Understanding and Optimizing Conjunctive Predicates under Memory-efficient Storage Layouts

Zeke Wang, Xue Liu, Kai Zhang, Haihang Zhou, Bingsheng He

Abstract—Database queries can contain multiple predicates. The optimization of conjunctive predicates is still vital to the overall performance of analytic data processing tasks. Prior work proposes several memory-efficient storage layouts, e.g., BitWeaving and ByteSlice, to significantly accelerate predicate evaluation, as circuit-level intra-cycle parallelism available in modern CPUs can be exploited such that the total number of instructions can be dramatically reduced. However, the performance potential of conjunctive predicates has not been harvested yet under such storage layouts as there is no accurate cost model to provide necessary insights that guide the optimization process. In this paper, we propose a hybrid empirical/analytical cost model (**Understanding**) to unveil the performance characteristics of such storage layouts when applying to predicate evaluation. Our cost model takes into account effect of non-linear factors, e.g., cache miss and branch misprediction, and easily applies to different CPUs. The main finding from our cost model is to distinguish high-cost instruction (which suffers from cache miss and/or branch misprediction) from low-cost instruction (which enjoys cache hit and correct branch prediction) in the context of predicate evaluation under these storage layouts. Guided by such a finding, we propose a simple execution scheme Hebe (**Optimizing**), which is order-oblivious while maintaining high performance. Hebe is attractive to the query optimizer (QO), as the QO does not need to go through a sampling process to decide the optimal evaluation order in advance. The intuition behind Hebe is to significantly reduce the number of high-cost instructions while keeping low-cost instructions unchanged. Our finding from Hebe sheds light on the importance of accurate cost model that guide us to derive an efficient execution scheme for query processing on modern CPUs.

Index Terms—Database, Conjunctive Predicates, Storage Layout, CPU.

1 Introduction

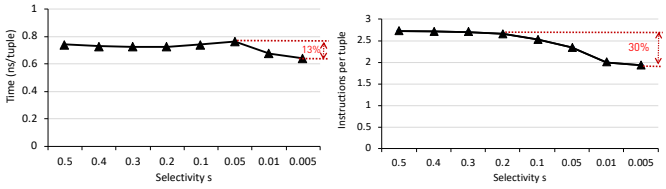
Decision support queries often contain conjunctive predicates in data warehouses. For example, most TPC-H queries contain at least two predicates. The optimization of queries with conjunctive predicates is challenging since the exploration space is large. Take the conjunction $p(1) \bigwedge p(2)$ of two predicates $p(1)$ and $p(2)$ as an example. They are evaluated with either *logical-and* $\&$ or *branching-and* $\&\&$ [33]. The conjunction outputs 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 $p(1)$ 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. In sum, *logical-and* is oblivious to the evaluation order but it has to evaluate all the involving predicates, without exploring 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 inter-predicate cut-off condition and thus is sensitive to predicate order.

In order to accelerate predicate evaluation, several memory-efficient storage layouts [11], [22] are proposed to reduce the number of required instructions by exploring *intra-predicate* cut-off condition, which exploits the cut-off condition within a predicate. Intuitively, the final result of a code evaluating a predicate can be determined after evaluating its partial bits (not all the bits). These storage layouts vertically partition the *codes* of one column, resulting in several *memory regions* to store the column, where the codes are generated from the column values using dictionary compression [10], [22]. The memory region (MR) denotes a data structure that stores data in a sequence. Under these storage layouts, the *early stopping* technique has been proposed to fully exploit the intra-predicate cut-off condition when evaluating a predicate. To illustrate, consider two 7-bit codes ($v_1 = 0\underline{0}000101$, $v_2 = 01\underline{0}0100$) 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. Through these memory-efficient storage layouts, the execution of scan with one single predicate can achieve high CPU efficiency with the help of the early stopping technique, which significantly reduces the number of evaluated instructions and the amount of memory traffic.

When evaluating conjunctive predicates $\widehat{p(1)} \bigwedge \widehat{p(2)}$ under these storage layouts [11], [22], the state-of-the-art approach is the

- *Z. Wang is with Collaborative Innovation Center of Artificial Intelligence by MOE and Zhejiang Provincial Government, Zhejiang University, China, wangzeke@zju.edu.cn*
- *X. Liu is with Department of Computer Science, Northeastern University, China*
- *K. Zhang is with Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China*
- *H. Zhou and B. He are School of Computing, National University of Singapore, Singapore.*

Manuscript received March 10, 2019; revised October 14, 2019.



(a) Elapsed time reduction (b) Instruction reduction

Fig. 1: Discrepancy between time and instruction reductions: 30% instruction reduction only leads to 13% time reduction.

column-first execution model (denoted by $\widehat{p(1)}\&\&\widehat{p(2)}$), where the tuples that do not satisfy $\widehat{p(1)}$ do not need to evaluate $\widehat{p(2)}$. In other words, the evaluation of $\widehat{p(2)}$ can take advantage of the inter-predicate cut-off condition stemming from $\widehat{p(1)}$. Meanwhile, both $\widehat{p(1)}$ and $\widehat{p(2)}$ can fully explore their own intra-predicate cut-off conditions. In sum, the column-first execution model has explored cut-off conditions in two dimensions (i.e., intra-predicate and inter-predicate) to improve the overall evaluation performance.

Despite the effectiveness of exploiting inter-predicate cut-off condition (from the column-first execution model) and exploiting intra-predicate cut-off condition (from early stopping technique and memory-efficient storage layouts), we still identify two issues.

S1: Mystifying Performance Characteristics of Conjunctive Predicates. The performance characteristics under the combining effect of inter-predicate and intra-predicate cut-off conditions are still unclear yet. Obviously, we leverage both cut-off conditions to reduce the number of evaluated instructions to make the computation faster. However, we observe that the amount of performance gain is not strongly coincident with the amount of the reduction of instructions. Figure 1 depicts the trend of "BS_best", in terms of elapsed time and instruction, when the selectivity decreases.[1] In particular, the instruction reduction by 30% only increases the performance by 13%. Therefore, the throughput is not strongly correlated with the number of instructions (as a linear factor), as non-linear factors, e.g., branch misprediction and cache miss, are critical to the overall performance.

S2: Difficult to Decide Evaluation Order of Predicates. Since database predicates can be very selective, there is plenty of related work [18], [25], [33], [34], [43] on how do the short-circuit evaluation (i.e., *branching-and* $\&\&$) to explore inter-predicate cut-off condition. Its approach is to try to guess the optimal predicate order 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 sampling [3], [18]. Based on the estimated selectivity, the QO produces the query execution plan (QEP) with an optimal evaluation order. Since the selectivity estimation itself can have errors, the quality of QEP cannot be guaranteed to be optimal after sampling. To make things more challenging, we also take the effect of memory-efficient storage layouts into account when evaluating conjunctive predicates on modern CPUs.

Therefore, the burden of optimizing conjunctive predicates under memory-efficient storage layouts still falls on the user without any rule-of-thumb guidelines. In this paper, our goal is to answer the following question:

1. "BS_best" represents the implementation with an optimal evaluation order under the ByteSlice memory layout. The exact experimental setup is shown in Subsection 6.1. Figure 1a is part of Figure 10a, while Figure 1b is part of Figure 10c.

Can we fully explore potentials of conjunctive predicates under memory-efficient storage layouts on modern CPUs?

We make the following two contributions to answer this question. First, we present a hybrid empirical/analytical cost model (**C1**) to unveil the performance characteristics (**S1**) of conjunctive predicates under memory-efficient storage layouts. Second, we propose an order-oblivious execution scheme (**C2**) to optimize conjunctive predicates to address the issue (**S2**).

C1: Hybrid Empirical/Analytical Cost Model (Understanding). We propose the hybrid empirical/analytical cost model to demystify the performance characteristics of conjunctive predicates under memory-efficient storage layouts on modern CPUs. First, we highlight two basic execution patterns which can constitute any form of conjunctive predicates. Second, we propose an empirical model to capture the performance characteristics of each basic execution pattern. The benefit of the empirical model is to be aware of CPU characteristics, e.g., branch mispredication and cache miss, while abstracting away the complexity from modeling the effect of branch mispredication and cache miss. Therefore, our model can be easily applicable to other CPUs. Third, we propose an analytical model to bridge the gap between overall performance and instantiated basic execution patterns introduced by the targeted conjunctive predicates. The benefit of the analytical model is to easily adapt to various number of conjunctive predicates. The key finding is that our cost model distinguishes high-cost instructions from low-cost instructions such that it becomes possible to harvest full performance potential when evaluating conjunctive predicates under memory-efficient storage layouts.

C2: Order-oblivious Execution Scheme (Optimizing). We argue for an alternative approach for the evaluation of conjunctive predicates. Instead of using selectivity estimation to guess the optimal evaluation order of predicates in advance, our approach explores the inter-predicate cut-off conditions while keeping the predicate evaluation order-oblivious. We propose Hebe, a simplified execution scheme for conjunctive predicates. It is order-oblivious while maintaining high-performance. Its order-oblivious property is attractive to the QO that does not need to estimate selectivities and then to determine the optimal evaluation order of predicates. Besides, its raw performance is always better than the column-first execution model with an optimal evaluation order.

We have conducted the experiments with synthesized and TPC-H workloads on two Intel CPUs. The experimental result shows that: 1) our hybrid empirical/analytical cost model captures the performance characteristics of predicate evaluation under memory-efficient storage layout, and 2) Hebe can also achieve up to 269% performance gain for the TPC-H queries over the state-of-the-art approach.

Limitations. Our work has two limitations. First, Hebe does not support user defined functions (UDFs) for predicate evaluation, as memory-efficient storage layout needs us to partition codes at a finer granularity (e.g., bit or byte) and then to directly perform operations on such a finer granularity. Typically, UDFs do not allow such finer-grained operations. Second, Hebe is not optimal in all the cases. When the selectivity of a predicate is extremely small, e.g., less than 0.001, the evaluation of conjunctive predicates can benefit, in terms of memory traffic, from evaluating this predicate first. However, we still need to identify this predicate in advance to harvest this benefit.

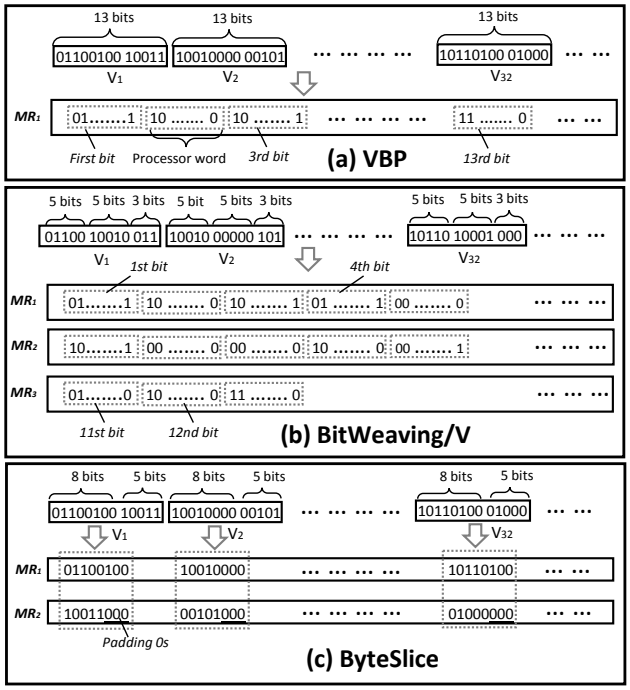


Fig. 2: A running example is the transposition of a segment that contains 32 13-bit codes to three memory-efficient storage layouts.

2 BACKGROUND

2.1 Memory-Efficient Storage Layouts

Prior work [7], [14], [20] stores column values in a compressed form after using the dictionary encoding technique which is widely used in commercial products, such as IBM Blink [31] and SAP HANA [9]. Meanwhile, several other light-weight compression methods [1], [4], [8], [42] are also used to accelerate main-memory columnar store. In this subsection, we mainly describe the characteristics of three memory-efficient storage layouts: VBP [22], BitWeaving/V [22] and ByteSlice [11]. Literally, all the above memory layouts can benefit from the *early stopping* mechanism [22], with the key idea that it is not necessary to access all the bits to determine the final result during predicate evaluation.

Vertical Bit Parallel (VBP) Layout. The VBP layout vertically partitions codes at a bit level [22]. The codes are divided into *segments*, each of which contains W codes, where W is the width of a register which accommodates codes. Inside a segment, W k-bit codes are transformed into k W-bit *register words*, where the most significant bits are stored at the lowest address. The j-th bit in the i-th word is the same as the i-th bit in the original j-th code. Figure 2a illustrates the transformation of a segment of 32 13-bit codes to 13 32-bit words. Such words 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 transposed codes. The performance of scan can be enhanced by early stopping technique under the VBP layout. However, its performance is not optimal. To illustrate, suppose a cache line consists of eight words. A *running example* is that the outcome of comparisons on 32 codes is determined after evaluating the first five words. Then, the other three words can be are skipped due to early stopping technique. However, VBP still loads such three words in CPU cache, wasting precious memory bandwidth for unnecessary data.

Bitweaving/V. Since VBP cannot full utilize early stopping technique to reduce memory traffic, Bitweaving/V [22] (built on VBP) is proposed to leverage both vertical and horizontal partitioning.

	Vector register (s_v)	Cache line (s_c)
VBP	$1-(1-(0.5)^t)^W$	$1-(1-(0.5)^t)^{CL}$
BitWeaving/V	$1-(1-(0.5)^t)^{\frac{W}{G}}$	$1-(1-(0.5)^t)^{\frac{CL}{G}}$
ByteSlice	$1-(1-(0.5)^t)^{\frac{W}{8}}$	$1-(1-(0.5)^t)^{\frac{CL}{8}}$

TABLE 1: Access probability after scanning t significant bits on modern CPUs

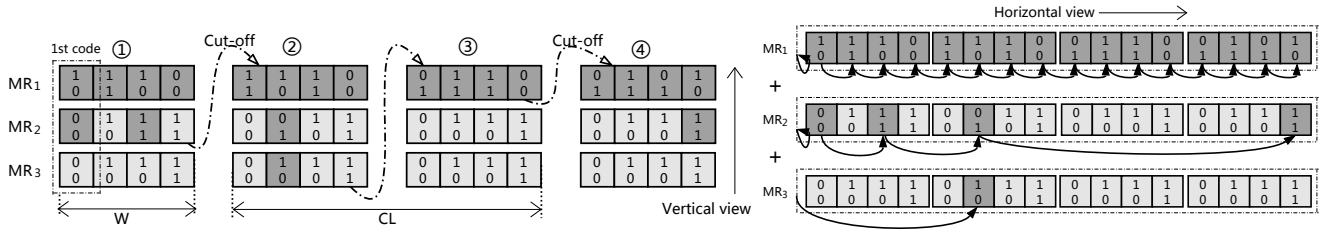
Bitweaving/V partitions a code not only at a bit level but also in a horizontal fashion such that scans need to really read less data from memory when cut-off condition is satisfied. In particular, BitWeaving/V divides all the bits of k words in a segment into $\lceil k/G \rceil$ *bit groups*, each of which is associated with a memory region to store sequential bits, where G is the number of bits in a bit group. Figure 2b depicts an example with $G = 5$, where three memory regions are used to store these words. The running example only needs to access the first bit group which consists of the first five bits in the memory region MR_1. Thus, compared with VBP, Bitweaving/V does not need to load the other three words into CPU cache, so Bitweaving/V has the potential to save memory bandwidth. Actually, VBP is a special case of BitWeaving/V with only one bit group and $G = k$.

ByteSlice. The byte-level columnar layout ByteSlice [11] vertically distributes bytes of a k-bit code across $\lceil k/8 \rceil$ memory regions, as the minimum bank width of a SIMD register is 8-bit in modern CPUs. ByteSlice is like BitWeaving/V in a sense that both leverages vertical and horizontal partitioning. However, ByteSlice uses the basic unit of byte, instead of bits, when vertically partitioning codes. Therefore, ByteSlice can fully leverage the data-level parallelism inside SIMD instructions. Figure 2c shows the transposition under the ByteSlice layout. In particular, each 13-bit code is partitioned into two memory regions.

2.2 Memory Efficient Storage Layouts on CPUs

Figure 3a shows an example with 16 6-bit codes evaluating the predicate $v < 100111_2$ under BitWeaving/v. The first code is "100000" in the first column. Its first two bits "10" are evaluated first. Its outcome has not been determined yet, since they are equal to the first two bits "10" of the literal. Therefore, it proceeds to the second two bits. It can safely terminate since they are different, then the evaluation proceeds to the second code. The bits which are evaluated are marked gray in the figure. Accordingly, we compute the access probability s after processing t significant bits for each code. The calculation is based on the assumption from the previous work [11]: the probability of a code matching the constant c at any bit position is 0.5. After scanning the most significant t bits of one code, its outcome is not determined only when they match the corresponding t bits of the literal, and then s is 0.5^t. Furthermore, when applying the early stopping technique to modern CPUs, its access probability is strongly associated with the hardware characteristics: width of vector register (W) and cache line size (CL).

From Vector Register Point of View. The CPU evaluates four codes at a time in a segment which is implemented with a vector register (W = 8 bits). So it requires four vector registers (①, ②, ③ and ④) to accommodate 16 codes. The evaluation can safely terminate when all the codes in the same vector register satisfy the intra-predicate cut-off condition introduced by early stopping technique. Figure 3a shows that ①, ②, ③ or ④ requires two, three, one or two evaluation rounds, respectively. Accordingly, its associated access probability s_v of each layout is shown in Table 1, since the number of codes in a vector register is W, $\frac{W}{G}$ or $\frac{W}{8}$ for VBP, BitWeaving/V or ByteSlice, respectively.



(a) A code-centric view (b) A memory-region-centric view

Fig. 3: An example predicate: $v < 100111_2$ from a code-centric view to a memory-region-centric view. Such a transformation is the key idea of our cost model. Essentially, non-linear factors like cache miss and branch misprediction can be easily expressed in a memory-region-centric view, in terms of access probability s.

From Cache Line Point of View. The memory is loaded into the cache hierarchy in cache-line-sized chunks.[2] Suppose the size (CL) of one cache line is 16 bits. Analogously, only when all the codes in one cache line can be skipped, a reduction in memory bandwidth can be achieved. The corresponding number of codes in a cache line is CL, $\frac{CL}{G}$ or $\frac{CL}{8}$ for VBP, BitWeaving/V or ByteSlice, respectively. Therefore, the associated access probability (s_c) is calculated accordingly, as shown in Table 1.

3 Hybrid Empirical/Analytical Model

In this section, we propose a hybrid empirical/analytical performance model, which demonstrates the underlying performance characteristics of conjunctive predicates under memory-efficient storage layouts.[3] We start with the design methodology, followed by basic execution patterns and the design details of the hybrid model. The detailed experimental setup can be found in Subsection 6.1.

3.1 Design Methodology

In this subsection, we summarize two concrete challenges and then present the overall design methodology of our hybrid cost model about how to address two challenges **H1** and **H2**.

H1: Unclear Compound Effect of Cut-off Conditions. The performance characteristics of conjunctive predicates under the compound effect of inter-predicate and intra-predicate cut-off conditions are still unclear due to its flexible execution model. For example, one segment (e.g., ①) in Figure 3a) of codes is evaluated, and then we proceed to the next segment (e.g., ②). Within a segment, the evaluation result on one memory region (e.g., MR_1) determines whether the evaluation on the next memory region (e.g., MR_2) is required or not under early stopping technique that explores the intra-predicate cut-off condition. Moreover, the input filter (e.g., 1011) can provide inter-predicate cut-off conditions. Thus, the whole segment ② does not need to be evaluated. To make things worse, the input filter varies with conjunctive predicates.

H2: Various CPU Platforms. We want to apply our cost model to predict the performance of conjunctive predicates on various CPUs, whose hardware characteristics (e.g., branch misprediction and cache miss) have not been well analyzed due to their non-linear factors. Due to the fact that CPU vendors like Intel do not unveil its hardware details, we are not able to know how aggressive the hardware prefetcher is and how smart the branch predictor is.

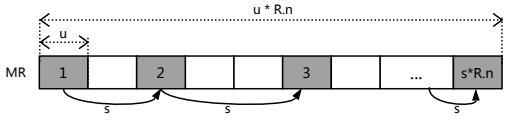


Fig. 4: Basic execution pattern: $stpr(MR, u, s)$

Our Approach. We propose a hybrid empirical/analytical performance model to address the above challenges. The key idea behind our hybrid model is to change the angle of view: from a code-centric view in Figure 3a to a memory-region-centric view in Figure 3b.[4] Therefore, the compound effect of two cut-off conditions (**H1**) on a memory region can be explicitly expressed in terms of access probability on the memory region. As such, the cost of conjunctive predicates is modeled as the *sum* of the cost on each involved memory region, where the cost on each memory region is measured when running in a *standalone way* (Subsection 3.4). The *a-priori* knowledge behind the sum comes from the sequential execution between any two consecutive memory regions due to their dependency within a segment, and the standalone way comes from the independence between any two segments.

The performance characteristics on each memory region can be covered by two basic execution patterns (Subsection 3.2) such that the performance of the interested conjunctive predicates can be the sum of the cost of the instantiated basic execution pattern on each interested memory region. Since the basic execution pattern has only one variable, we can employ the black-box approach (i.e., running microbenchmark on the targeted CPU) to sufficiently learn the relationship between the performance and the input variable (i.e., access probability s) for each basic execution pattern. As such, we can accurately model the compound effect of branch misprediction and cache miss, which are two non-linear factors on the targeted CPU. Besides, our cost model can easily apply to other CPUs (**H2**), as we only need to re-run the microbenchmark on other CPUs. The related notations are summarized in Table 2.

3.2 Basic Execution Patterns

Inspired by the generic database cost model [5], [24], [27], we highlight two basic execution patterns as building blocks of our hybrid analytical/empirical cost model, which is dedicated for conjunctive predicates under memory-efficient storage layouts.

2. Modern cache hierarchy is more aggressive due to the impact of hardware prefetcher. Instead of fetching a 64 bytes cache line each time, it loads data and instructions into the cache in blocks of 128 bytes, indicating that the adjacent cache line is loaded automatically. Take the execution pattern *stpr* on MR_2 as an example. One accessed code can cause more than two cache lines (1024 bits) loaded into CPU cache.

3. We mainly focus on BitWeaving/V and ByteSlice. VBP is omitted since VBP is a special case of BitWeaving/V (only one bit group).

4. It is analogous to the transformation from *time domain* to *frequency domain*.

Model Parameter	Definition	Source
$stpr$	One basic execution pattern: sequential traversal with probable read	Basic execution pattern
sts	One basic execution pattern: sequential traversal with store	Basic execution pattern
MR	Memory region. (sts operates only on MR_0, while $stpr$ operates on memory region MR_i, where $i \geq 1$)	Basic execution pattern
u	Memory access granularity (bits). For ByteSlice, u = 8; For BitWeaving, $u = G$ (number of bits in one bit group).	Basic execution pattern
s	Access probability. In each memory region, both cut-off conditions are explicitly expressed in terms of s.	Basic execution pattern
U_{wr}^{comp}	Unit cost of computation of $sts(MR_0, 1b, 1)$ on MR_0, e.g., 0.006 ns/code on the Haswell CPU	Empirical Model
U_{wr}^{mem}	Unit cost of memory of $sts(MR_0, 1b, 1)$ on MR_0, e.g., 0.0083 ns/code on the Haswell CPU	Empirical Model
TPC_{rd}	Peak read bandwidth from external memory, e.g., 30GB/s on the Haswell CPU	Empirical Model
TPC_{wr}	Peak writing bandwidth from external memory, e.g., 15GB/s on the Haswell CPU	Empirical Model
$UC(s)$	Unit cost of computation of $stpr$ with varying access probability s	Empirical Model
$UM(s)$	Unit cost of memory of $stpr$ with varying access probability s	Empirical Model
$C_{rd}(s)$	Number of memory bits required by each code for $stpr$	Empirical Model
T, T^{comp}, T^{mem}	The estimated value of total time, computation time, memory access time	Analytical Model
N_c	Number of CPU cores used	Analytical Model
U^x	The total unit cost of computation or memory, $x \in \{comp, mem\}$	Analytical Model
$U_{rd}^x(j)$	The unit cost of computation or memory for the j-th memory region, $j \geq 1$, $x \in \{comp, mem\}$	Analytical Model
s_f	Selectivity of input filter, used to explore inter-predicate cut-off condition	Analytical Model
$P^x[j]$	Unit cost of computation or memory of the j-th predicate of conjunctive predicates, $x \in \{comp, mem\}$	Analytical Model

TABLE 2: Summary of parameters

Sequential Traversal with Probable Read (stpr). It sweeps over the memory region MR and each time reads u bits with the access probability s, abbreviated to $stpr(MR, u, s)$, as shown in Figure 4. The memory region is traversed in order so that no item is referenced more than once, and its access probability s is determined by inter-predicate and intra-predicate cut-off conditions. ByteSlice (or BitWeaving/V) is equivalent to the case with u = 8 (or G), where G is the number of bits of each code in one bit group.[5]

Sequential Traversal with Store (sts). The basic execution pattern sts is dedicated for the result bit vector stored in the memory region MR_0. After the outcome of each code is determined, its result bit is sequentially written to MR_0, where each code has one bit to indicate whether the code satisfies the predicate (1) or not (0). Thus, u is 1 bit and s is 1. It is abbreviated to $sts(MR_0, 1b, 1)$.

3.3 Empirical Model

The empirical model estimates the cost of each basic execution pattern on various CPUs. Such an empirical approach can easily capture all the non-linear dynamics and relations from two cut-off conditions on various CPUs, while keeping the effort reasonably small. In particular, we focus on estimating the *unit cost* of each basic execution pattern, where the unit cost represents the average cost, in terms of computation and memory traffic, required by each code. In the following, we estimate the unit cost for *sts* and *stpr*.

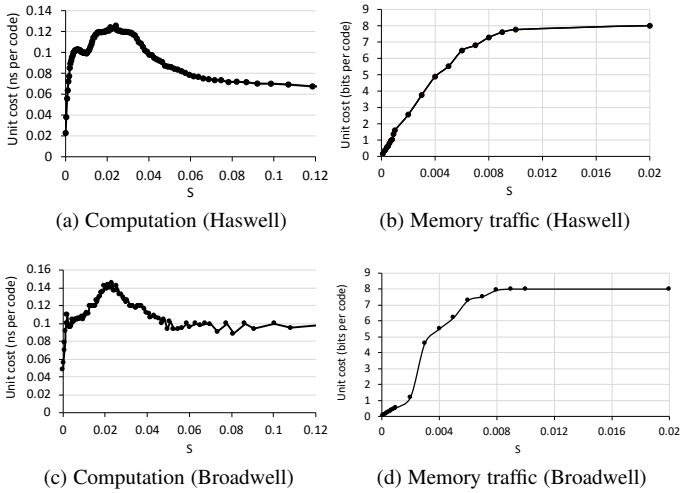
3.3.1 Estimating Unit Cost for Sts

Since the execution pattern of $sts(MR_0, 1b, 1)$ is fixed and has no variable, its unit cost of computation U_{wr}^{comp} (or memory U_{wr}^{mem}) is constant. They can be easily determined from calibrations.

Estimating U_{wr}^{comp}. We benchmark the sequential memory writing operations, using 64-bit store instruction which is used in real implementation. U_{wr}^{comp} is 0.006 ns/code on the Haswell CPU, while 0.0063 ns/code on the Broadwell CPU.

Estimating U_{wr}^{mem}. U_{wr}^{mem} is calculated to be u divided by TPC_{wr}, where TPC_{wr} is the peak writing external memory traffic handled by the memory subsystem per ns. TPC_{wr} is determined by the calibrations. In our experiments, we measure the elapsed time of the sequential memory writing operations (using 256-bit AVX2 instruction), and then calculate the peak memory bandwidth accordingly: 15GB/s on the Haswell CPU and 30GB/s on Broadwell CPU. Therefore, U_{wr}^{mem} is estimated to be 0.0083 ns/code (or 0.0041 ns/code) on the Haswell (or Broadwell) CPU.

5. Though the bits of each code are stored separated under the BitWeaving/V layout, we use the sequential case to approximate for convenience. For example, the example predicate in Figure 3a can be used to approximate the BitWeaving/V layout with $G = 2$.



(a) Computation (Haswell) (b) Memory traffic (Haswell)

(c) Computation (Broadwell) (d) Memory traffic (Broadwell)

Fig. 5: Unit cost of $stpr$ with varying s on the two targeted CPUs. Broadwell is a bit slower than Haswell under $stpr$. Two CPUs have different prefetching policies. Haswell is more aggressive when s is less than 0.002. Broadwell is more aggressive when s is larger than 0.003. Note, different CPUs with the same generation can have slightly different unit cost due to frequency difference, while CPUs across generations have obviously different curve trends.

3.3.2 Estimating Unit Cost for Stpr

Though *stpr* has only one variable (i.e., access probability s), it is difficult to develop an analytical model to accurately predict its accurate performance with varying probability s on various CPUs, due to the non-linear interference between branch mis-predication penalty and cache miss latency. To make things worse, the CPU vendors, e.g., Intel, do not unveil their implementation details. Therefore, we resort to develop a microbenchmark to quantitatively examine the relationship between unit cost and s, which varies from 0 to 1 with the step 0.001. Such a microbenchmarking method takes into account all the hardware characteristics (e.g., branch misprediction and cache miss) of the targeted CPU via running the real microbenchmark on it. Therefore, it is simple yet accurate, while maintaining effort reasonably small.

```
vector v_sum = {0, 0,..., 0};//Initialize to 0
vector v_con = {7, 7,..., 7};//For comparison

for (i = 0; i < MR.n/32; i++) {
  if (sel[i%16K] <= C){ //Control selectivity
   vector v_data = ((vector*)R)[i];
   vector v_cmp = vec_cmpgt(v_data, v_con);
   v_sum = vec_or(v_sum, v_cmp);
  }
}
```

Listing 1: Microbenchmark *stpr* with vector instruction

Microbenchmarking. Suppose a data region *MR* contains *MR.n* items, each of which contains u bits. Its cardinality *R.n* is 2^{32}, so the *MR* is out of last level cache and the initialization overhead becomes trivial. Suppose the microbenchmark is implemented with W-bit vector instructions with u-bit bank, as shown in List 1. In order to accommodate varying selectivity s, we introduce an inner array sel and one parameter C. The array sel contains 16*1024 chars, each of which is uniformly distributed random value from 0 to 255. The array can stay in L1 cache after the first-round reference, so its overhead is trivial, in term of computation time and external memory access. The parameter C is used to control the input selectivity s. Note C controls whether a memory chunk (containing 32 items) is accessed or not. Due to the fact that all the banks of a vector register are processed simultaneously when only one bank (code) needs to be processed, the relationship between C and s is illustrated in Equation 1.

$$C = \lceil 255 \times (1 - (1 - s)^{32}) \rceil \tag{1}$$

Estimating Unit Cost of Computation $UC(s)$. We run the microbenchmark and then measure the relation between s and $UC(s)$ on both CPUs, as shown in Figures 5a, 5c. When s is larger than 0.12, $UC(s)$ is stable since it almost evaluates all the elements in this memory region due to the impact of SIMD instructions. Therefore, we omit this range in Figures 5a, 5c. Now we employ the curve fitting to capture the relation between s and $UC(s)$ on the Haswell CPU.[6] We choose to approximate the $UC(s)$ by a cheaper piecewise approximation, as shown in Equation 2. The fitting result shows that the approximation works well with the metric (goodness of fit R^2) larger than 0.97 for each curve.

$$UC(s) = \begin{cases} 388114s^3 - 7041.8s^2 + 40.68s + 0.027, & 0 \le s \le 0.01 \\ 2390.5s^3 - 277.2s^2 + 8.85s + 0.038, & 0.01 < s \le 0.08 \\ -12.37s^3 + 5.9175s^2 - 0.94s + 0.116, & 0.08 < s \le 0.12 \\ 0.0619, & s > 0.12 \end{cases} \tag{2}$$

Estimating Unit Cost of Memory $UM(s)$. $UM(s)$ is estimated to be $C_{rd}(s)$ divided by the peak memory read bandwidth TPC_{rd}, where $C_{rd}(s)$ denotes the number of memory bits required by each code for $stpr$ in Equation 3.

$$UM(s) = \frac{C_{rd}(s)}{TPC_{rd}} \tag{3}$$

We estimate TPC_{rd} from calibrations. In particular, we measure the elapsed time of the sequential memory reading operations (using 256-bit AVX2 instruction), and then calculate the peak memory bandwidth accordingly: 30GB/s on the Haswell CPU and 59GB/s on the Broadwell CPU.

We estimate $C_{rd}(s)$ from running the microbenchmark on real CPUs. In particular, obtain the training pairs (s, $C_{rd}(s)$), as shown in Figures 5b, 5d. We employ the Intel Performance Counter Monitor [36] to obtain the exact amount of data read from main memory into cache hierarchy, since not only targeted data are read into memory hierarchy, but also the extra traffic from hardware prefetcher, page table loads et al. All the memory traffic competes for the precious memory bandwidth. We also use the curve fitting method to determine this relationship (s, $C_{rd}(s)$). Then, we develop a quadratic regression model to predict the value of $C_{rd}(s)$ with varying s (from 0 to 0.012), as shown in Equation 4. The fitting result shows that the regression model works well with the metric (goodness of fit R^2) equal to 0.9991.

$$C_{rd}(s) = \begin{cases} 0, & s = 0 \\ -70302.4s^2 + 1477.84s - 0.0096, & 0 < s < 0.012 \\ 8, & s \ge 0.012 \end{cases} \tag{4}$$

3.4 Analytical Model

The analytical model predicts the performance of various conjunctive predicates under memory-efficient storage layouts in both single- and multi-threaded scenarios and is applicable to various CPU platforms. It is calculated to be the maximal value of computation time T^{comp} and memory access time T^{mem}, as shown in Equation 5. The computation time is T^{comp}/N_c when N_c CPU cores are used to evaluate predicates simultaneously. T^{mem} is measured when all the memory transactions are served with the peak achievable memory bandwidth. It is useful for the multi-threaded implementation, where the memory bandwidth can be the main bottleneck, as all the threads can compete for precious memory bandwidth. The computation (or memory) time is calculated to be the total unit cost (U^x) multiplied by the number ($R.n$) of codes, depicted in Equation 6, where $x \in \{comp, mem\}$.

$$T = \max\left(\frac{T^{comp}}{N_c}, T^{mem}\right) \tag{5}$$

$$T^x = U^x \times R.n \tag{6}$$

In the following, we predict U^x for a single predicate (Subsection 3.4.1) and conjunctive predicates (Subsection 3.4.2).

3.4.1 Evaluating U^x of One Single Predicate

For a single predicate, we only consider the intra-predicate cut-off condition introduced by the early stopping technology. Therefore, U^x is estimated to be the sum of the unit cost of computation or memory on each involved memory region under the effect of intra-predicate cut-off condition, as shown in Equation 7. In the following, we illustrate how to compute the unit cost on each memory region.

$$U^x = U^x_{wr} + \sum_{j=1}^{\lceil k/u \rceil} U^x_{rd}(j) \tag{7}$$

On MR_0, each code writes one bit back to the result bit vector. Therefore, its unit cost U^x_{wr} is fixed and directly obtained from the empirical model, as shown in Subsection 3.3.1.

On MR_1, its execution pattern sequentially sweeps over MR_1 since the first u bits of each code have to be scanned before its outcome is determined. Thus, its execution pattern is abbreviated to $stpr(MR_1, u, 1)$ and $U^{comp}_{rd}(1)$ is $UC(1)$ while $U^{mem}_{rd}(1)$ is $UM(1)$, as shown in Subsection 3.3.2.

6. We can easily apply the same curve fitting to the pair obtained on the Broadwell CPU. In the following, we omit the numbers obtained on the Broadwell CPU due to page limit.

On MR_j ($j > 1$), its execution pattern can utilize the early stopping technique to explore the intra-predicate cut-off condition and thus it conditionally loads u bits of each code on MR_j with the access probability $s = 0.5^{u*(j-1)}$.[7] The intuition is that only when the evaluation of the first $(j-1)*u$ bits does not have the definite comparison result, it proceeds to the j-th memory region. Therefore, its execution pattern is abbreviated to $stpr(MR_j, u, 0.5^{u*(j-1)})$ and $U_{rd}^{comp}(j)$ is $UC(0.5^{u*(j-1)})$ while $U_{rd}^{mem}(j)$ is $UM(0.5^{u*(j-1)})$.

To sum up, U^{comp} and U^{mem} are evaluated as shown in Equations 8, 9.

$$U^{comp} = U_{wr}^{comp} + \sum_{j=1}^{\lceil k/u \rceil} UC(0.5^{u*(j-1)}) \tag{8}$$

$$U^{mem} = U_{wr}^{mem} + \sum_{j=1}^{\lceil k/u \rceil} UM(0.5^{u*(j-1)}) \tag{9}$$

3.4.2 Evaluating U^x of Conjunctive Predicates

We evaluate conjunctive predicates using a column-first execution model, which evaluates predicates sequentially. Therefore, U^x of N conjunctive predicates is calculated to be the sum of the unit cost $P^x[j]$ of the j-th predicate, where j is from 1 to N, as shown in Equation 10.

$$U^x = \sum_{j=1}^{N} P^x[j] \tag{10}$$

Evaluating $P^x[1]$. The first predicate does not have any input filter to explore inter-predicate cut-off condition, so $P^x[1]$ is exactly the same as in Subsection 3.4.1.

Evaluating $P^x[j](j \geq 2)$. The j-th predicate not only exploits its intra-predicate cut-off condition, as well as inter-predicate cut-off condition from its input filter. The filter comes from the fact that codes which do not satisfy the previous predicates do not need to be evaluated by the current predicate. Suppose the selectivity of the input filter is $s_f[j]$.[8] Now we compute $Z^x[j]$ of the j-th predicate under compound effect of two cut-off conditions. As expected, $Z^x[j]$ is estimated to be the sum of the unit cost on each related memory region.

For MR_0, it does not change the access probability s since it will write one bit per code back to main memory regardless of the content of the input filter bit vector.

For MR_j ($j \geq 1$), s becomes $s_f[j] * 0.5^{u*(j-1)}$, where $s_f[j]$ comes from the inter-predicate cut-off condition and $0.5^{u*(j-1)}$ comes from the intra-predicate cut-off condition.[9]

For sum up, the unit cost of computation ($P^{comp}[j]$) or memory ($P^{mem}[j]$) is evaluated as shown in Equations 11, 12.

$$P^{comp}[j] = U_{wr}^{comp} + \sum_{i=1}^{\lceil k/u \rceil} UC(s_f[j] * 0.5^{u*(i-1)}) \tag{11}$$

$$P^{mem}[j] = U_{wr}^{mem} + \sum_{i=1}^{\lceil k/u \rceil} UM(s_f[j] * 0.5^{u*(i-1)}) \tag{12}$$

7. This is based on one assumption that the input table is uniform. When the input table is skewed, s can be different in Equations 8, 9, 11, 12. We leave the calculation of s for the skewed dataset to our future work.

8. $s_f[j]$ is equal to the combined selectivity of of j-1 predicates (from 1 to j-1). Based on the independence assumption among predicates, $s_f[j] = \prod_{i=1}^{j-1} ps(i)$, where $ps(i)$ represents the selectivity of the i-th predicate. For example, $s_f[2]$ is the selectivity $ps(1)$ of the first predicate.

9. Similarly, the probability becomes $(1 - s_f) * 0.5^{u*(j-1)}$ for disjunctive predicates.

4 Understand Conjunctive Predicates Via Hybrid Cost Model

In this section, we aim to understand the characteristics of conjunctive predicates under memory-efficient storage layouts on modern CPUs via our hybrid cost model. First, we use our cost model to predict the performance of conjunctive predicates (Subsection 4.1). Second, we do the performance profiling to unveil the underlying characteristics so as to motivate the further optimization of conjunctive predicates on these storage layouts (Subsection 4.2).

4.1 Evaluation of Hybrid Cost Model

We evaluate our hybrid model under ByteSlice with two cases: one predicate and two conjunctive predicates. Assume both codes and constant c are 17-bit, indicating $\lceil 17/8 \rceil = 3$ memory regions are required to store codes.[10] The cardinality is 2^{32}, so its size is larger than the capacity of last level cache.

One Single Predicate. The hybrid empirical/analytical cost model computes the overall unit cost of computation to be the sum of the unit cost from each memory region, with the unit cost breakdown shown in Figures 6a, 6c. We observe that the actual unit cost ("actual") roughly matches the predicted unit cost ("predicted") from our cost model, with relative error of 3%. Another interesting observation is that a single CPU on the Broadwell CPU is a bit slower than that on the Haswell CPU, as the frequency of the Haswell CPU is higher. Additionally, we also compare the memory traffic, whose breakdown is illustrated in Figures 6b, 6d. "ByteSlice" depicts the memory traffic estimation from the existing work [11]. The experimental result shows that the relative error of the hybrid model (3.6%) is significantly less than that of "ByteSlice" (26.4%), since our hybrid model considers the effect of modern memory hierarchy, e.g., hardware prefetcher, while the work [11] only does the theoretical computation.

Two Predicates. The selectivity of the first predicate $\widehat{p(1)}$ is 50%, while the selectivity of the second predicate $\widehat{p(2)}$ varies, from 0.5 to 0.0005. Then, the optimal evaluation order represents the case with $\widehat{p(2)}$ evaluated first, while the worst evaluation order evaluates $\widehat{p(1)}$ first. Our hybrid model can well captures the performance trend of both optimal and worst evaluation orders on the Haswell CPU, as shown in Figure 7. Specifically, our hybrid model can predict that when the selectivity of $\widehat{p(2)}$ decreases, the optimal evaluation order increases the overall performance while the worst evaluation order keeps stable. The performance difference can reach up to 41.5% with only two predicates.

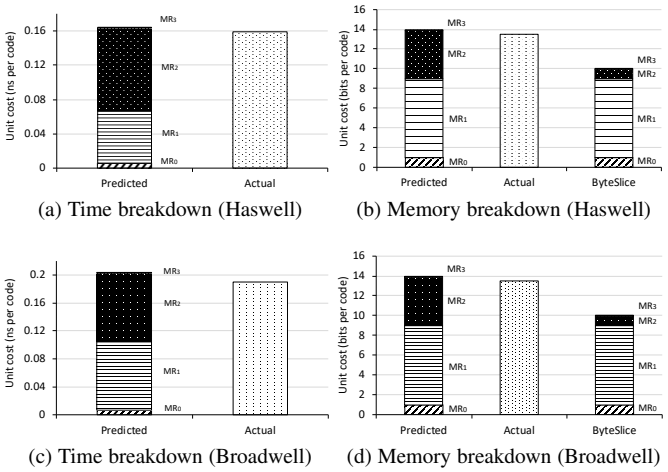
4.2 Insights from Performance Profiling

To have a better understanding on the performance characteristics of conjunctive predicates on modern CPUs, we obtain three insights from the compound effect of intra-predicate and inter-predicate cut-off conditions based on our hybrid cost model. Quantitatively, we make three observations that guide the further optimization of conjunctive predicates.

4.2.1 Insight about Intra-predicate Cut-off Condition

Now we quantitatively analyze the performance characteristics of the single predicate which only explores the intra-predicate cut-off condition.

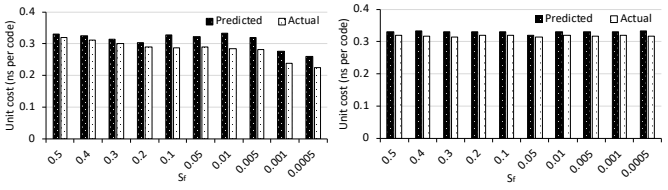
10. It can be easily applied to different code bitwidth that corresponds to different number of memory regions.



(a) Time breakdown (Haswell) (b) Memory breakdown (Haswell)

(c) Time breakdown (Broadwell) (d) Memory breakdown (Broadwell)

Fig. 6: Cost model evaluation for one predicate on both CPUs



(a) Optimal evaluation order (b) Worst evaluation order

Fig. 7: Cost model evaluation of two predicates

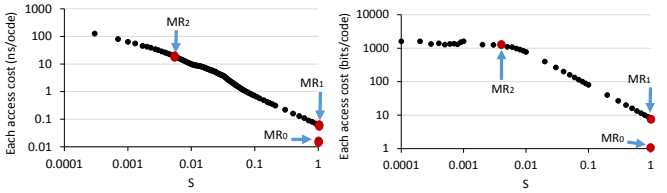
Observation 1: Selective Traversal Can Be Slower than Sequential Traversal. As depicted in Figure 5a,5c, the unit cost $U_{rd}^{comp}(2)$ of computation on MR_2 (s = 0.0039) is larger than that $U_{rd}^{comp}(1)$ on MR_1 (s = 1). $U_{rd}^{comp}(1)$ is 0.061 ns/code on MR_1, where s is 1 due to the fact that the first byte is always evaluated. However, $U_{rd}^{comp}(2)$ is surprisingly 0.096 ns/code on MR_2, where s is only 0.0039 (0.5^8) due to the effect of intra-predicate cut-off condition. Its underlying reason is that each real access on MR_2 is very expensive due to high cache miss and branch misprediction penalties, while each real access on MR_1 is cheap due to its correct branch predication and cache hit.

4.2.2 Insight about Inter-predicate Cut-off Condition

Now we quantitatively analyze the performance characteristics of the inter-predicate cut-off condition on the predicate, which also explores the intra-predicate cut-off condition. The inter-predicate cut-off condition is introduced by the input filter whose selectivity is s_f. More specifically, we quantitatively examine its effect on the execution pattern on each memory region[11] and make the following two observations.

Observation 2: Reduction of Low-cost Instructions can always Degrade Performance. In particular, the reduction of the access probability on MR_1 can always degrade the performance. From Figure 5a, we can observe that $UC(s)$ is larger than $UC(1)$, when $0.001 \leq s < 1$. It means that the input filter can degrade the performance of the execution pattern on MR_1 when its selectivity s_f is larger than 0.001, since each code is evaluated (s = 1) for the execution pattern on MR_1, which are highly optimized with the help from hardware prefetcher (hit in cache) and branch predictor (correct predication). The input filter potentially incurs high branch misprediction and cache miss penalties, which can not be amortized by instruction reduction.

11. For the execution pattern on MR_0, the result bit of each code is stored back to the memory (s = 1) regardless of the input filter. In other words, the inter-predicate cut-off condition has no effect on MR_0.



(a) Computation (b) Memory Traffic

Fig. 8: Average cost for each accessed code of $stpr$. The average cost differs by three orders of magnitude among memory regions.

Observation 3: Reduction of High-cost Instructions Significantly Increases Performance. In particular, the slight reduction of its access probability on MR_j can result in the significant reduction of its unit cost, where $j > 1$. We have similar observations for all the memory regions MR_j, where $j > 1$. We use the performance characteristics on MR_2 as an example. In order to quantitatively show the effect of inter-predicate cut-off condition, we introduce one metric *average computation cost for each accessed code* (O^{comp}), which is defined to be $UC(s)$ divided by s. We can observe that O^{comp} on MR_2 is 417 times larger than that on MR_1, as shown in Figure 8a, since MR_2 suffers from high branch mispredication and cache miss penalties for each access. Similarly, the average memory cost O^{mem} for each accessed code on MR_2 is roughly 153 times larger than that on MR_1, where O^{mem} is defined to be $UM(s)$ divided by s, as shown in Figure 8b. Each code access on MR_2 may load more than two cache lines (1024 bits) due to the effect of hardware prefetcher. In contrast, O^{mem} on MR_0 (or MR_1) only requires 1 bit (or 8 bits) per code. Suppose the predicate has an input filter (s_f = 0.5), s on MR_2 shifts from the original $0.5^8 = 0.0039$ to 0.00195. Consequentially, $U_{rd}^{comp}(2)$ can be reduced from 0.096 to 0.065 ns/code (benefiting single-threaded implementation), while $U_{rd}^{mem}(2)$ shifts from 4.88 to 2.56 bits/code (benefiting multi-threaded implementation).

Put it All Together. The intuition of our findings is that *an instruction has extremely different evaluation cost and then our optimization direction should focus on reducing high-cost instructions while leaving low-cost instructions untouched*. In particular, the reduction of low-cost instructions has the potential to degrade performance due to introducing branch misprediction and cache miss (*Observations 1 and 2*). However, the reduction of evaluated instructions that have high branch misprediction and cache miss penalties significantly improves performance (*Observation 3*). The above three observations serve as the design guidelines of the further optimization process under memory-efficient storage layouts on modern CPUs.

5 Order-oblivious Execution Scheme Hebe (Optimization)

5.1 Design Methodology

Based on the above three observations, we present the motivating example for the proposed execution scheme Hebe which is order-oblivious and high-performance. Specifically, we demonstrate its advantage over the column-first execution model, in terms of *raw performance* and *order sensitivity*. In order to concretely demonstrate the difference, we import $\bigoplus$ [24], [27] to indicate the sequential execution (not commutative), like &&. $\bigodot$ indicates the concurrent execution (commutative), like &. Suppose we have

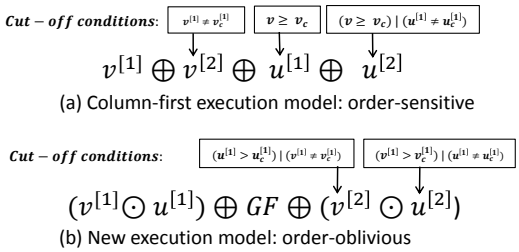


Fig. 9: Execution schemes: from column-first to Hebe.

a conjunction of two predicates: $\widehat{p(1)} \wedge \widehat{p(2)}$.

$\widehat{p(1)} : v < v_c$	$\widehat{p(2)} : u < u_c$
$v_1 = (\underline{11111111}\ 11111)_2$	$u_1 = (\underline{11110000}\ 00000)_2$
$v_2 = (\underline{10101010}\ 00000)_2$	$u_2 = (\underline{11111010}\ 11111)_2$
$v_3 = (\underline{00000000}\ 11001)_2$	$u_3 = (\underline{00110000}\ 10011)_2$
$v_4 = (\underline{10101010}\ 00010)_2$	$u_4 = (\underline{11111111}\ 10101)_2$
$v_c = (\underline{10101010}\ 00111)_2$	$u_c = (\underline{11110000}\ 01101)_2$

5.1.1 Column-first Execution Model

Without loss of generality, suppose $\widehat{p(1)}$ is evaluated first in column-first execution model. After evaluating the first iteration which compares the first bytes (underlined) of v and v_c, v_1 and v_3 have the definite result due to $v_1^{[1]} > v_c^{[1]}$ and $v_3^{[1]} < v_c^{[1]}$. Therefore, they can skip the next iteration under early stopping technique. However, v_2 and v_4 need to proceed to the next iteration due to $v_2^{[1]} = v_4^{[1]} = v_c^{[1]}$. After evaluating $\widehat{p(1)}$, we can obtain its result bit vector (0, 1, 1, 1), indicating that only v_1 does not satisfy the predicate. When evaluating $\widehat{p(2)}$ with the previous bit vector, the codes u_2, u_3 and u_4 need to compare their first bytes $u^{[1]}$ with $u_c^{[1]}$. To better illustrate, the evaluated bytes are marked blue. In general, $\widehat{p(1)} \wedge \widehat{p(2)}$ can be expressed in Figure 9a, where $u^{[1]} \bigoplus u^{[2]}$ depicts the sequential evaluation on $u^{[1]}$ and $u^{[2]}$ and cut-off conditions for each byte are also illustrated.

Order Sensitivity. As indicated by our cost model, the performance of conjunctive predicates is determined by the access probability s on each involved memory region, and s is determined by two cut-off conditions. If $\widehat{p(2)}$ is evaluated first, the expression becomes $u^{[1]} \bigoplus u^{[2]} \bigoplus v^{[1]} \bigoplus v^{[2]}$. As such, cut-off conditions for each memory region can be significantly changed. For example, the cut-off condition for $u^{[1]}$ changes from $v \geq v_c$ to null. We conclude that the overall performance is sensitive to the evaluation order of predicates.

Raw Performance. We evaluate the total cost (marked blue). In particular, seven low-cost comparisons are performed on the first bytes: four in $\widehat{p(1)}$ and three in $\widehat{p(2)}$, and two high-cost comparisons on the second bytes are performed.

5.1.2 New Execution Scheme: Hebe

We observe that the predicate evaluation under memory-efficient storage layout can be broken down into the evaluations on its associated memory regions. This observation motivates our new execution scheme Hebe. The key idea of Hebe is to tune the evaluation order of predicates at memory region level such that we can reduce the high-cost instructions while keeping the correctness of predicate evaluation. Figure 9b depicts the abstraction of the new execution scheme.

The first step is to perform the evaluation on $v^{[1]}$ and $u^{[1]}$ concurrently, denoted by $u^{[1]} \bigodot v^{[1]}$. The second step is to aggressively explore the inter-predicate cut-off conditions from the intermediate statuses of $v^{[1]}$ and $u^{[1]}$ (marked GF in the figure).[12] In particular, GF can add one inter-predicate cut-off condition $(u^{[1]} > u_c^{[1]})$ to the evaluation of $v^{[2]}$, together with $(v^{[1]} \neq v_c^{[1]})$ from its own intra-predicate cut-off condition. Similarly, one inter-predicate cut-off condition $(v^{[1]} > v_c^{[1]})$ is added to the evaluation of $u^{[2]}$. The probability of $(v^{[1]} > v_c^{[1]})$ is roughly the same as that of $(v > v_c)$.[13] The third step is to evaluate $v^{[2]}$ and $u^{[2]}$ if necessary. We find that no code (either v or u) needs to enter the second iteration. For example, $v_2^{[2]}$ supposes to be evaluated in column-first execution model. However, Hebe does not need to evaluate $v_2^{[2]}$ since the outcome of the second tuple has already been determined (0) due to $u_2^{[1]} > u_c^{[1]}$.

Order Sensitivity. Hebe is oblivious to evaluation order due to the following two factors. First, the memory region with the same index is evaluated interchangeably for each predicate at the same step. For example, the evaluation order of $v^{[1]}$ and $u^{[1]}$ is not important. Second, after evaluating the first memory region of each predicate, it proceeds to the second step (i.e., GF) and then enters the evaluation of the second memory region if necessary. So, Hebe is oblivious to predicate order.

Raw Performance. We evaluate its total cost (marked underlined). In particular, eight comparisons are performed on the first bytes, and no comparison is performed on the second bytes. According to *Observation 2*, its performance on the first bytes is better than that of column-first execution model. According to *Observation 3*, its performance on the second bytes is also better due to the reduction of two high-cost evaluations. Therefore, Hebe produces better performance.

5.2 Design and Implementation of Hebe

In this subsection, 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 computed once as it is shared by each segment (Lines 1-5).

For each segment, the detailed execution flow (Lines 6-31) is illustrated in three steps.

First, we initialize three W-bit segment-level status masks (Lines 7-11) for each predicate: less-than mask $\mathcal{M}_{lt}$ (0^W), greater-than mask $\mathcal{M}_{gt}$ (0^W) and equal-to mask $\mathcal{M}_{eq}$ (1^W), indicating uncertain status of a predicate.[14] Each mask consists of $W/8$ 8-bit banks, where all the eight bits in a bank are 1^8 or 0^8.

Second, codes are examined one byte (i.e., one memory region) per iteration until the cut-off condition is reached or $\max_{1 \leq i \leq N} B(i)$ iterations are finished (Lines 12-28). Before each iteration, the cut-off condition is checked for each predicate to

12. The detailed description of GF is shown in Subsection 5.2.

13. We assume that the values of 13-bit codes are uniformed distributed in the range [0, 2^{13}), so the probability of $(v > v_c)$ is $1 - v_c/2^{13} = 33.5\%$ while the probability of $(v^{[1]} > v_c^{[1]})$ is $1 - v_c^{[1]}/2^8 = 33.6\%$.

14. For ease of understanding, we 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).

Algorithm 1: PROPOSED EXECUTION SCHEME: HEBE

```
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.
/* Initialization Step.                                              */
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
   /* Iterate each segment.                                          */
6  for (each segment with codes v_{l+1} ... v_{l+W/8}) do
       /* 1, zero segment-level status masks.                       */
7      for i = 1 to N do
8          M_lt(i) = 0^W
9          M_gt(i) = 0^W
10         M_eq(i) = 1^W
11     end
       /* 2, evaluate the j-th byte.                                */
12     for j = 1 to max_{1≤i≤N} B(i) do
           /* 2.1, evaluate the i-th predicate.                     */
13         for i = 1 to N do
               /* Evaluate if cut-off condition is not met.         */
14             if (M_eq(i) ≠ 0^W) && (j ≤ B(i)) then
15                 D^[j](i) = v_load(v_{l+1}^[j](i) ... v_{l+W/8}^[j](i))
                   /* Compute byte-level state masks.               */
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))
                   /* Update segment-level state masks.             */
19                 M_lt(i) = v_or(M_lt(i), v_and(M_eq(i), M_lt(i)))
20                 M_gt(i) =
                     v_or(M_gt(i), v_and(M_eq(i), M_gt(i)))
21                 M_eq(i) = v_and(M_eq(i), M_eq(i))
22             end
23         end
           /* 2.2, compute global pruning factor M.                 */
24         M = global_filter(M_eq(1:N), M_gt(1:N), M_lt(1:N))
           /* 2.3, use M to prune N predicates.                     */
25         for i = 1 to N do
26             M_eq(i) = v_and(M_eq(i), M)
27         end
28     end
       /* 3, compute bit vector for this segment.                   */
29     M_final = final_mask(M_eq(1:N), M_gt(1:N), M_lt(1:N))
30     r = v_movemask(M_final)
31     Append r to bitvector
32 end
```

explore the cut-off possibility. 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). Note, the evaluation order of predicates here does not matter. The j-th byte in this segment is loaded into a 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* $\leq$, $\neg\mathcal{M}_{lt}$ for $>$ *or* $\geq$, $\neg\mathcal{M}_{lt}|\mathcal{M}_{gt}$ for $=$, and 1^W for $\neq$. Then, $\mathbb{M}$ is calculated 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 or not after evaluating j bytes. If the false state is detected, no further evaluation on this tuple is required. Therefore, $\mathbb{M}$ can be used to further prune the uncertain conditions (Lines 25-27) such that high-cost instructions, which suffer from branch misprediction and cache miss, can be significantly eliminated at the expense of a few low-cost arithmetic instructions (Line 24). As such, Hebe harvests performance potential of memory-efficient storage layouts and then achieves better performance on modern CPUs.

Third, after the above iterations, the final result of each tuple in this segment is determined. Then $\mathcal{M}_{lt}$, $\mathcal{M}_{gt}$ and $\mathcal{M}_{eq}$ of this segment are sent to the *final_mask* module that computes the final result mask $\mathcal{M}_{final}$ (Line 29) for this segment.[15] In particular, W-bit $\mathcal{M}_{final}$ is computed to be ANDed the output result mask of each predicate together, while the output result mask of each predicate is evaluated to be $\mathcal{M}_{lt}$ for $<$, $\mathcal{M}_{lt}|\mathcal{M}_{eq}$ for $\leq$, $\mathcal{M}_{gt}$ for $>$, $\mathcal{M}_{gt}|\mathcal{M}_{eq}$ for $\geq$, $\mathcal{M}_{eq}$ for $=$, and $\mathcal{M}_{gt}|\mathcal{M}_{le}$ for $\neq$. Then, $\mathcal{M}_{final}$ is condensed to a $W/8$-bit mask r using the *v_movemask* instruction (Line 30). Lastly, the mask (r) is appended to the result bit vector $bitvector$.

	Intel Haswell-E	Intel Broadwell
CPU	Core i7-5960X	Xeon E5-2680 v4
Cores/Threads	8 / 16	14 / 28
Frequency	3.0 GHz	2.4 GHz
SIMD	256-bit AVX2	256-bit AVX2
L3 Cache	20 MB	35MB
Memory	DDR4, 68 GB/s	DDR4, 76.8 GB/s

TABLE 3: Hardware Platforms

6 EXPERIMENTAL EVALUATION OF HEBE

6.1 Experimental Setup

Hardware Configuration. We conduct our experiments on two Intel CPUs of two generations: Haswell and Broadwell, as shown in Table 3. All the related programs are compiled using ICC 16.0.3 with the highest optimization effort -O3. In order to accurately collect the performance profiles, we use the Intel Performance Counter Monitor [36] to collect the performance counters on the program of interest.

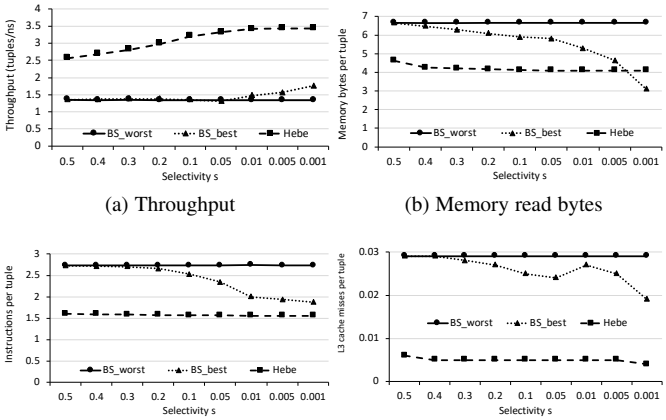
Workloads. In our experiment, there are two kinds of workloads: synthesized workload and TPC-H workload. For the synthesized workload, we create the table with different number of columns, where each column contains one billion k-bit codes. By default, values of codes are uniformly distributed in the range $[0, 2^k)$, where k is 17 by default. The corresponding advantage is that the selectivity of each predicate can be tuned so that we can analyze the performance characteristics with varying selectivity. For the TPC-H dataset, we evaluate twelve TPC-H queries (Q1, Q3, Q5, Q6, Q7, Q8, Q10, Q12, Q14, Q15, Q17 and Q19) with the scale factor (SF) of 10. The number of predicates varies from 2 to 36.

Comparison Methodology. Five implementations are used for performance comparison. The first one is Hebe (denoted as "Hebe"). Two cases come from the state-of-the-art column-first execution model [11] under ByteSlice memory layout. "BS_best" (or "BS_worst") is the implementation with an optimal (or worst) evaluation order. The other two approaches come from the SIMD-scan method [39] with the naive column store, where "Naive_best" (or "Naive_worst") is the implementation with the optimal (or worst) order.

6.2 Evaluation on Synthesized Workload

We evaluate Hebe using synthesized workload. 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,

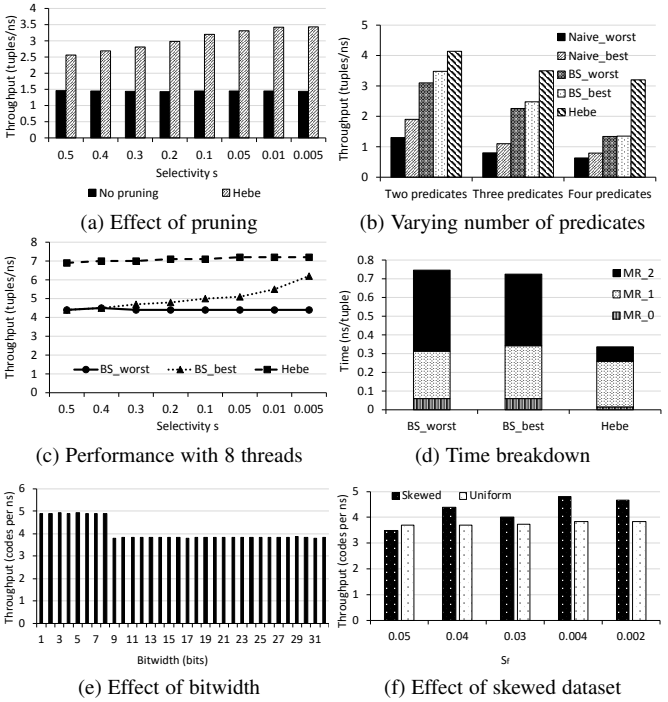
15. The fact that $\mathcal{M}_{eq}$ is pruned by $\mathbb{M}$ (Lines 25-27) will not violate the correctness, since $\mathcal{M}_{eq}$ is pruned only when the result of the tuple has already been determined. The determination comes from $\mathcal{M}_{lt}$, $\mathcal{M}_{gt}$ and is oblivious to $\mathbb{M}$.



(a) Throughput (b) Memory read bytes

(c) Instructions (d) L3 cache misses

Fig. 10: Evaluation of four conjunctive predicates



(a) Effect of pruning (b) Varying number of predicates

(c) Performance with 8 threads (d) Time breakdown

(e) Effect of bitwidth (f) Effect of skewed dataset

Fig. 11: Performance comparisons

$s(2)$, $s(3)$ and $s(4)$ are set to be 50%, and $s(1)$ varies from 50% to 0.1%. "BS_best" is the case with the evaluation order where $\widehat{p(1)}$ is evaluated firstly, while "BS_worst" is the case with the evaluation order where $\widehat{p(1)}$ is evaluated lastly.

Now, we study the performance of the conjunctive predicates $\widehat{p(1)}\&\&\widehat{p(2)}\&\&\widehat{p(3)}\&\&\widehat{p(4)}$. Figure 10a compares the throughput of three cases in single-threaded approach, in terms of tuples/ns. The x-axis (s) stands for the varying selectivity $s(1)$ of $\widehat{p(1)}$. To unveil the underlying reason of performance improvement, we also provide three performance metrics (collected from Intel Performance Counter Monitor [36]): memory read bytes, instructions, L3 cache misses. We make two observations. First, when $\widehat{p(1)}$ becomes more selective (i.e., 50% to 0.1%), the performance of "BS_best" cannot be significantly better than that of "BS_worst", since the column-first execution model does not aggressively reduce high-cost instructions on memory region MR_2. Therefore, "BS_best" still has high L3 cache miss ratio (Figure 10d), although the number of consumed instructions has already been significantly reduced (Figure 10c). Second, Hebe can achieve 89%-153% performance gain over "BS_best", since Hebe aggressively reduces high-cost instructions. Take $\widehat{p(2)}$ for example, three inter-predicate cut-off conditions ($v1^{[1]} > c1^{[1]}$, $v3^{[1]} > c3^{[1]}$ and $v4^{[1]} > c4^{[1]}$) are exploited, together with the intra-predicate condition $v1^{[2]} \neq c1^{[2]}$. In contrast, "BS_best" only exploits two cut-off conditions ($v1 > c1$, $v2^{[1]} > c2^{[1]}$). Note, Hebe is still faster than "BS_best" that requires less memory traffic when $s(1)$ is 0.1%.

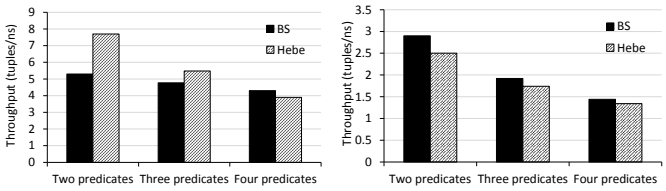
Effect of Inter-predicate Cut-off Conditions. The *global_filter* module is used to explore the cut-off conditions among predicates. "No pruning" is the case without *global_filter* module. From Figure 11a, we make two observations. First, "No pruning" achieves roughly the same performance when varying the selectivity for conjunctive predicates since the inter-predicate cut-off condition is not explored and each predicate can only benefit from its own intra-predicate cut-off condition. Second, Hebe significantly benefits from the reduction of selectivity s (from 0.5 to 0.001). In particular, the low value of s of $\widehat{p(1)}$ can reduce the access probability of the execution pattern on MR_2 of the other three predicates.

Effect of Predicate Number. Figure 11b 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 make two observations. First, as expected, two naive approaches are the slowest, since each code takes one 32-bit bank of SIMD register, indicating that naive approaches only achieve 8-way parallelism while the others achieves 32-way parallelism. Second, Hebe achieves more performance improvement over the others when the number of predicates increases, since only Hebe focuses on reducing high-cost instructions which suffer from high branch misprediction and cache miss penalties.

Effect of Code's Bitwdith. We examine the effect of the bitwidth k of each code when evaluating two conjunctive predicates under Hebe, where $s(1)$ of $\widehat{p(1)}$ is 0.01 and $s(2)$ is 0.5. Figure 11e illustrates the throughput of Hebe with varying bitwidth. We have two observations. First, the throughput stays roughly the same and high when k is less than 9, as Hebe, whose storage layout is ByteSlice, always pads each k-bit code to a 8-bit code before evaluation, where 8-bit codes are stored in one memory region that is entirely scanned. Second, the throughput also stays roughly the same when k is larger than 8, as evaluating the first two bytes is almost sufficient to determine the final predicate result, and then the high-cost instructions on the third and fourth bytes are rarely needed even when each code has four bytes.

Effect of Skewed Dataset. We examine the effect of the skewed dataset when evaluating two conjunctive predicates under Hebe, where $\widehat{p(2)}$'s dataset is always uniform. Figure 11f illustrates the throughput of Hebe with varying $s(1)$. "Skewed" represents the case that $\widehat{p(1)}$'s dataset is highly skewed, with Zipf factor z = 1.0, while "Uniform" represents the case that $\widehat{p(1)}$'s dataset is uniform. We observe that when $s(1)$ decreases, the skewed dataset leads to a slight performance fluctuation, while the uniform dataset leads to roughly the same performance. The underlying reason is that the skewed dataset provides more inter-predicate and intra-predicate cut-off conditions that are exploited by Hebe. Essentially, codes in the skewed dataset always have less probability of being equal to the predicate literal.

Effect of Multiple Threads. We study the performance of



(a) s = 0 (b) s = 1

Fig. 12: Evaluation of two corner cases.

database scans using eight threads on all the eight hardware cores on the tested CPU, and find that simultaneous multi-thread (SMT) cannot improve the overall performance due to the main bottleneck from memory bandwidth. Paralleling scans on modern CPU is easy: distributing the codes into eight chunks, each chunk running on one thread. Figure 11c shows the throughput of conjunctive predicates implemented with eight threads. The performance of conjunctive predicates is bounded by the limited memory bandwidth, so its performance is inversely proportional to the amount of memory read bytes per code, as shown in Figure 10b. We observe that Hebe can achieve up to 57% performance improvement over "BS_best".

Time Breakdown. We present the time breakdowns for conjunctive predicates with $s(1)$ equal to 20%, as shown in Figure 11d. "MR_x" indicates the aggregative time per tuple spent on the memory region MR_x from all four predicates, where x equals 0, 1 or 2. We omit the time spent on MR_3, since it is negligible compared with the overall execution time. We can observe that Hebe can significantly improve the performance of conjunctive predicates, since four inter (or intra)-predicate cut-off conditions are explored to aggregatively reduce the execution time on memory region MR_2.

Corner Cases. We study two corner cases, the selectivity of each predicate s = 0 or 1. Figure 12 shows the throughput of conjunctive predicates, whose number varies from 2 to 4. When s = 0, the column-first execution model ("BS") only evaluates the first predicate. The remaining predicates can benefit from inter-predicate cut-off condition from the first predicate. The only overhead is to check the input filter. However, Hebe evaluates the first byte of each predicate. Therefore, Hebe can be worse than the column-first execution model in Figure 12a. When s = 1, neither execution model can benefit from the inter-predicate cut-off condition. The *global_filter* module in Hebe cannot explore any inter-predicate cut-off condition, but incurs the extra logical instructions. We can observe that the overhead becomes smaller when the number of predicates becomes larger, as shown in Figure 12b.

6.3 Evaluation on TPC-H

We evaluate Hebe by using twelve queries from TPC-H benchmark. To focus on the performance of predicates, we use the technique from WideTable [23] to flatten a database schema into several denormalized tables. Then, queries with complex joins can become simple scans on the denormalized tables. Figure 13 shows the experimental results of twelve queries that contain only conjunctive predicates, no disjunctive predicates involved.[16] We make two observations.

16. To focus on the performance of scans, we follow ByteSlice [11] that the performance of the other operators, e.g., order by and group by, are not taken into account in the overall performance.

First, there are plenty of feasible evaluation orders for each query under column-first execution model. For example, there are four predicates in Q8 and the number of evaluation orders for Q8 can reach up to 4!=24. Since the evaluation order is sensitive to the overall performance [18], [33], we observe that the performance difference can be from 19% to 62%.

Second, even with the optimal evaluation order under column-first execution model "BS_best", Hebe can still achieve 39%-209% performance improvement (in terms of tuples per ns), as shown in Figure 13a. The underlying reason is that Hebe can aggressively explore the inter-predicate cut-off conditions so as to reduce high-cost instructions on the memory region MR_2 of each predicate. Therefore, the access probability on MR_2 of each involved predicate can be significantly reduced by Hebe, and then Hebe requires significantly less L3 cache misses per tuple for each query, as shown in Figure 13b. In particular, for the query Q19 that has 36 predicates, Hebe can achieve significant performance improvement since inter-predicate cut-off possibilities among 36 predicates can be aggressively explored. Another thing to be mentioned is that Hebe does not need to take into account the evaluation order of predicates.

7 Related Work

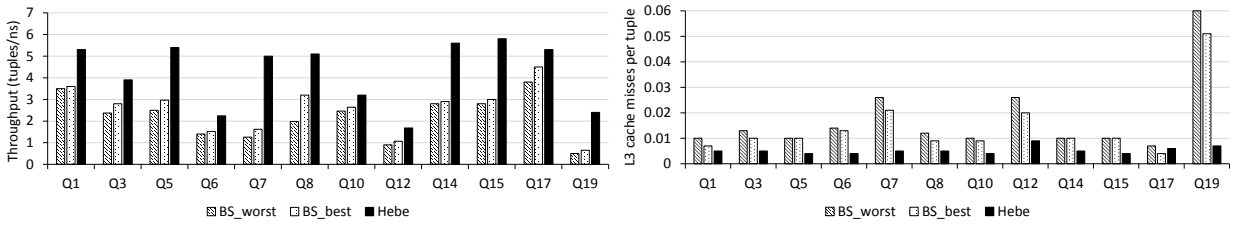
A preliminary version of this manuscript has been published in [38]. Compared with the preliminary version, this manuscript has made significant contributions in building a hybrid empirical/-analytical cost model in understanding the efficiency of different memory layouts and their execution strategies, and performing more in-depth and extensive studies on Hebe.

Memory-efficient Storage Layouts. Prior works [11], [21], [22], [23], [26], [28], [29], [30], [32], [37], [38], [40], [41] leverage memory-efficient storage layout to partition values at the bit/byte level such that predicate evaluation can fully utilize intra-cycle parallelism in modern CPUs and can exploit intra-predicate cut-off condition to reduce required memory traffic. In contrast, this work proposes a hybrid cost model to unveil performance characteristics of predicate evaluation for better understanding and then proposes an order-oblivious execution scheme Hebe for conjunctive predicates.

Attribute Groupings. Prior works [2], [12], [13], [15], [16] leverage access patterns of queries over tables to do attribute grouping so as to increase the overall query throughput. In contrast, this work does not reply on attribute grouping but explores on-the-fly cut-off possibilities to reduce required memory traffic, as well as required instructions.

Cost Models for Databases. Previous works [5], [24], [27] propose generic database cost model to predict operator execution time. Such a prediction is vital to a query optimizer that determines query execution plan for the input query. About cost model, Our work is closest to the work by Pirk et al. [27], which proposes a new access pattern *sequential traversal with conditional reads* to analytically model the performance of selective projections. In contrast, we resort to an empirical approach (i.e., directly running microbenchmark on the targeted CPU) to easily capture hardware characteristics on various CPUs. Besides, our hybrid cost model is dedicated to predicting the performance of conjunctive predicates under memory-efficient storage layouts.

Optimization of Predicate Order. Previous works [6], [17], [18], [19], [25], [33], [35] propose various cost models, which take into account branch misprediction and cache miss, to determine the optimal predicate order for the input query. Briefly, predicates can



(a) Throughput (b) L3 cache misses

Fig. 13: Evaluation of 12 TPC-H queries

be ordered by increasing selectivity or rank. Since the selectivity estimation itself can be inaccurate for an ad-hoc query, it is hard for QO to produce an optimal evaluation order. In contrast, Hebe is order-oblivious, while keeping high performance.

8 Conclusion

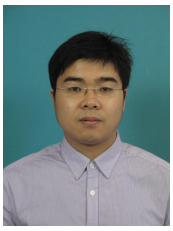
The optimization of conjunctive predicates is still critical to database queries. Recently, several memory-efficient storage layouts have been proposed to significantly accelerate database scans. However, the performance potential of such storage layouts on conjunctive predicates has not been fully harvested. In this paper, we propose a hybrid empirical/analytical cost model to fully understand these storage layouts on modern CPUs. Such understanding enables us to propose an order-oblivious execution scheme Hebe to evaluate conjunctive predicates, while maintaining high performance. With Hebe, the QO does not need to go through a sampling process to guess the optimal evaluation order in advance.

Acknowledgement. This work is supported by National Key R&D Program of China (No. 2017YFC0805000/2017YFC0805005), NSFC (No. 61732004, 61370080), the Shanghai Innovation Action Project (Grant No. 16DZ1100200), Zhejiang Natural Science Foundation (LR19F020002, LZ17F020001), National Natural Science Foundation of China (61976185), a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore.

References

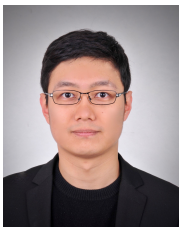
[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.
[2] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *SIGMOD*, 2016.
[3] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD*, 2005.
[4] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.
[5] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 2008.
[6] D. Broneske, V. Köppen, G. Saake, and M. Schäler. Accelerating multi-column selection predicates in main-memory - the elf approach. In *ICDE*, 2017.
[7] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, 2001.
[8] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 2010.
[9] F. Farber, N. May, W. Lehner, I. Muller, H. Rauhe, J. Dees, and S. Ag. The sap hana database: An architecture overview, 2012.
[10] Z. Feng and E. Lo. Accelerating aggregation using intra-cycle parallelism. In *ICDE*, 2015.
[11] 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*, 2015.
[12] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: A main memory hybrid storage engine. *PVLDB*, 2010.
[13] R. A. Hankins and J. M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *PVLDB*, 2003.
[14] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: Compression and bandwidth trade offs for database scans. In *SIGMOD*, 2007.
[15] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: Right shoes for a running elephant. In *SOCC*, 2011.
[16] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 2008.
[17] S. Karthik, J. R. Haritsa, S. Kenkre, and V. Pandit. A concave path to low-overhead robust query processing. *PVLDB*, 2018.
[18] F. Kastrati and G. Moerkotte. Optimization of conjunctive predicates for main memory column stores. *PVLDB*, 2016.
[19] F. Kastrati and G. Moerkotte. Generating optimal plans for boolean expressions. In *ICDE*, 2018.
[20] L. Lamport. Multiple byte processing with full-word instructions. *Communication of the ACM*, 1975.
[21] Y. Li, C. Chasseur, and J. M. Patel. A padded encoding scheme to accelerate scans by leveraging skew. In *SIGMOD*, 2015.
[22] Y. Li and J. M. Patel. Bitweaving: Fast scans for main memory data processing. In *SIGMOD*, 2013.
[23] Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *PVLDB*, 2014.
[24] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *PVLDB*, 2002.
[25] T. Neumann, S. Helmer, and G. Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE*, 2005.
[26] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, 1997.
[27] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten. Cpu and cache efficient management of memory-resident databases. In *ICDE*, 2013.
[28] H. Pirk, T. Sellam, S. Manegold, and M. Kersten. X-device query processing by bitwise distribution. In *DaMoN*, 2012.
[29] O. Polychroniou and K. A. Ross. Efficient lightweight compression alongside fast scans. DaMoN, 2015.
[30] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood. Implications of emerging 3d gpu architecture on the scan primitive. 2015.
[31] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, 2008.
[32] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. In *SIGMOD*, 2001.
[33] K. A. Ross. Conjunctive selection conditions in main memory. In *PODS*, 2002.
[34] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, 2011.
[35] D. Song and S. Chen. Exploiting simd for complex numerical predicates. In *ICDEW*, 2016.
[36] P. F. Thomas Willhalm, Roman Dementiev. Intel Performance Counter Monitor - A better way to measure CPU utilization. Technical report, Intel, https://software.intel.com/en-us/articles/intel-performance-counter-monitor, 2016.
[37] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang. Accelerating generalized linear models with mlweaving: A one-size-fits-all system for any-precision learning. *PVLDB*, 2019.
[38] Z. Wang, K. Zhang, H. Zhou, X. Liu, and B. He. Hebe: An order-oblivious and high-performance execution scheme for conjunctive predicates. In *ICDE*, 2018.
[39] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2009.
[40] W. Xu, Z. Feng, and E. Lo. Fast multi-column sorting in main-memory column-stores. In *SIGMOD*, 2016.
[41] W. Xu, E. Lo, and P. Zhang. Difusion: Fast skip-scan with zero space overhead. In *ICDE*, 2018.

[42] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.
[43] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. In *ICDE*, 2012.

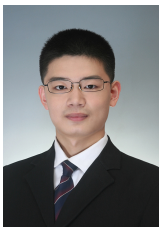


Zeke Wang received his Ph.D. degree from Zhejiang University, China in 2011. He is a Research Professor at Collaborative Innovation Center of Artificial Intelligence by MOE and Zhejiang Provincial GovernmentZhejiang University, China. His current research interests include heterogeneous computing (with a focus on FPGA), machine learning and database systems.

Xue Liu received the B.S. and Ph.D. degrees from Zhejiang University, Hangzhou, China, in 2006 and 2011, respectively. He is currently a lecturer with the School of Computer Science and Engineering, Northeastern University, Shenyang, China. His current research interests include FPGA-based system design and software defined radio.



Kai Zhang is a Pre-tenure Associate Professor at Fudan University. His research areas include database systems, networked systems, parallel and distributed computing.



Haihang Zhou received the BE and MS degree from Shanghai Jiao Tong University, PR China, in 2012 and 2015 respectively. He is pursuing a Ph.D degree with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include resource management, game theory and mobile cloud computing.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Associate Professor in School of Computing, National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.