-based aggregation in Section 4.2.2
top-4k	Threshold-based heavy hitter aggregation in Section 4.2.3
hybrid	Hybrid data structures for multithreading in Section 4.3

threads access to their own private versions of the hot data, while using atomic operations on shared representations of the less frequent data. Such an approach reduces conflicts while remaining space-efficient, similar to recent approaches to parallel aggregation [35], [36], [37]. Given the available memory space and the frequency estimator, it is possible to derive (or obtain experimentally) a threshold for distinguishing the frequent data based on the transformed identifiers. As we demonstrate in our experiments, a small threshold is often enough to speed up queries in the case of skewed data. For example, typically each thread can use a private buffer for the most frequent 8,192 data items so that the buffer fits in the L1 cache, using four-byte identifiers. A limitation of the current SIMD extensions is that they lack support for vector atomic instructions. Introduction of such instructions could significantly improve performance [38].

5 Experiments

We conducted experiments on an Intel Skylake CPU and on an Intel Xeon Phi (Knights Landing) processor. Both experimental platforms support AVX-512, but they have very different architecture and hardware characteristics, as summarized in Table 1. The Xeon Phi has a lot more (weaker) physical cores, but it does not have a L3 cache, and its 1 MB L2 cache is shared between two adjacent cores. Both machines are configured to use 2 MB hugepages to avoid the TLB thrashing problem discussed in Section 2.2. 1 GB hugepages are also available and they result in similar performance. Both hugepage options lead to much better performance than using the regular 4 KB pages.

5.1 Microbenchmark Results

We have implemented microbenchmarks to evaluate the potential performance improvements enabled by the permutation index. Our code was compiled using GCC 7.3 with `-O3` optimization and loop-unrolling enabled, and ran on 64-bit Linux operating systems. Performance counters such as cache misses and branch misses were obtained using the perf events interface. We employed thread pinning to avoid undesired thread migration, maximizing the utilization of private cache and local memory on multicore NUMA platforms. Experiments are done in memory without disk I/Os.

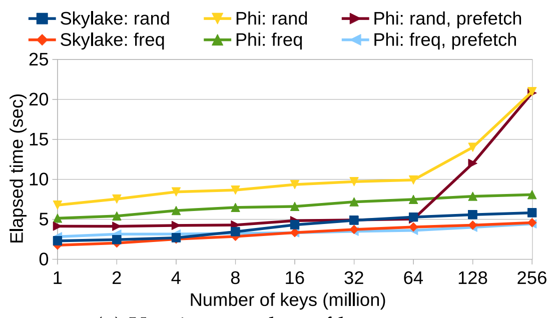
We simulate skew in empirical data by generating zipf distributions with varying zipf parameters. For a particular zipf factor, we vary the number of keys in a dimension table (i.e., the cardinality of the domain). We fix the column cardinality to 1 billion tuples in a fact table, and assign key values (4-byte integers) from the dimension domain to the fact table column under the chosen zipf distribution, in random order. We also experiment with varying sizes of the dimension domain.

For the baseline approach, the tuple identifiers in a dimension table are randomized (denoted as "rand" in the figures). For the permutation index approach, the dimension column is sorted by frequency and the referencing column in a fact table is preprocessed to hold transformed identifiers (denoted as "freq"). Using both scalar and SIMD (AVX-512) implementations, we measure the elapsed time (i.e., latency of query execution) for database operations discussed in Section 4. For a particular operation, all tuples in a column are executed in a tight loop. We also evaluate the approaches both with and without software prefetching of the data references. Table 2 summarizes the algorithm variants used in our experiments.

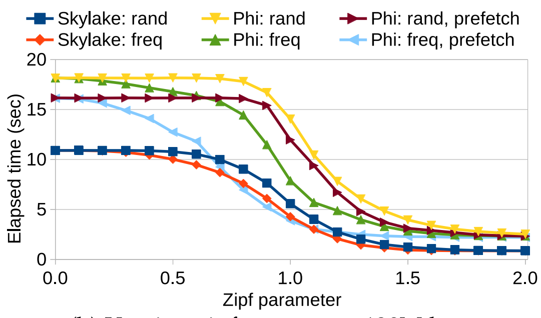
Fig. 4 shows the performance of materialization (`Q1`) using a SIMD implementation, which was faster than the scalar implementation for this query. Fig. 4a shows that at $z = 1.0$, a common distributional parameter in practice, permutation indexes speed up the query by more than 20 percent on both architectures. With the number of keys fixed at 128 million, Fig. 4b shows that performance improvements occur over a range of z values. For smaller z values, the data is close to uniform and both methods suffer cache misses. For larger z values, the skew is so concentrated in a handful of data values that both methods enjoy cache hits most of the time. Without software prefetching, the speedup at $z = 1$ and 128 million keys is 1.3x on Skylake and 1.8x on Phi.

Fig. 4c demonstrates that the time improvements on the Phi are indeed due to cache behavior. Half of the data loads are to the array of probes (almost always hits due to clustered sequential access and hardware prefetching), while half are to the accessed data in the dimension array. Thus, at small z values we see a 50 percent load hit rate in the L1 cache, and also a 50 percent miss rate for the L2 cache. The dimension table footprint is much bigger than the L2 cache, making L2 hits that are L1 misses insignificant at small z values. As z increases, the L1 hit rate of the permutation index method is better (higher) than the randomized method, while the L2 miss rate corresponding to the access frequency from RAM is also better (lower). In addition, the permutation index method has much better data TLB utilization at 128 million keys. For example, at $z = 1.0$, the DLTB miss rate is 2.6 percent using permutation indexes, while the miss rate of the baseline method is 20 percent.

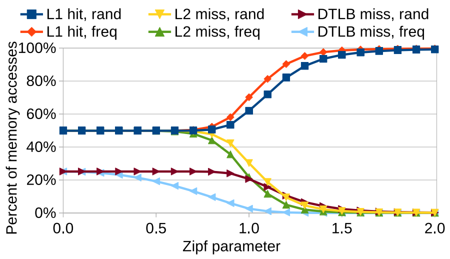
As discussed in Section 2.3, it is often possible to utilize out-of-order execution and software prefetching to hide the latency of cache misses. Figs. 4a and 4b also show that the



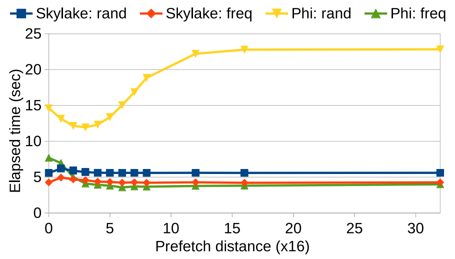
(a) Varying number of keys, $z = 1$



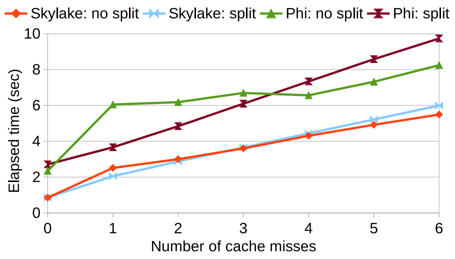
(b) Varying zipf parameter, 128M keys



(c) Cache behavior (Phi), 128M keys



(d) Varying prefetch distance, $z = 1$, 128M keys



(e) Varying cache misses per 16 elements, using threshold-based processing

Fig. 4. Materialization (Q1) performance using permutation indexes (SIMD).

benefits of permutation indexes are still present even when prefetching is used. For example, at $z = 1.0$ and 128 million keys, using permutation indexes on top of prefetching makes the query 3x faster on the Phi. For these results, we chose the prefetch distances empirically. Fig. 4d shows performance with varying prefetch distances. Since Skylake does not support AVX-512 PF extensions, we simulate the `vpgatherpf` instruction with a loop of 16 scalar prefetch instructions. As shown in the figure, the best prefetch distance on the Phi is 3(x16) for the baseline and 6(x16) for the permutation index method. As we prefetch more, the baseline approach quickly suffers due to cache thrashing, while the permutation index approach is almost unaffected. In general, we find that permutation indexes enable longer prefetch distances comparing with the baseline, making our approach even faster. On the Skylake, it turns out that the prefetch overhead is more than the benefits from prefetching, so we did not show its prefetch performance in Figs. 4a and 4b.

In the case of high skew, Fig. 4e presents a zoomed-in view of the SIMD performance (the red and green curves) from Fig. 3 for 0–6 cache misses, together with the performance of the threshold-based splitting methods described in Section 4.2.1. Without splitting, there is a big difference of 2.1–3.1x from no cache misses at all to just a single miss, while subsequent misses cause much less latency. When the number of cache misses is small (1 or 2 per 16 elements), the threshold-based implementation splitting in- and out-of-cache accesses is better than the baseline. With more misses, the benefits of this optimization are gradually outweighed by the overhead of writing out intermediate buffers.

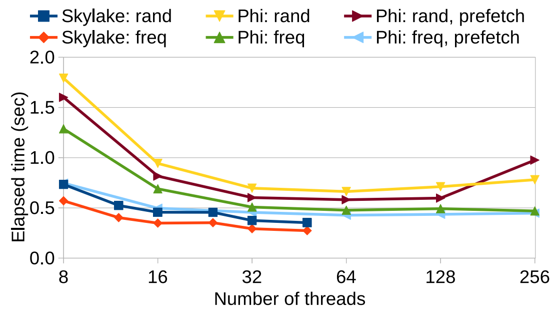
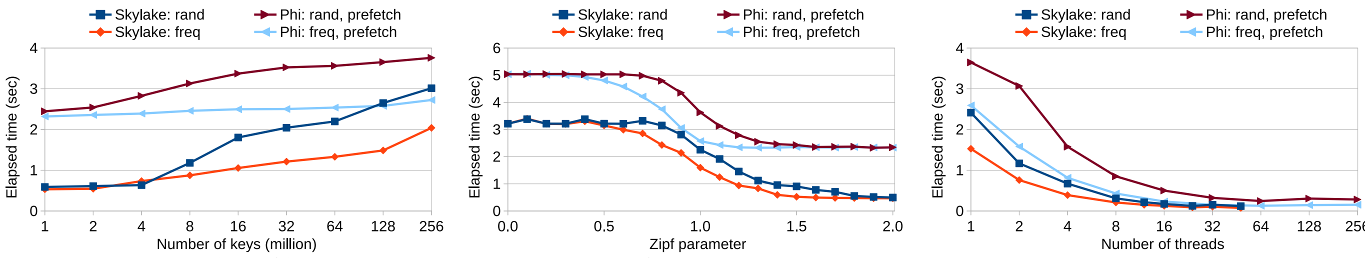


Fig. 5. Multithreaded materialization, $z = 1$, 128M keys.

Fig. 5 shows the performance of multithreaded materialization. We vary the number of threads from 1 to 256 on the Phi, and to 48 on the Skylake. Since the performance scales almost linearly up to 16 threads, we omitted the results with fewer than 8 threads used. Even though threads are sharing cache resources, the permutation index method always performs better, achieving a speedup of over 1.3x on both platforms with varying threads. On the Phi, using more than 64 threads does not provide any more benefits than using the baseline approach, and prefetching makes the query even slower. In contrast, the permutation index approach still slightly improves performance.

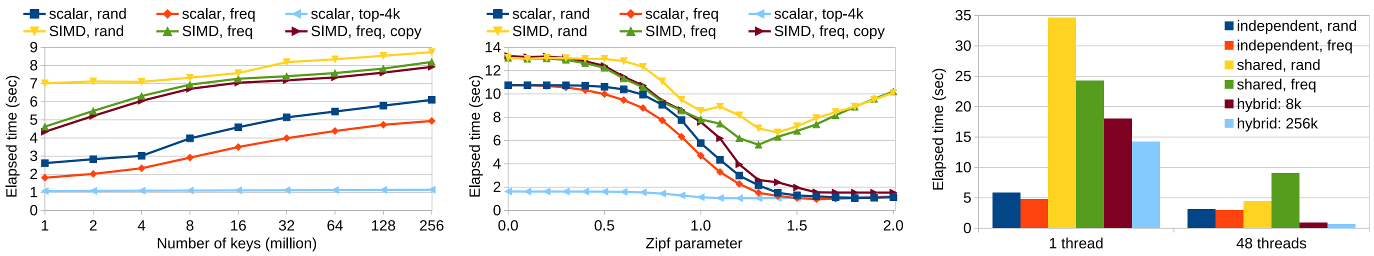
Fig. 6 shows the performance of selection (Q2) using the SIMD implementation, which was again faster than the scalar implementation for this query. We omit the results without prefetching on the Phi since they are slower. The performance profile is similar to Fig. 4, except that for low key-counts the performance is better. Because only one bit (rather than 32 bits) is needed per key, a larger number of keys is required before the cache capacity is exceeded. The relative performance of the Skylake processor is better in Fig. 6 than in Fig. 4 because it has a large L3 cache that can hold the entire bitmap. Fig. 6c shows the multithreaded performance. Comparing with the baseline, the permutation index method is 1.4–2.2x faster on the Phi, and 1.3–1.8x faster on the Skylake.

Fig. 7 shows the performance of aggregation (Q3) using the scalar and SIMD implementations on Skylake. Scalar code generally outperforms the SIMD code because the SIMD code has the overhead of conflict detection and resolution. In fact, for high z values, the impact of conflict



(a) Varying number of keys, $z = 1$ (b) Varying zipf parameter, 128M keys (c) Multithreaded, $z = 1$, 128M keys

Fig. 6. Selection (Q2) performance using permutation indexes (SIMD).



(a) Varying number of keys, $z = 1$ (b) Varying zipf parameter, 128M keys (c) Multithreaded, $z = 1$, 128M keys

Fig. 7. Aggregation (Q3) performance using permutation indexes (Skylake).

resolution creates a severe degration in performance. The threshold-based copying method described in Section 4.2.2 addresses this issue. On Skylake, the scalar version with our permutation index optimization is fastest for all z values in Fig. 7b. The modified SIMD algorithm with copying is the best-performing method at high skew levels on the Phi (not shown in the figure), but not on the Skylake machine.

Fig. 7 also shows the performance of heavy hitter aggregation for computing the counts of the 4,000 most frequent data items ("top-4k"). Our optimization discussed in Section 4.2.2 significantly improves query performance, since the threshold-based approach only updates cache-resident data. In comparison, performing full aggregates without using the permutation index would have much larger memory footprint and have to do extra work to extract the top-4,000 results. As an example, for $z = 0.5$, the performance improvement over computing the entire aggregate is 8.6x for the Phi, and 6.6x for the Skylake processor.

Fig. 7c shows the aggregation performance using all 48 threads on the Skylake. Single-threaded results are shown as a reference. We compare the performance of independent-buffer implementations, shared buffer implementations using atomic operations, and the threshold-based hybrid approach described in Section 4.3. For independent-buffer implementations, the permutation index method is better than the baseline, but both methods do not scale well because of the increased contention and the overhead of final combination when all threads are used. On the Phi, they are the slowest methods when all threads are used, despite using much more memory. For the shared-buffer implementations, the permutation index method performs worse with multithreading because it leads to more severe contention over popular cache lines for atomic operations. The hybrid method addresses the contention problem. As shown in the figure, the hybrid methods perform the best. Using 256 K as the threshold results in improved latency comparing to 8 K, but uses more memory. Comparing the independent-buffer baseline approach, this method is 4.7x faster on the Skylake (and 10.4x faster on the Phi using all 256 threads), with much smaller memory usage.

As discussed in Section 3.1, the cost of building a permutation index includes an aggregation query (Q3) followed by sorting the frequency counts, as well as a materialization query (Q1) using the baseline (rand) data layout. On both platforms, our experiments above have shown that these two queries take at most 20 seconds using a single thread over the largest data, and thus the total time of building the index takes less than a minute.

5.2 Results on Real Datasets

We tested the permutation index method on three types of real-world datasets with different sizes and degrees of skew:

1. *Pageview*[1] is a dataset of web request logs of wikipedia pages. The data is cleaned to filter out requests from search engine spiders, leaving only human traffic. In a database system, the request logs are stored as a fact table referencing a dimension table describing page information. We used the monthly request data from December 2018.
2. *Product*[2] is the Amazon reviews data. We believe the number of public product reviews is a proxy of the sales data, and it also exhibits skew. We consider the products as dimension data, and each review is a data item in the fact table. The data is separated into different categories, and each category exhibits different skewed data distribution. Our experiments use 8 categories with the most products.
3. *Graph*[3]. We tested several large graphs from the Stanford Large Network Dataset Collection [39],

1. https://dumps.wikimedia.org/other/analytics
2. https://s3.amazonaws.com/amazon-reviews-pds
3. https://snap.stanford.edu/data

TABLE 3
Table Cardinality in Real-World Datasets

Dataset		Dimension	Fact
Pageview	wikipedia	11,880,596	3,351,629,753
Product	book	2,717,050	19,531,329
	dvd	221,086	5,069,140
	ebook	1,292,480	17,622,415
	home	918,287	6,221,559
	music	675,893	4,751,577
	pc	382,331	6,908,554
	sports	753,280	4,850,360
	wireless	767,830	9,002,021
Graph	friendster	65,608,366	1,806,067,135
	orkut	3,072,441	117,185,083
	livejournal	4,847,571	68,993,773
	pokec	1,632,803	30,622,564
	youtube	1,134,890	2,987,624
	wiki-talk	2,394,385	5,021,410
	patent	3,774,768	16,518,948
	ca-road	1,965,206	5,533,214

including social, communication, citation, and road networks. In our experiments, edges are stored as a fact table, with nodes stored in a dimension table. As an example, in a social network, nodes represent users and the dimension table describes user information. An aggregation query on the fact table, for instance, is to count the number of friends for every user (i.e., node degrees).

Details of the datasets are summarized in Table 3.

We report the results of SIMD implementations for materialization and selection, and scalar implementations for aggregation (and heavy hitter aggregation). These implementations are generally fast as revealed in our microbenchmark analysis. For heavy hitter aggregation, we again compute the counts for the top 4,000 most frequent data items.

Fig. 8 shows the performance improvements on the largest dataset *Pageview*, using a varying number of threads on the Skylake. On the Skylake with a single thread, the permutation index speedup of materialization, selection, full aggregation, and heavy hitter aggregation is 1.5x, 2.2x, 1.3x, and 4.0x, respectively. The results on the Phi are generally slower than on the Skylake, but using permutation indexes similarly achieves 1.10–2.27x performance using a single thread. With more threads, permutation indexes speed up queries similarly for materialization, selection, and full aggregation, up to 2x. When all threads are used, the performance ratio for heavy hitter aggregation is 20.4x on the Skylake and 22.3x on the Phi. Because all threads can use their cache-resident private buffer to execute heavy hitter aggregation in parallel, the permutation index method is particularly effective.

Table 4 presents the single-threaded latency results (in milliseconds) and performance speedups of using permutation indexes on all other datasets, on the Skylake. On these product review and graph data, the performance improvements are up to 6.0x for heavy hitter aggregations, and up to 1.4–1.5x for other operations. Results on the Phi are generally slower and omitted due to space constraints, but they achieve similar speedups using permutation indexes.

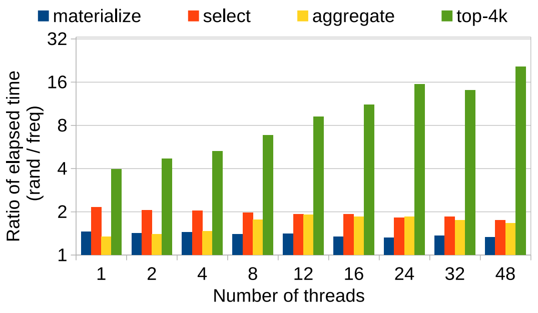


Fig. 8. Performance speedup on *Pageview* (Skylake).

5.3 Results in Database Systems

Full support of permutation indexes inside a database system requires changes to multiple components including the query optimizer and execution engine. However, one can simulate the data clustering effect of permutation indexes by using re-encoded tables with permutated dimension columns and transformed fact table columns that would have been created by our approach. By comparing the query processing over the orginal tables and the transformed tables, we can evaluate the benefits of using permutation indexes inside actual database systems.

We evaluted our approach in two column-oriented main-memory databases: MonetDB [10] (version 11.35.19) and Quickstep [15] (built from source[4]). Both database systems were configured to run entirely in in-memory mode, and tables were pre-loaded into RAM before query execution. Unless otherwise specified, we used the default parameters of the systems including storage formats and optimizer options.

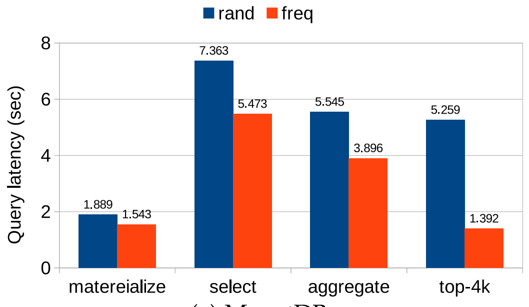
For this set of experiements, we create tables following the book sales schema used by the example queries in Section 1. The microbenchmark data from Section 5.1 with zipf parameter $z = 1.0$ is ingested, so that there are 16 million distinct keys in the dimension table `Books`, and 128 million tuples in the fact table `Sales`. We compare the latencies of query execution over the transformed tables using permutation indexes (freq), against the baseline method (rand) where the dimension table is filled with keys in a randomized order. This comparison demonstrates how much performance improvement can be achieved by simply changing the data layout to improve cache utilization, without modifying the database operator implementation as described in Section 4 (except for Section 4.2.3, where the heavy hitter query `Q3a` can be rewritten to `Q3b` easily). Note that neither system uses SIMD instructions to implement database operators.

Fig. 9a presents the single-thread query performance in MonetDB on the Skylake platform. The speedups of using permutation indexes for materialization (`Q1`), selection (`Q2`), aggregation (`Q3`) and heavy hitters (`Q3a`) are 1.22x, 1.34x, 1.42x, and 3.78x, respectively. Results on the Phi have slightly better speedups, but they are slower in absolute time and omitted. We further analyze the query plans and implementation to understand where the improvement comes from. For `Q1`, MonetDB exploits the fact that the dimension table is dense, so it chooses a very efficient fetch-join to directly executes positional lookups into the dimension

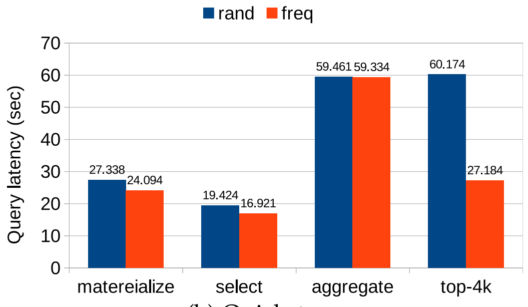
4. https://github.com/UWQuickstep/quickstep

TABLE 4
Performance Results on Real Data (Skylake)

Dataset \ Query	Materialize			Select			Aggregate			Top-4k	
	rand	freq	speedup	rand	freq	speedup	rand	freq	ratio	freq	speedup
wikipedia	14608.2	10051.7	1.45	5691.21	2644.98	2.15	15713.0	11707.7	1.34	3972.64	3.96
book	74.89	53.54	1.40	11.88	10.07	1.18	71.40	62.23	1.15	25.46	2.80
dvd	5.57	4.61	1.21	1.77	1.75	1.01	5.47	4.84	1.13	5.34	1.02
ebook	49.04	35.53	1.38	10.10	8.20	1.23	53.09	39.48	1.34	20.76	2.56
home	14.40	10.46	1.38	3.10	2.53	1.23	14.38	10.60	1.36	6.99	2.06
music	10.23	7.49	1.37	2.18	1.88	1.16	10.24	7.70	1.33	5.61	1.82
pc	10.96	7.60	1.44	2.95	2.80	1.05	10.19	6.82	1.49	6.99	1.46
sports	10.13	7.64	1.33	2.41	1.95	1.23	10.04	6.87	1.46	5.47	1.84
wireless	19.96	13.46	1.48	4.62	3.67	1.26	21.61	13.38	1.62	9.78	2.21
friendster	16514.6	12136.3	1.36	5243.83	3753.34	1.40	16423.1	12692.4	1.29	2718.59	6.04
orkut	400.85	386.93	1.04	73.30	69.67	1.05	459.44	440.40	1.04	169.06	2.72
livejournal	248.14	230.95	1.07	47.31	44.02	1.07	276.49	267.41	1.03	99.58	2.78
pokec	100.37	88.78	1.13	18.34	16.66	1.10	112.33	101.78	1.10	44.56	2.52
youtube	6.23	4.32	1.44	1.45	1.12	1.30	6.00	4.53	1.32	3.46	1.73
wiki-talk	11.25	9.46	1.19	2.54	1.95	1.30	10.07	9.38	1.07	5.00	2.02
patent	59.39	55.11	1.08	10.10	9.92	1.02	62.32	60.71	1.03	24.44	2.55
ca-road	16.94	16.57	1.02	2.95	2.9	1.02	16.46	16.19	1.02	8.37	1.97



(a) MonetDB



(b) Quickstep

Fig. 9. Query performance in main-memory database systems.

column. This is similar to the query processing algorithm discussed in Section 1, so the permutation index method improves cache utilization and query performance. For `Q2`, different from what we have discussed with selections, MonetDB chooses to execute a hash join over the filtered tables. In this case, the query plan reveals that a big hash table without any collisions is built over the dimension table using an identity hash function of `Books.bid`. Similarly for `Q3`, MonetDB executes a hash-based aggregation, building a hash table with the same number of slots as the size of the dimension table using an identity function as the hash function. As a result, the memory access pattern of these two queries is the same as accessing the dimension table directly, so our optimization translates to a better cache locality when accessing the hash table, leading to better performances. By tracing the query execution, we also verified that the improvement was indeed from the probe phase. For `Q3a`, the rewriting to `Q3b` is very effective in reducing latency when executing over the transformed tables. In contrast, the query execution over the original tables (rand) has to aggregate over the entire domain and thus has similar performance to `Q3`.

Fig. 9b shows the query performance in Quickstep. Quickstep chooses to execute a hash join for `Q1` and `Q2`. For these two queries, Quickstep also uses an identity function as the hash function, and builds the hash table over the dimension column, so the cache locality optimization is preserved in the constructed hash table. The speedups are 1.13x and 1.15x, respectively. The aggregation query `Q3` appears to be slow in Quickstep. We find that although Quickstep also uses a hash-based aggregation algorithm, the aggregation hash table is implemented differently from join hash tables. And unlike MonetDB, the hash table is built directly over the fact table column `Sales.bid`, without exploiting the foreign key relationship to `Books`. As a result, we do not observe any performance improvements. For `Q3a`, the speedup is 2.21x after rewriting.

As we observe from these results, the permutation index method can already improve cache utilization and query performance inside database systems with the existing query processors. To further accelerate queries, the operator implementation can be optimized to exploit the permutation index as discussed in Section 4.2.

6 Related Work

Data Skew. As common in empirical databases, skew is often a challenge to efficient query processing. Databases have to carefully handle parallel query execution and avoid performance degradation due to various long tail effects [40]. When skew causes load imbalances in a parallel join or sort, explicit scheduling of common keys and specially designed partitioning strategies can overcome this issue [41], [42],

[43], [44]. Aggregations are affected because heavy hitters can create contention in shared data structures [35], [36]. MapReduce-based execution also has to mitigate skew to reduce the job running time [45]. In addition, skew has an impact on the performance of selectivity estimators in query optimization, and skew-aware methods are required for accurate estimation [46]. Different from these studies, we demonstrate that skew can in fact improve performance when the skewed access pattern is properly exploited.

Orthogonal to our approach, data skew has been used to improve query execution in other ways. High-frequency data items can be represented with fewer bits to improve compression rate and reduce data transfer costs [47], [48]. At lower levels of the memory hierarchy, popular rows may be surrounded by cold rows in a disk page. This observation has led some systems to adopt row caches that keep popular rows in RAM even if their page has been evicted from the buffer pool [49]. Such a solution is not viable at higher levels of the memory hierarchy because programmers have little direct control of what data is loaded into the caches.

Standard benchmarks such as TPC-H and SSB have specified uniform distributions and avoided skew [50]. Even when benchmarks such as TPC-DS adopt skew, they are constrained by query comparability issues to make the data more uniform than it would otherwise be [51].

Main-Memory Column Store. The advent of large main memory capacity paved the way for high-performance OLAP query execution. Column-oriented execution [52] and cache-conscious operators [52] were proposed before the advent of multi-core CPUs. Analytical database systems adopted column-oriented storage, while focusing on compression [6], [32], [53] and complex materialization strategies [54], [55] to further optimize memory access. Block-at-a-time execution [56] and code generation [57], [58] are both state-of-the-art designs for analytical query engines [59].

SIMD implementations of stand-alone operators such as sorting [24], [33], [60], [61] and join [19], [31], [62], [63], [64] are common. Linear-access operators such as scans [21] and compression [32], [65], are inherently data-parallel. Advanced SIMD optimizations [34] include non-linear-access operators that use vector gathers, such as joins executed by dereferencing join indexes, which are likely to benefit from the permutation indexes when there is skew in the join attribute distribution.

Other architecture-specific optimizations such as software prefetching and query compilation are used to improve database performance [25]. Cache partitioning is used to address the cache pollution problem in shared cache to accelerate concurrent workloads [66]. The performance of frequent pattern mining algorithms can also be improved by tuning data layout and access patterns [67].

7 Conclusion

We propose permutation indexes to reorder data so that popular data items are concentrated in the cache hierarchy. We find this approach improves cache utilization at all levels, and works best with realistic degree of data skew inherent in many practical domains. Efficient database operators using SIMD vectorization can be combined with permutation indexes to speed up query execution. Extensive experiments with real and synthetic data demonstrate that the performance of materialization, selection, and aggregation queries can be significantly improved.

Acknowledgments

This work was supported in part by a gift from Oracle Corporation.

References

[1] F. Faerber *et al.*, “Main memory database systems,” *Found. Trends® Databases*, vol. 8, no. 1/2, pp. 1–130, 2017.
[2] R. Kallman *et al.*, “H-store: A high-performance, distributed main memory transaction processing system,” *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.
[3] C. Diaconu *et al.*, “Hekaton: SQL server’s memory-optimized OLTP engine,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1243–1254.
[4] A. Kemper and T. Neumann, “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots,” in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 195–206.
[5] R. Barber *et al.*, “Business analytics in (a) blink,” *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 9–14, Mar. 2012.
[6] V. Raman *et al.*, “DB2 with BLU acceleration: So much more than just a column store,” *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1080–1091, 2013.
[7] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, “SAP HANA database: Data management for modern business applications,” *ACM SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, 2012.
[8] M. Zukowski, M. Van de Wiel, and P. Boncz, “Vectorwise: A vectorized analytical DBMS,” in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 1349–1350.
[9] T. Lahiri, M.-A. Neimat, and S. Folkman, “Oracle TimesTen: An in-memory database for enterprise applications,” *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 6–13, Jun. 2013.
[10] P. A. Boncz, M. L. Kersten, and S. Manegold, “Breaking the memory wall in MonetDB,” *Commun. ACM*, vol. 51, no. 12, pp. 77–85, 2008.
[11] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner, “Hyrise re-engineered: An extensible database system for research in relational in-memory data management,” *EDBT*, pp. 313–324, 2019.
[12] K. Krikellas, S. D. Viglas, and M. Cintra, “Generating code for holistic query evaluation,” in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 613–624.
[13] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi, “Building efficient query engines in a high-level language,” *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 853–864, 2014.
[14] A. Pavlo *et al.*, “Self-driving database management systems,” in *Proc. Biennial Conf. Innovative Data Syst. Res.*, 2017, pp. 1–6.
[15] J. M. Patel *et al.*, “Quickstep: A data platform based on the scaling-up approach,” *Proc. VLDB Endowment*, vol. 11, no. 6, pp. 663–676, 2018.
[16] D. Abadi *et al.*, “The design and implementation of modern column-oriented database systems,” *Found. Trends® Databases*, vol. 5, no. 3, pp. 197–280, 2013.
[17] A. Clauset, C. R. Shalizi, and M. E. Newman, “Power-law distributions in empirical data,” *SIAM Rev.*, vol. 51, no. 4, pp. 661–703, 2009.
[18] M. E. Newman, “Power laws, Pareto distributions and Zipf’s law,” *Contemp. Phys.*, vol. 46, no. 5, pp. 323–351, 2005.
[19] S. Schuh, X. Chen, and J. Dittrich, “An experimental comparison of thirteen relational Equi-joins in main memory,” in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1961–1976.
[20] I. Moreno-Sánchez, F. Font-Clos, and Á. Corral, “Large-scale analysis of Zipf’s law in english texts,” *PloS One*, vol. 11, no. 1, 2016, Art. no. e0147073.
[21] J. Zhou and K. A. Ross, “Implementing database operations using SIMD instructions,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 145–156.
[22] J. Doweck *et al.*“Inside 6th-generation Intel Core: New microarchitecture code-named Skylake,” *IEEE Micro*, vol. 37, no. 2, pp. 52–62, Mar./Apr. 2017.
[23] A. Sodani *et al.*“Knights landing: Second-generation Intel Xeon Phi product,” *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar./Apr. 2016.
[24] O. Polychroniou and K. A. Ross, “A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 755–766.

[25] P. Menon, T. C. Mowry, and A. Pavlo, "Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last," *Proc. VLDB Endowment*, vol. 11, no. 1, pp. 1–13, 2017.

[26] J. I. Munro, R. Raman, V. Raman, and S. Rao, "Succinct representations of permutations and functions," *Theor. Comput. Sci.*, vol. 438, pp. 74–88, 2012.

[27] J. Barbay, "Succinct and compressed data structures for permutations and integer functions," in *Encyclopedia of Algorithms*. Berlin, Germany: Springer, 2008, pp. 1–7.

[28] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proc. 17th Annu. Int. Conf. Supercomput.*, 2003, pp. 150–159.

[29] N. Beckmann and D. Sanchez, "Modeling cache performance beyond LRU," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 225–236.

[30] J. Alstott, E. Bullmore, and D. Plenz, "Powerlaw: A Python package for analysis of heavy-tailed distributions," *PloS One*, vol. 9, no. 1, 2014, Art. no. e85777.

[31] C. Kim *et al.*, "Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, 2009.

[32] 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," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 385–394, 2009.

[33] N. Satish *et al.*, "Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 351–362.

[34] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1493–1508.

[35] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 339–350.

[36] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye, "Automatic contention detection and amelioration for data-intensive operations," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 483–494.

[37] Y. Ye, K. A. Ross, and N. Vesdapunt, "Scalable aggregation on multicore processors," in *Proc. 7th Int. Workshop Data Manage. New Hardware*, 2011, pp. 1–9.

[38] S. Kumar *et al.*, "Atomic vector operations on chip multiprocessors," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 441–452, 2008.

[39] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

[40] H. Märtens, "A classification of skew effects in parallel database systems," in *Proc. Eur. Conf. Parallel Process.*, 2001, pp. 291–300.

[41] J. L. Wolf, D. M. Dias, and P. S. Yu, "An effective algorithm for parallelizing hash joins in the presence of data skew," in *Proc. 7th Int. Conf. Data Eng.*, 1991, pp. 200–209.

[42] D. J. DeWitt *et al.*, "Practical skew handling in parallel joins," in *Proc. 18th Int. Conf. Very Large Data Bases*, 1992, pp. 27–40.

[43] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 61–72.

[44] J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker, "Skew-aware join optimization for array databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 123–135.

[45] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating skew in MapReduce applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 25–36.

[46] C. A. Lynch, "Selectivity estimation and query optimization in large databases with highly skewed distribution of column values," in *Proc. Int. Conf. Very Large Data Bases*, 1988, pp. 240–251.

[47] Y. Li, C. Chasseur, and J. M. Patel, "A padded encoding scheme to accelerate scans by leveraging skew," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1509–1524.

[48] B. Hentschel, M. S. Kester, and S. Idreos, "Column sketches: A scan accelerator for rapid and robust predicate evaluation," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 857–872.

[49] N. Lastovica, "Guide to database performance and tuning: Row cache enhancements." Accessed: Jan. 15, 2019, 2003. [Online]. Available: https://www.oracle.com/technetwork/database/rdb/0308-row-cache-712a-134300.pdf

[50] T. Rabl, M. Poess, H.-A. Jacobsen, P. O'Neil, and E. O'Neil, "Variations of the star schema benchmark to test the effects of data skew on query performance," in *Proc. 4th ACM/SPEC Int. Conf. Perform. Eng.*, 2013, pp. 361–372.

[51] R. O. Nambiar and M. Poess, "The making of TPC-DS," in *Proc. 32nd Int. Conf. Very Large Data Bases*, 2006, pp. 1049–1058.

[52] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing database architecture for the new bottleneck: Memory access," *VLDB J.*, vol. 9, no. 3, pp. 231–246, 2000.

[53] M. Stonebraker *et al.*, "C-Store: A column-oriented DBMS," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 553–564.

[54] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden, "Materialization strategies in a column-oriented DBMS," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, 2007, pp. 466–475.

[55] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear, "Materialization strategies in the Vertica analytic database: Lessons learned," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 1196–1207.

[56] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-pipelining query execution," in *Proc. Biennial Conf. Innovative Data Syst. Res.*, 2005, pp. 225–237.

[57] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.

[58] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 307–322.

[59] 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," *Proc. VLDB Endowment*, vol. 11, no. 13, pp. 2209–2222, 2018.

[60] J. Chhugani *et al.*, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.

[61] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "AA-sort: A new parallel sorting algorithm for multi-core SIMD processors," in *Proc. 16th Int. Conf. Parallel Archit. Compilation Techn.*, 2007, pp. 189–198.

[62] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *Proc. VLDB Endowment*, vol. 7, no. 1, pp. 85–96, 2013.

[63] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 362–373.

[64] 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," *Proc. VLDB Endowment*, vol. 8, no. 6, pp. 642–653, 2015.

[65] 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 *Proc. Int. Conf. Manage. Data*, 2016, pp. 311–326.

[66] S. Noll, J. Teubner, N. May, and A. Böhm, "Accelerating concurrent workloads with CPU cache partitioning," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 437–448.

[67] M. Wei, C. Jiang, and M. Snir, "Programming patterns for architecture-level software optimizations on frequent pattern mining," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, 2007, pp. 336–345.

Wangda Zhang received the MPhil degree from the University of Hong Kong, Hong Kong. He is working toward the PhD degree with the Computer Science Department, Columbia Univeristy, New York City, New York. His research interests include large-scale data management and database systems.

Kenneth A. Ross received the PhD degree from Stanford University, Stanford, California. He is currently a professor with the Computer Science Department, Columbia University, New York City. His research interests include various aspects of database systems, including query processing, query language design, data warehousing, and architecture-sensitive database system design. He also has an interest in computational biology, including the analysis of large genomic data sets. He has received several awards, including a Packard Foundation Fellowship, a Sloan Foundation Fellowship, and an NSF Young Investigator Award.