2018 IEEE 34th International Conference on Data Engineering

Hebe: An Order-oblivious and High-performance Execution Scheme for Conjunctive Predicates

Zeke Wang [1*], Kai Zhang[2], Haihang Zhou[3], Xue Liu[4], Bingsheng He[3]

[1] *Systems Group, Department of Computer Science, ETH Zürich, Switzerland*

[2] *Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China*

[3] *School of Computing, National University of Singapore, Singapore*

[4] *Department of Computer Science,Northeastern University, China*

***Abstract*—The optimization of conjunctive predicates is still critical to the overall performance of analytic data processing tasks, especially on a denormalized table, where queries with time-consuming joins on the original normalized tables are converted into simple scans. Existing work relies on the query optimizer to do the selectivity estimation and then produce the optimal evaluation order of predicates. In this paper, we argue for an order-oblivious approach, based on memory-efficient storage layouts. Accordingly, we propose Hebe, a simplified execution scheme which is attractive to the query optimizer, as it does not need to go through a sampling process to determine an optimal evaluation order of predicates. Compared with the state-of-the-art implementation with the optimal evaluation order, Hebe can also achieve up to 153% performance improvement.**

I. Introduction

Decision support queries often contain conjunctive predicates in data warehouses. For example, most TPC-H queries contain at least two predicates. The optimization of queries with conjunctive predicates is challenging since the exploration space is large. Take the conjunction $p(1) \bigwedge p(2)$ of two predicates $p(1)$ and $p(2)$ as an example. They can be evaluated with either *logical-and* & or *branching-and* && [7]. The conjunction will output the *result bit vector*, where one bit indicates the result of one tuple. The former evaluates the two predicates independently to generate one-bit result for each predicate. It then performs the logical *and* operation on the two one-bit results. Suppose *branching-and* will evaluate the predicate $p(1)$ first. If its outcome is false, the final result (false) is determined, and there is no need to evaluate $p(2)$. If it is true, $p(2)$ is evaluated, and then its outcome determines the final result. Therefore, *logical-and* is oblivious to the evaluation order but it has to evaluate all the involving predicates. In other words, it does not explore any *cut-off* condition (i.e., short-circuit) among predicates (the cut-off condition among predicates is called *inter-predicate* cut-off condition). In contrast, *branching-and* explores the inter-predicate cut-off condition and thus is sensitive to the evaluation order.

Since database predicates can be very selective, there is plenty of related work [4], [6], [7], [8], [13] on how do the short-circuit evaluation (i.e., *branching-and* &&) to achieve better performance. The main idea is to try to figure out the optimal evaluation order of predicates using different metrics, e.g., selectivity and rank. Since the selectivity of each predicate is unknown for ad-hoc queries, the query optimizer (QO) needs to calculate them via some estimation approaches such as sampling [1], [4]. Based on the estimated selectivities, the QO produces the query execution plan (QEP) with an optimal evaluation order. Since the selectivity estimation itself can be inaccurate, the quality of QEP cannot be guaranteed to be optimal after sampling.

In this paper, we argue for an alternative approach for the evaluation of conjunctive predicates. Specifically, our approach explores the inter-predicate cut-off conditions while keeping the predicate evaluation order-oblivious. To do this, we rely on memory-efficient storage layouts [3], [5].

Memory-efficient storage layouts [3], [5] vertically partition the *codes* of one column, resulting in several *memory regions* to store the column. The codes are generated from the column values using dictionary compression [2], [5]. The memory region denotes a data structure that stores data in a sequence. More specifically, BitWeaving/V [5] proposed a set of storage layouts to fully exploit the intra-cycle parallelism available in modern CPUs. More recently, a byte-level columnar storage layout (called ByteSlice) was proposed to accelerate database scans by fully leveraging SIMD computing [3], [12]. Accordingly, an *early stopping* technique has been proposed to fully exploit the *intra-predicate* cut-off condition, based on these memory-efficient storage layouts. With the early stopping technique, the final result of a code evaluating a predicate can be determined after evaluating its partial bits (not all the bits). To illustrate, consider two 7-bit codes ($v_1 = 0\underline{0}00101$, $v_2 = 01\underline{0}0100$) try to evaluate the predicate $\widehat{p} : v < 0\underline{11}0110$, where $\widehat{p}$ indicates that the predicate p is evaluated under memory-efficient storage layouts. We can observe that v_1 (or v_2) can terminate its evaluation after evaluating the first two (or three) bits, with the last evaluated bit underlined. Therefore, there is no need to evaluate the remaining bits.

Leveraging insight from memory-efficient storage layouts and early stopping technique, we propose a simplified execution scheme Hebe for conjunctive predicates. Hebe can aggressively explore intra-predicate and inter-predicate cut-off conditions to significantly reduce branch misprediction and

[*] This work was done while was at NUS.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00125

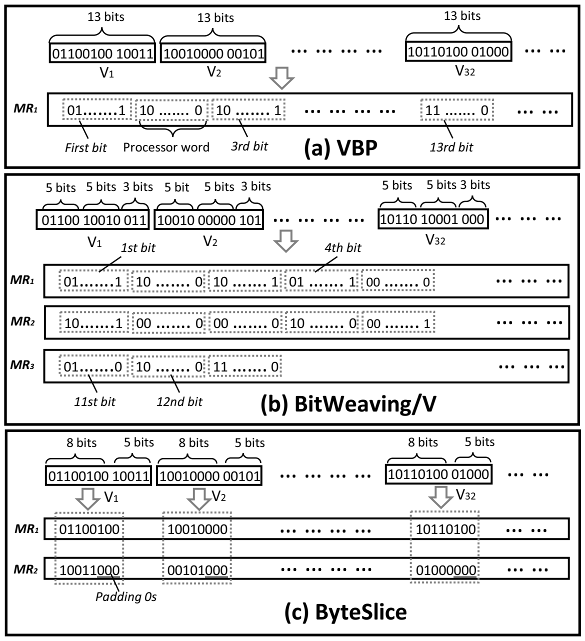


Fig. 1: Three memory-efficient storage layouts

cache miss penalties. Therefore, its raw performance is always better than the state-of-the-art execution model (i.e., column-first) under memory-efficient storage layouts [3]. Specifically, our experimental result shows that Hebe is capable of achieving up to 153% performance improvement over the column-first execution model. Meanwhile, its order-oblivious property is attractive to the QO which does not need to estimate selectivities and then to determine the optimal evaluation order of conjunctive predicates.

II. Background and Related Work

In this section, we briefly introduce the efficient storage layouts and the processing of conjunctive predicates.

A. Memory-Efficient Storage Layouts

In this subsection, we mainly discuss the characteristics of three memory-efficient storage layouts: VBP [5], BitWeaving/V [5] and ByteSlice [3]. All of them can benefit from the early stopping technique.

Vertical Bit Parallel (VBP) Layout. VBP vertically partitions the codes at the bit level [5]. In particular, the codes are divided into *segments*, and each segment contains W codes, where W is the width of a processor word. In a segment, the W k-bit codes are transposed into k W-bit *processor words*, with the most significant bits stored at the lowest address. The j-th bit in the i-th word equals to the i-th bit in the original j-th code. For example, Figure 1a shows the transposition of one segment, from 32 13-bit codes to 13 32-bit words, which are stored in a continuous memory space. The consecutive segments are also stored continuously. Therefore, it only needs one memory region MR_1 to store the transposed codes. Within VBP layout, the performance of scan can be improved by the early stopping technique. However, it is far from optimal. Suppose one cache line contains eight words. The outcome of comparisons on 32 codes of this segment in Figure 1a is determined after evaluating the first five words. Then, the other three words, which have already been loaded in the same cache, are skipped for the evaluation due to early stopping technique. Therefore, resulting in wasted memory bandwidth.

Bitweaving/V. Since the VBP layout does not make full utilization of early stopping technique, the Bitweaving/V layout [5] (built on VBP) is proposed to combine vertical and horizontal partitioning. It partitions the code not only at the bit level, but also in a horizontal fashion to achieve better CPU cache performance. In particular, BitWeaving/V divides the k words in a segment into $\lceil k/G \rceil$ *bit groups*, where G is the number of bits in a bit group. Each bit group is associated with one memory region to store the sequential words. Figure 1b shows the case with three bit groups and $G = 5$, so there are three memory regions for the compressed codes. With BitWeaving/V, the previous example only needs to access the first bit group containing the first five bits in the memory region MR_1. Thus, it can save plenty of memory bandwidth compared with the VBP layout. VBP is a special case of BitWeaving/V (only one bit group).

ByteSlice. Since the SIMD register of modern CPUs supports the 8-bit bank width, the byte-level columnar layout ByteSlice [3] can fully leverage the data-level parallelism of SIMD instructions by vertically distributing bytes of a k-bit code across $\lceil k/8 \rceil$ memory regions. It is like BitWeaving/V since both layouts contain vertical and horizontal partitioning. However, byte, instead of bits, serves as the basic unit when vertically partitioning the codes under ByteSlice. Figure 1c shows the transposition of one segment under ByteSlice layout: each 13-bit code is partitioned into two memory regions.

B. Processing of Conjunctive Predicates

The database query can consist of several predicates. The optimization of predicates is still an attractive research topic. There is plenty of related work [4], [6], [7], [8], [9], [13] to optimize the evaluation order of predicates using different metrics, e.g., ranking and selectivity. However, they do not have the order-oblivious property as the proposed design.

III. Design and Implementation of Hebe

In this section, we present the implementation details of Hebe, a simplified execution scheme which is order-oblivious and high-performance on modern CPU architectures.

The detailed execution flow of Hebe is shown in Algorithm 1. We use ByteSlice as the default storage layout. N conjunctive predicates are taken as input and a result bit vector $bitvector$ is generated to indicate whether each tuple satisfies conjunctive predicates or not.

In the initialization step, the bytes of literal $c(i)$ of $p(i)$ are broadcast to $\lceil B(i)/8 \rceil$ SIMD registers $D_c(i)$ (Line 3). $B(i)$ is the number of memory regions where $p(i)$ is evaluated. $D_c(i)$ is shared by each segment (Lines 1-5), and thus only need to be computed once. For each segment, the detailed execution flow (Lines 6-31) is given step by step.

First, we initialize three W-bit segment-level status masks (Lines 7-11) for each predicate: less-than mask $\mathcal{M}_{lt}$ (0^W),

Algorithm 1: PROPOSED EXECUTION SCHEME

```
   Input   : $N$: the number of predicates,
             $c(i)$: the literal of $p(i)$,
             $v_l(i)$: the $l$-th code of $p(i)$,
             $B(i)$: the number of bytes of code which evaluates $p(i)$.
   Output  : $bitvector$: result bit vector of conjunctive predicates.
1  for $i = 1$ to $N$ do
2      for $j = 1$ to $B(i)$ do
3          $D_c^{[j]}(i)$ = v_broadcast($c^{[j]}(i)$)
4      end
5  end
   /* Evaluate the codes in each segment in parallel.          */
6  for (each segment with codes $v_{l+1} \dots v_{l+W/8}$) do
7      for $i = 1$ to $N$ do
8          $\mathcal{M}_{lt}(i) = 0^W$
9          $\mathcal{M}_{gt}(i) = 0^W$
10         $\mathcal{M}_{eq}(i) = 1^W$
11     end
12     for $j = 1$ to $\max_{1 \le i \le N} B(i)$ do
13         for $i = 1$ to $N$ do
               /* Check cut-off condition for $p(i)$.              */
14             if $(\mathcal{M}_{eq}(i) \neq 0^W)$&&$(j \le B(i))$ then
15                 $D^{[j]}(i) = v\_load(v_{l+1}^{[j]}(i) \dots v_{l+W/8}^{[j]}(i))$
16                 $M_{lt}(i) = v\_cmp\_lt(D^{[j]}(i), D_c^{[j]}(i))$
17                 $M_{gt}(i) = v\_cmp\_gt(D^{[j]}(i), D_c^{[j]}(i))$
18                 $M_{eq}(i) = v\_cmp\_eq(D^{[j]}(i), D_c^{[j]}(i))$
19                 $\mathcal{M}_{lt}(i) = v\_or(\mathcal{M}_{lt}(i), v\_and(\mathcal{M}_{eq}(i), M_{lt}(i)))$
20                 $\mathcal{M}_{gt}(i) = v\_or(\mathcal{M}_{gt}(i), v\_and(\mathcal{M}_{eq}(i), M_{gt}(i)))$
21                 $\mathcal{M}_{eq}(i) = v\_and(\mathcal{M}_{eq}(i), M_{eq}(i))$
22             end
23         end
           /* $\mathbb{M}$ further prunes each predicate.                */
24         $\mathbb{M}$ = global_filter($\mathcal{M}_{eq}(1:N)$, $\mathcal{M}_{gt}(1:N)$, $\mathcal{M}_{lt}(1:N)$)
25         for $i = 1$ to $N$ do
26             $\mathcal{M}_{eq}(i) = v\_and(\mathcal{M}_{eq}(i), \mathbb{M})$
27         end
28     end
29     $\mathcal{M}_{final}$ = final_mask($\mathcal{M}_{eq}(1:N)$, $\mathcal{M}_{gt}(1:N)$, $\mathcal{M}_{lt}(1:N)$)
30     $r$ = v_movemask($\mathcal{M}_{final}$)
31     Append $r$ to $bitvector$
32 end
```

greater-than mask $\mathcal{M}_{gt}$ (0^W) and equal-to mask $\mathcal{M}_{eq}$ (1^W)[1], indicating the uncertain status of each predicate. $\mathcal{M}_{lt}$ ($\mathcal{M}_{gt}$ or $\mathcal{M}_{eq}$) contains $W/8$ 8-bit banks, where all the eight bits in the l-th bank are 1^8 if v_l is less than (greater than or equal to) c, or 0^8 otherwise.

Second, the codes are examined one byte (i.e., one memory region) per iteration until the cut-off condition is reached, with $\max_{1 \le i \le N} B(i)$ iterations (Lines 12-28). Before each iteration, the cut-off condition is checked for each predicate to early stop. The j-th byte needs to evaluate $p(i)$ (Lines 16-22) when its cut-off condition ($\mathcal{M}_{eq}(i) \neq 0^W$) is not satisfied and when each code of $p(i)$ contains at least j bytes (Line 14). The j-th byte in this segment is loaded into one SIMD register (Line 15) to compare with the corresponding j-th byte of literal $c(i)$ (Lines 16-18), with the comparison statuses stored into three local masks (M_{lt}, M_{gt} and M_{eq}). Then, these local masks are used to update three segment-level status masks (Lines 19-21). After all the N predicates are evaluated, their segment-level status masks are sent to the *global_filter* module (Line 24) which explores the inter-predicate cut-off conditions for N predicates. In particular, the filter mask of each predicate is evaluated to be $\neg\mathcal{M}_{gt}$ for the comparison type $<$ *or* $\le$, $\neg\mathcal{M}_{lt}$ for $>$ *or* $\ge$, $\neg\mathcal{M}_{lt}|\mathcal{M}_{gt}$ for $=$, and 1^W for $\neq$. Then, $\mathbb{M}$ is evaluated to be ANDed each predicate's filter mask together. Intuitively, $\mathbb{M}$ indicates whether the result of the evaluated tuple has already reached the false state (i.e., 0) or not (i.e., 1) after evaluating j bytes. If the false state is detected, no further evaluation on the tuple is required. Therefore, $\mathbb{M}$ can be used to further prune the uncertain conditions (Lines 25-27). Thus, high-cost branch mispredictions and cache misses can be significantly reduced at the expense of a few low-cost arithmetic instructions. This is the underlying reason why Hebe can achieve better performance.

Third, after the above iterations, the final result of each tuple in this segment is determined. Then $\mathcal{M}_{lt}$, $\mathcal{M}_{gt}$ and $\mathcal{M}_{eq}$[2] of this segment are sent to the *final_mask* module, which generates the final result mask $\mathcal{M}_{final}$ (Line 29). In particular, the output result mask of each predicate is evaluated to be $\mathcal{M}_{lt}$ for $<$, $\mathcal{M}_{lt}|\mathcal{M}_{eq}$ for $\le$, $\mathcal{M}_{gt}$ for $>$, $\mathcal{M}_{gt}|\mathcal{M}_{eq}$ for $\ge$, $\mathcal{M}_{eq}$ for $=$, and $\mathcal{M}_{gt}|\mathcal{M}_{le}$ for $\neq$. Then, $\mathcal{M}_{final}$ is calculated to be ANDed the output result mask of each predicate together. The W-bit $\mathcal{M}_{final}$ is condensed to a $W/8$-bit mask r using the *v_movemask* instruction (Line 30). Lastly, the final result of this segement (r) is appended to $bitvector$.

IV. EXPERIMENTAL EVALUATION

In this section, we present the experimental setup and evaluate the efficiency of Hebe.

A. Experimental Setup

Hardware Configuration. We conduct our experiments on a workstation with a 3.0GHz Intel i7-5960X 8-core CPU and 64GB DDR4 memory. Each core has 32KB L1 cache and 256KB L2 cache. Besides, all cores share the 20MB L3 cache. The CPU is based on Haswell microarchitecture which supports 256-bit AVX2 instruction set. All the related programs are compiled using ICC 16.0.3 with the optimization effort -O3. In order to accurately collect the performance profiles, we use the Intel Performance Counter Monitor [10] to collect the performance counters on the program of interest.

Workloads. In our experiment, we create the table with different number of columns, where each column contains one billion k-bit codes. By default, values of codes are uniformly distributed in the range $[0, 2^k)$, where k is equal to 17. The corresponding advantage is that the selectivity of each predicate can be tuned so that we can analyze the performance characteristics with varying selectivities.

Comparison methodology. Five implementations are used for performance comparison. The first one is Hebe (denoted as "Hebe"). Two implementations come from the state-of-the-art column-first execution model [3] under ByteSlice memory layout, where "BS_best" (or "BS_worst") indicates the implementation with the optimal (or worst) evaluation order. The other two implementations comes from the SIMD-scan method [11] with the naive column store, where "Naive_best"

[1]For better readability, we will use plain $\mathcal{M}_{lt}$, $\mathcal{M}_{gt}$ $\mathcal{M}_{eq}$ (without i) whenever we refer to all the predicates (i is from 1 to N).

[2]The fact that $\mathcal{M}_{eq}$ is pruned by $\mathbb{M}$ (Lines 25-27) will not violate the correctness, since $\mathcal{M}_{eq}$ is pruned only when the result of the tuple has already been determined. The determination comes from $\mathcal{M}_{lt}$, $\mathcal{M}_{gt}$ and is oblivious to $\mathbb{M}$.

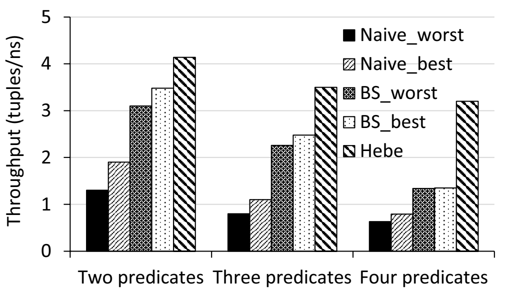
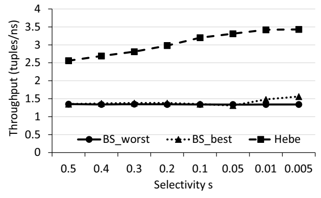
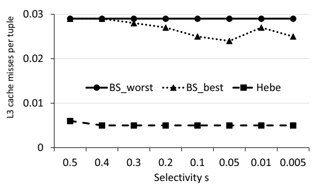


Fig. 2: Comparison with other implementations



(a) Throughput



(b) L3 cache misses

Fig. 3: Impact of selectivity

(or "Naive_worst") indicates the implementation with the optimal (or worst) evaluation order.

B. Hebe vs. Other Implementations

Suppose there are four predicates $\widehat{p(1)}$: $v1 < c1$, $\widehat{p(2)}$: $v2 < c2$, $\widehat{p(3)}$: $v3 < c3$ and $\widehat{p(4)}$: $v4 < c4$, whose selectivities are $s(1)$, $s(2)$, $s(3)$ and $s(4)$, respectively. We set the selectivity (e.g., $s(1)$) of each predicate by controlling the value of literal (e.g., $c1$). Specifically, $s(2)$, $s(3)$ and $s(4)$ are set to be 50%, and $s(1)$ is set to be 10%. "BS_best" (or "Naive_best") indicates the case with the evaluation order where $\widehat{p(1)}$ is evaluated firstly, while "BS_worst" (or "Naive_worst") indicates the case with the evaluation order where $\widehat{p(1)}$ is evaluated lastly.

Figure 2 shows the throughput of conjunctive predicates, whose number varies from 2 to 4. The predicate with low index is picked first. For example, "Two predicates" contains the predicates $\widehat{p(1)}$ and $\widehat{p(2)}$. We can make two observations.

First, as expected, two naive implementations are the slowest, since each code takes one 32-bit bank of SIMD register to evaluate. In contrast, with ByteSlice under the early stopping technique, at most 9 bits are evaluated for each code in average [3]. It means that a lot of computing resources are wasted by the naive implementations.

Second, Hebe can achieve more performance improvement over the other implementations when the number of predicates increases, since more conjunctive predicates can result in higher cut-off probability to be explored by each predicate.

C. Hebe vs. Column-first Execution Model

We compare the throughput (in terms of tuples/ns) of three cases ("BS_worst", "BS_best" and "Hebe") on conjunctive predicates $\widehat{p(1)}\&\&\widehat{p(2)}\&\&\widehat{p(3)}\&\&\widehat{p(4)}$. $s(2)$, $s(3)$ and $s(4)$ are set to be 50%, and $s(1)$ varies from 50% to 0.5% to show the effect of selectivity on the overall performance, as shown in Figure 3a. The x-axis (s) stands for the varying selectivity $s(1)$. To unveil the underlying reason of the performance improvement, we also provide one performance metric (i.e., L3 cache misses), which is collected from Intel Performance Counter Monitor [10]. We can make two observations.

First, when $\widehat{p(1)}$ becomes more selective (i.e., 50% to 0.5%), the performance of "BS_best" cannot be significantly better than that of "BS_worst", since the column-first execution model does not aggregatively explore the inter-predicate cut-off conditions to reduce the high-cost branch mispredictions and cache misses. One evidence is that the optimal order still has high L3 cache miss ratio, as shown in Figure 3b.

Second, Hebe can achieve obviously better performance when $\widehat{p(1)}$ becomes selective, since Hebe can automatically explore the inter-predicate cut-off condition. In contrast, the performance of column-first execution model is roughly stable.

V. Conclusion

The optimization of conjunctive predicates is crucial to the modern database queries. State-of-the-art works perform the sampling to guess the optimal evaluation order of predicates, which can be inaccurate and incurs high cost. Alternatively, we propose Hebe, a simplified execution scheme for the evaluation of conjunctive predicates. Hebe is order-oblivious while maintaining high-performance on modern CPU architectures.

VI. Acknowledgement

We thank Gustavo Alonso and David Sidler for their comments on the paper. Kai Zhang's research is supported by NSFC (No. 61732004, 61370080) and the Shanghai Innovation Action Project (Grant No. 16DZ1100200). Xue Liu's research is supported by the National Natural Science Foundation of China under Grant 61501103. Bingsheng He's work is supported by a MoE AcRF Tier 2 (MOE2017-T2-1-122) in Singapore.

References

[1] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. SIGMOD '05, 2005.
[2] Z. Feng and E. Lo. Accelerating aggregation using intra-cycle parallelism. In *ICDE'15*, 2015.
[3] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD '15*.
[4] F. Kastrati and G. Moerkotte. Optimization of conjunctive predicates for main memory column stores. *Proc. VLDB Endow.*, 2016.
[5] Y. Li and J. M. Patel. Bitweaving: Fast scans for main memory data processing. In *SIGMOD '13*.
[6] T. Neumann, S. Helmer, and G. Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE'05*.
[7] K. A. Ross. Conjunctive selection conditions in main memory. In *PODS '02*.
[8] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DaMoN '11*.
[9] D. Song and S. Chen. Exploiting simd for complex numerical predicates. HardBD '16.
[10] P. F. T. Willhalm, R. Dementiev. Intel Performance Counter Monitor - A better way to measure CPU utilization. Technical report, Intel, 2016.
[11] 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 Endow.*, 2009.
[12] W. Xu, Z. Feng, and E. Lo. Fast multi-column sorting in main-memory column-stores. In *SIGMOD '16*.
[13] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. In *ICDE'12*.