Boosting Data Filtering on Columnar Encoding with SIMD

Hao Jiang
The University of Chicago
hajiang@cs.uchicago.edu

Aaron J. Elmore
The University of Chicago
aelmore@cs.uchicago.edu

ABSTRACT

In columnar databases, data is generally stored in an encoded format to save storage space and reduce I/O. Popular encoding schemes include dictionary encoding, delta encoding, run-length encoding, and bit-packed encoding. In many open-source columnar data formats, performing queries on encoded data requires the data to be first decoded to memory, which is time-consuming. In this paper, we design several novel SIMD-based algorithms to speed up query execution on encoded data. Our algorithms use SIMD to vectorize the execution and skip unnecessary decoding for higher efficiency, achieving a throughput of filtering up to 18 billion numbers per second with single thread. We build SBoost, a columnar data store utilizing these algorithms to speed up filtering on encoded data, thus improving query efficiency. SBoost is written in Java and invokes the SIMD algorithms using JNI, making it readily available for Java-based query platforms, which are dominant in open-source data analytic systems. SBoost demonstrates great potential in speeding up query efficiency in both disk-based analytic queries and in-memory queries by reducing query time by up to 90% compared to Apache Parquet.

ACM Reference Format:
Hao Jiang and Aaron J. Elmore. 2018. Boosting Data Filtering on Columnar Encoding with SIMD. In *DaMoN'18: 14th International Workshop on Data Management on New Hardware, June 11, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3211922.3211932

1 INTRODUCTION

Columnar databases such as C-Store [23] and MonetDB [10], are increasingly playing important roles in this big data era. Different from traditional RDBMS, which stores data in a row-by-row fashion, columnar databases adapt a column-oriented fashion to organize its data. In a columnar database, values from different columns are physically separated, and data in the same column is stored consecutively. Such a physical layout allows for faster scans on data columns, benefiting from sequential access and irrelevant columns to be skipped during query to avoid I/O overhead. These features make columnar databases the perfect choice for query and analysis on gigantic datasets.

Columnar stores allow efficient encoding techniques to be adopted. Abadi et al. show that in addition to space saving, executing queries on encoded data also exhibits great potential in improving query efficiency[1]. In practice, lightweight encoding algorithms, which trade compression ratio for much faster decompression operations, are preferred as they allow decoding to be performed on the fly without obvious impact to query performance. Widely used encoding schemes includes bit-packed encoding, dictionary encoding, delta encoding, and run-length encoding.

Many previous researches focus on using new hardware features, such as single-instruction-multiple-data (SIMD) instructions, to improve query performance on encoded data. Willhalm et al. [24] demonstrates a new algorithm using 128 bit SIMD instructions to decode 4 bit-packed integers in parallel. Polychroniou et al. [18] propose using SIMD to speed up selection scan, sort, and join operations. Variations of encoding schemes further explore the potential of SIMD processors. BitWeaving [15] and BP-128 [13] are variations of bit-packed encoding. SIMD-PFOR [13] is a variation of patched encoding. These variations all exhibit significant better performance comparing to corresponding scalar version and demonstrate that using SIMD to speed up encoding/decoding operations in database systems has great potential.

However, most of these algorithms work only on customized variation of encoding schemes that either need extra space in the storage format or requires data to be re-organized in a special order, making them space-inefficient and incompatible with standard encoding specifications. For example, one of the variation formats BitWeaving proposes, BWH, requires a separator bit between entries and entries residing within 64-bit lanes that can lead to a space waste of up to 30%. Another variation, BWV packs data tightly, yet requires data to be stored vertically instead of horizontally, e.g, adjacent bits in same entry are separated into adjacent words. In addition to wasting space, converting existing datasets that are already encoded with standard encodings to the new storage format is time-consuming and impractical considering the enormous amount of existing datasets.

To fill the gap, we propose several novel SIMD-based algorithms for fast filtering / decoding data stored in standard encoding formats, including bit-packed encoding, run-length encoding, and dictionary encoding. Our data filtering algorithms works directly on encoded data, efficiently skipping a decoding process, saving both CPU effort and memory space. Comparing to previous methods, our algorithms can process more numbers in parallel and achieves a throughput of filtering up to 18 billion numbers per second with a single thread.

We implement these algorithms in SBoost, a columnar data store based on Apache Parquet's storage format. SBoost is implemented in Java, and invokes SIMD algorithms through JNI to speed up data filtering on Parquet tables. SBoost works on widely used standard encoding schemes, making it readily available for existing data stores, and outperforms existing solutions by at least a order of magnitude. By improving query times for both on-disk and in-memory queries in Parquet, SBoost demonstrates great potential in speeding up query efficiency for Java-based query platforms.

DaMoN'18, June 11, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5853-8/18/06…$15.00
https://doi.org/10.1145/3211922.3211932

The contributions of this paper include

- **Fast Table Filtering on Bit-packed encoded data.** During a selection scan / filtering, predicates such as equality and range search will be applied on data to obtain a comparison result. Many previous systems require data to be either fully or partially decoded before the comparison can be performed. We propose a fast SIMD-based table scan algorithm on bit-packed data. The new vectorized algorithm allows executing predicates directly on encoded data to skip decoding process, and having more numbers processed in parallel to improve throughput, thus achieving ultra-fast data filtering.
- **Fast Table Filtering for Run-length and Dictionary encoded data.** Using query rewriter to convert queries on the encoded data to predicates on underlying bit-packed data and utilizing our fast bit-packed scan algorithm, we propose fast SIMD-based table filtering algorithms for both run-length and dictionary encoded data.
- **Fast Decoding and Table Filtering for Delta encoded data.** Decoding delta encoded data involves an iterative add operation through all data entries. We introduce a new vectorized algorithm for decoding delta encoded values, and further support efficient filtering on the decoded data.
- **Speeding up Java-based Query Platforms with SIMD + JNI.** We build SBoost to demonstrate that our algorithms are able to speed up both OLAP and in-memory queries for Apache Parquet. It also provides JNI interfaces and can be easily migrated to other Java-based query platforms. Our experiments shows this architecture has potential to improve query efficiency for other Java-based query platforms such as Spark, ORC, and CarbonData.

The remains of the paper is organized as follows. In Section 2 we introduce the supported encoding schemes, as well as previous work. In Section 3 we describe the design and implementation of the SBoost framework for table filtering and decoding. In Section 4 we describe in detail the algorithms proposed for each encoding scheme. Section 5 demonstrates the experimental results. Section 6 conclude our contribution and describe future research directions.

2 BACKGROUND AND RELATED WORK

In this section, we describe the encoding schemes involved in this paper, and introduce previous related research.

2.1 Encoding Schemes

Here we summarize encoding schemes in this paper.
Bit-Packed Encoding Bit-Packed Encoding stores a number using as few bits as possible. Given a list of non-negative numbers $[a_0, a_1, \dots, a_n]$, bit-packed encoding find a w satisfying $a_i < 2^w, \forall i \in [0, n]$, and represents each number losslessly using w bits. The bits are then concatenated in sequence as the encoding output.
Run-length Encoding Run-length Encoding encodes a consecutive run of repeating values as a pair *(val, run-length)*. The list $[a_0, a_0, a_1, a_2, a_2, a_2, a_3, a_3, a_3, a_3]$ will be encoded as $[a_0, 2, a_1, 1, a_2, 3, a_3, 4]$. In most prevalent columnar data stores, the result is then bit packed to further save space. The value fields and run-length fields may use different bit packing size.
Dictionary Encoding Dictionary Encoding uses a bijective mapping (a dictionary) to map input values of variable length to compact integer codes, and bit pack the result. The dictionary used in the encoding process is prefixed or attached to the encoded data. Dictionary allows conversion from data of arbitrary type to integer codes, further enabling them to be efficiently bit packed or run-length encoded.
Delta Encoding Delta Encoding stores the delta between consecutive numbers. Given a list of numbers $[a_0, a_1, a_2, \dots, a_n]$, delta encoding encode it as a list $[b_0 = a_0, b_1 = a_1 - a_0, b_2 = a_2 - a_1, \dots, b_n = a_n - a_{n-1}]$. The result may then bit-packed. As the delta between numbers are generally smaller than the numbers themselves, bit-packing the delta allows higher compression ratio than directly bit-packing the original data.

2.2 Related Work

Database Encoding and SIMD Database systems involves extremely intensive IO operations. Compression techniques greatly reduce the amount of data to be transferred at the cost of CPU occupancy upon decompression. Various studies [3, 11, 20] explore the impact of compression on database performance.

Columnar data stores save data from the same column in consecutive manner, allowing efficient application of encoding techniques mentioned in this paper. Encodings achieves high compression ratios with relative low CPU consumption. They also allows in-situ query execution without decoding the entire data block [1]. These advantages make them more favorable than generic compression algorithms, such as GZip and Snappy, in database systems.

As decoding processes generally involves independent simple operations on multiple data entries, SIMD seems like a perfect solution to the problem. Willhalm et al. [24] describe a SIMD-based algorithm for decoding and filtering tightly bit-packed data. While Willhalm's algorithm uses one 32-bit lane to filter each entry, and can process at most 16 entries in parallel using AVX-512, our algorithm fits as many entries as possible into a 64-bit lane, and can process up to 256 entries for entry size of 2, or 168 entries for entry size of 3 in parallel. In the experiments, our algorithm is able to achieve up to 12x throughput in filtering speed compare to Willhalm's algorithm and allowing us to filter up to 18 billion values per second.

Others focus on designing encoding variations that work well with SIMD. Stepanov et al. [22] introduce a SIMD version of varint-G8IU [5]. Lemire et al. propose SIMD-FastPFOR [13], a SIMD variation of PFOR [27] that pads both binary-packed data and exception arrays to be aligned with SIMD word boundaries. Li et al. demonstrate BWH and BWV, variations of bit-packed encoding that supports SIMD based fast filtering and early-pruning [15].

SIMD Acceleration in other Database Operations SIMD has many advantages comparing to other hardware acceleration alternatives. Most importantly, SIMD is built in CPU and has direct access to CPU databus and cache, avoiding data movement between different device memories. SIMD also has instruction level inter-operability with control flow codes, allowing fine-grain transition between parallel and scalar mode.

SIMD based algorithms have been proposed for almost every aspects of database execution. Zhou et al. [26] describe the general

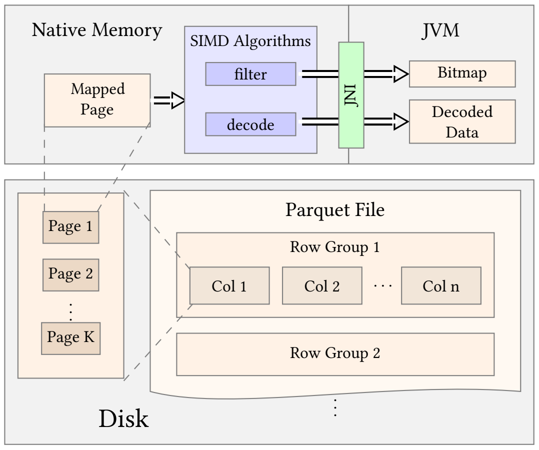


Figure 1: SBoost System Architecture

idea of using SIMD for various database operators including scan, aggregation, index scan and join. Chhugan et al. [4] use SIMD to implement a bitonic merge network for merge sort. Ross et al. [21] proposes to speed up hash join by optimizing Cuckoo hashtable [6] with SIMD. Jha et al. [12] experimentally explores the hardware oblivious and hardware conscious joins on Xeon Phi platform with SIMD optimization. Other applications including vectorized bloom filter [19] and bitmap counting [17].

3 SYSTEM DESIGN

We build SBoost, a columnar data store supporting SIMD-based fast table scan based on Apache Parquet[7], a prevalent open source columnar format. We design SBoost to be system independent, allowing it to be easily migrated to other columnar stores.

Figure 1 describes SBoost's system architecture. A Parquet file is comprised of multiple row groups, each consisting of multiple fixed size pages, which are binary data buffers storing encoded column data. When filtering or decoding data from a column, SBoost locates corresponding data pages from the Parquet file, maps each page to off-heap memory, and invokes the corresponding SIMD algorithms, which are implemented in C++, through JNI to process all data items in that page. Results are then passed back to the JVM for further processing. This design avoids data movement between the JVM and native memory, also minimize the number of JNI invocations, which has non-negligible cost.

SBoost defines two APIs, **filter** and **decode**, for each encoding scheme. **filter** executes a predicate on an encoded column, and outputs a bitmap indicating values satisfying the predicate. **decode** decodes encoded data to ready-for-output format.

For columns that appear only in a select but not in a project, SBoost applies **filter** directly on encoded data buffer to generate the bitmap, which can be further used to filter other columns. Most open-source systems decode data before they can be fed to a predicate, which incurs both unnecessary CPU and memory overhead. SBoost provides highly parallelized algorithms involving minimal decoding operations, greatly reducing both CPU and memory consumption.

For columns appears only in project but not in select, SBoost executes **decode** on them. SBoost designs novel algorithms utilizing SIMD parallelization to speed up decoding process.

For columns involved in both select and projection, SBoost first uses **filter** to generates bitmap on the column, and uses the bitmap result to efficiently perform data skipping, saving time for decoding operations on unmatched data.

3.1 Operator for Data Filtering

SBoost supports common predicates, including equal, not equal, greater than, less than, and their logical combinations in data filtering. We implement these predicates using two operators: `equal`, which tests whether the target is equal to a given value a, and `less`, which tests whether the target is less than a given upper-bound a. These operators take as input the encoded data and output a bitmap.

It is easy to see that all predicates and their combinations can be implemented using these two operators with simple logical operations. For example, less-equal$(x, a) = x \leq a$ can be obtained by or(less(x, a), equal(x, a)), and range$(x, a, b) = a \leq x < b$ can be obtained by xor(less(x, a), less(x, b)), with the presumption that $a \leq b$. When introducing our implementation of `filter`, we will focus on describing how we implement `equal` and `less` operators.

4 SIMD ALGORITHMS

In this section, we detail the SIMD algorithms we design for each encoding scheme to speed up predicate execution and decoding on encoded data.

In subsequent sections, we use uppercase letters to denote SIMD words and lowercase letters for scalars. We use subscripts to indicate elements in SIMD words. E.g., for a SIMD word A, we use $A_0, A_1, \ldots, A_n$ to denote the data entries in it, in small-endian fashion. Entry size varies and will be clarified when needed.

4.1 Data Filtering for Bit-Packed Encoded Integers

In this section, we introduce our algorithm using AVX-512 for `filter` on bit-packed encoded integers. It also serves as the foundation of subsequent algorithms.

Preprocessing The first step of our algorithm is loading encoded data in a 512-bit SIMD word, and align them to 64-bit lanes. We load 4 128-bit SIMD words separately, combining them as one 512-bit word, and use `_mm512_shuffle_epi8` to do 128-bit lane shuffling, sending bytes belonging to each entry into corresponding 64-bit lanes. We then use `_mm512_srlv_epi64` to shift data to be aligned to lane boundary.

The purpose of this operation is to get data ready for the arithmetic operation we perform in the next step. Intel's SIMD instruction set only provides arithmetic instructions within 64-bit lane. While previous methods, such as BitWeaving, handle the problem by aligning data to 64-bit lanes when storing data, we perform such alignment on the fly. This saves both storage space and data transformation cost. Our experiments show executing the alignment operation at runtime introducing negligible performance impact.

In addition, we also study an alternative of directly performing 512-bit arithmetic operations, eliminating the need of performing data alignment. This is detailed in Section 5.

Equal Operator Given a SIMD word X containing n entries, each consisting of e bits, and a scalar a, the `equal` operator checks how many entries in X are equal to a. Let M be the most significant bit (MSB) mask that has 1 at the MSB of every entry, and 0 everywhere else, e.g., $\forall i, M_i = 1 \ll (e-1)$, A a SIMD word having every entry equals to a, e.g. $A_i = a$. The algorithm computes

$$\begin{aligned} D &= X \oplus A \\ R &= D \mid ((D \,\&\, {\sim}M) + {\sim}M) \end{aligned} \tag{1}$$

and return R as a sparse bitmap containing the equality test result in the MSB of each entry.

$$X_i = a \iff (R_i)_{msb} = 0 \tag{2}$$

We demonstrate how this algorithm works with an example. Let X be a SIMD word containing two 3-bit entries $\{X_1 = 3, X_2 = 5\}$, and a be 3, we have $X = 101011, A = 011011$. The MSB mask $M = 100100$. Applying the computations above, we obtain $R = 101000$. The 6th bit (e.g., MSB of X_2) of R is 1, meaning that X_2 fails the equality test. The 3rd bits (e.g., MSB of X_1) of R is 0, meaning that X_1 passes the equality test.

The algorithm checks whether $x = a$ by examining if $d = x \oplus a = 0$. Let d_{rb} be the remaining bits in d excluding MSB, $d_{rb} = d \,\&\, {\sim}m$, $d \neq 0$ if and only if one of the following is true:

- $d_{msb} = 1$
- $d_{rb} \neq 0 \iff (d_{rb} + {\sim}m)$ generates a carry to MSB
  $\iff (d_{rb} + {\sim}m)_{msb} = 1$
  $\iff ((d \,\&\, {\sim}m) + {\sim}m)_{msb} = 1$

Let $r = d \mid ((d \,\&\, {\sim}m) + {\sim}m)$, we see

$$x = a \iff d = 0 \iff r_{msb} = 0$$

Less Operator The `less` operator takes a SIMD word X and a scalar a, determining whether for each entry $X_i \in X, X_i < a$. We construct M and A in the same way as described above, and compute

$$\begin{aligned} U &= (X \mid M) - (A \,\&\, {\sim}M) \\ R &= ({\sim}\, A \,\&\, (X \mid U)) \mid (X \,\&\, U) \end{aligned} \tag{3}$$

then return R as a sparse bitmap satisfying

$$X_i < a \iff (R_i)_{msb} = 0 \tag{4}$$

The algorithm checks whether $x < a$ by examining if one of the following cases happens

- $x_{msb} = 0$ and $a_{msb} = 1$
- $x_{msb} = a_{msb}$ and $x_{rb} - a_{rb}$ causes a carry

In the first case,

$$x_{msb} = 0 \text{ and } a_{msb} = 1 \iff (a \,\&\, {\sim}x)_{msb} = 1 \tag{5}$$

In the second case, let $u = (x \mid m) - (a \,\&\, {\sim}m)$

$$x_{msb} = a_{msb} \iff [{\sim}(x \oplus a)]_{msb} = 1 \tag{6}$$

$$\begin{aligned} & x_{rb} - a_{rb} \text{ generates a carry} \\ \iff & (x \,\&\, {\sim}m) - (a \,\&\, {\sim}m) \text{ generate a carry} \\ \iff & [(m + x \,\&\, {\sim}m) - (a \,\&\, {\sim}m)]_{msb} = 0 \\ \iff & [(x \mid m) - (a \,\&\, {\sim}m)]_{msb} = 0 \\ \iff & u_{msb} = 0 \end{aligned} \tag{7}$$

Combining the equations above we have

$$x < a \iff (\underbrace{(a \,\&\, {\sim}x)}_{\text{Equation (5)}} \mid (\underbrace{{\sim}(a \oplus x)}_{\text{Equation (6)}} \,\&\, {\sim} \underbrace{u}_{\text{Equation (7)}}))_{msb} = 1$$

Using boolean algebra to simplify the formula, we have

$$(a \,\&\, {\sim}x) \mid ({\sim}(a \oplus x) \,\&\, {\sim}u)) = {\sim}({\sim}a \,\&\, (x \mid c)) \mid (x \,\&\, c) = {\sim}r$$

This shows:

$$x < a \iff r_{msb} = 0$$

Our algorithm exhibits several advantages comparing to previous methods. SIMDScan [24] moves each data entry into a separate 32-bit lane and makes comparisons, allowing it to process at most 16 entries in parallel with AVX-512. We perform the comparison *in situ*, avoiding unnecessary data movement, and process up to 256 entries in parallel. BitWeaving [15] requires one bit to be preserved between data entries, and data be aligned to 64-bit lanes. We allows data to be tightly packed when stored, saving up to 30% storage space, and process up to 50% more data in parallel.

Dealing with cross-boundary entries For entries crossing SIMD word boundary, we use unaligned load instruction to load the next SIMD word including that entry. Previous research [24] suggests that unaligned load/store leads to negligible performance penalties on recent Intel CPUs, and our experiments also justify this conclusion.

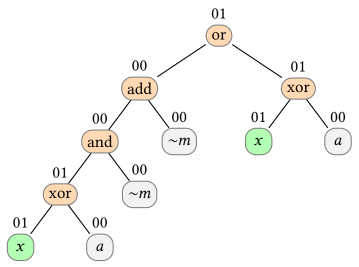
On platforms where unaligned load/store may lead to unacceptable performance penalties, we propose an alternative solution that simply extracts the involved bytes from SIMD register and use scalar comparison to execute predicates on them. The result is then written back to the corresponding location in the result data stream. We only need to write MSB for the given entry, which can be done with a bitwise operation involving one single byte in memory.

4.2 Data Filtering for Run-Length Encoded Integers

As is described in Section 2.1, run-length encoded data comprises of consecutive number pairs `(val, run-length)`. These pairs are often then tightly bit packed. While the approach described in this section targets bit-packed integers, the approach can be generalized to run-length encoding of other fixed-size attributes.

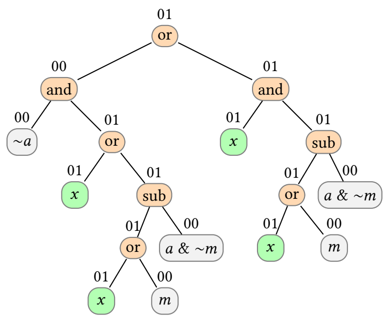
We utilize the bit-packed `filter` algorithm described in Section 4.1 to generate this run-length bitmap. The basic idea is to execute predicates on `val` fields, while leaving `run-length` fields unchanged. This generates a run-length encoded bitmap. For example, when executing predicate $x < 200$ on a run-length encoded data sequence $\{105, 2, 339, 4, 242, 1, 132, 8\}$, the output is $\{1, 2, 0, 4, 0, 1, 1, 8\}$. This kind of bitmap had been widely adopted in previous works [8, 9, 14, 25].

We show that by setting bits corresponding to run-length fields to 0 in all input parameters except for X used in Equation (1) and



$((x \oplus a)\ \&\ {\sim}m + {\sim}m)\ |\ (x \oplus a)$

Figure 2: Operation Tree for `equal` operator



$({\sim}a\ \&\ (x\ |\ (x\ |\ m - a\ \&\ {\sim}m)))\ |\ (x\ \&\ (x\ |\ m - a\ \&\ {\sim}m))$

Figure 3: Operation Tree for `less` operator

Equation (3), the run-length fields from X will be preserved during the computation of bit-packed `filter` algorithm. In Figure 2, we draw the operation tree for Equation (1). The numbers above each nodes shows how the bits from input change after each operation. We can see that if all parameters except for X(the gray blocks in figure) have their run-length fields set to 0, the run-length values from will be preserved. We perform the same check for `less` operator, as is shown in Figure 3 and get the same conclusion. This shows that by leaving the run-length fields as 0 in all parameters except for X used in Equation (1) and Equation (3), we can get a run-length bitmap generated by applying the bit-packed `filter` algorithm.

Some complex operators may need additional processing, though. For example, `range` operator can be obtained by $range(x, a, b) = less(x, a) \oplus less(x, b)$. Per our analysis above, $less(x, i)$ will preserve the run-length fields in input. Thus both operands of `xor` will have the same value in their run-length fields, and leads to 0 after the operation. To solve such problem, we simply rewrite $range(x, a, b) = less(x, a) \oplus (less(x, b)\ \&\ \underbrace{11\dots1}_{\text{value fields}}\ \underbrace{00\dots0}_{\text{run-length fields}}\ )$. That is, adding a mask that erases run-length fields from the right operand. This allows run-length fields data to be preserved during *range* opeartor. Similar technique can be applied to other operators.

4.3 Data Filtering for Dictionary Encoded Data

Our `filter` algorithm for dictionary encoded data again makes use of `filter` of bit-packed integers. The basic idea is that by rewriting predicates on dictionary encoded data to predicates on bit-packed data, we convert a predicate on dictionary encoded data into a predicate on bit-packed integer, which can be efficiently processed using the algorithm described before.

Formally, if we use dictionary d `:(key)->code` to encode list $[a_i]$ to $[d(a_i)]$, then for any predicate p on a_i, it is always possible to build a predicate p' on $d(a_i)$, satisfying $\forall i, p(a_i) = p'(d(a_i))$.

For `equal` operator $p(a_i) = \mathbb{I}[a_i = \alpha]$, we define $p'(b_i) = \mathbb{I}[d(a_i) = d(\alpha)]$ It is easy to verify p' satisfies the condition above as d is bijective. Non-equality cases can be processed in the same way.

For `less` operator to work we must have a order-preserving dictionary [2, 16]. An order-preserving dictionary d_o makes sure the codes follow the same order as their corresponding keys, e.g., $a_i > a_j \iff d_o(a_i) > d_o(a_j)$. Thus assume d_o is order-preserving, `less` predicate $p(a_i) = \mathbb{I}[a_i < floor]$ can be rewritten as $p'(d(a_i)) = \mathbb{I}[d(a_i) < d(floor)]$.

In addition, for string types, order-preserving dictionary support a prefix lookup operation `lookup(prefix)-> (mincode, maxcode)`, which allows us to effectively support prefix scan by rewriting it as a range scan.

Have shown that all predicates on a dictionary encoded data can be rewritten as predicates on bit-packed data, it is then straightforward to execute them using the bit-packed `filter` algorithm.

4.4 Fast Decoding and Filtering for Delta Encoded Data

In this section, we introduce our vectorized algorithm for decoding delta encoded integer and float data utilizing AVX2's `hadd` instruction.

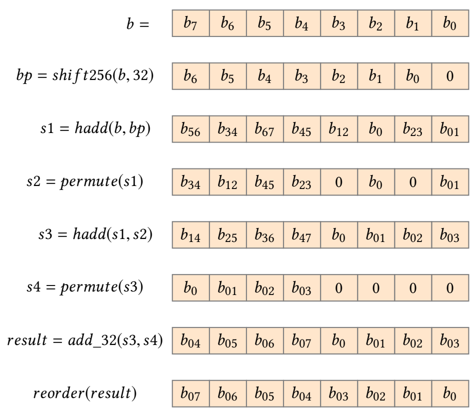
As is described in Section 2.1, delta encoding stores delta between consecutive numbers in a tightly bit-packed format. We first use the same algorithm as is described in the pre-processing step of Section 4.1 to unpack the bit-packed numbers into 16-bit or 32-bit lanes, depending on the size of original data.

With data unpacked as either 16-bit or 32-bit integers in SIMD words, the next step is to compute their cumulative sum in order to obtain original data. We introduce a `cumsum` function that computes the cumulative sum of each entry in a SIMD register. That is, given SIMD word $B = [B_0, B_1, \dots, B_n]$, $A = \texttt{cumsum}(B)$ computes $A = [A_0, A_1, \dots, A_n]$ where $A_i = \sum_{k=0}^{i} B_k$. The `cumsum` function for 256 bit SIMD word and 16/32 bit integer is demonstrated in Algorithm 1.

Figure 4 illustrates how the 32-bit algorithm works, where we use b_{ij} to denote $\sum_{k=i}^{j} b_k$. 16-bit `cumsum` works in a similar manner and we skip the description here for succinctness. Line 7 uses `permute` instruction to shift the input to left by 32 bits, shifting in 0. Line 8 uses `hadd` on the original input b and the shifted input bp to obtain sum of adjacent number pairs. Line 9 reorders the result using `permute` instruction, and line 10 uses `hadd` one more time to obtain partial sum of at most 4 consecutive numbers. Line 11 shifts the result of line 10 to the left by 128 bits, shifting in 0. Line 12 performs a 32-bit `add` to obtain cumulative sum for each index, and line 13 reorders the entry to correct sequence.

Algorithm 1 Vectorized Cumulative Sum with 256 bit SIMD and 16/32 bit Integer

```
const ZERO = _mm256_set1_epi64(0);
const IDX = _mm256_setr_epi32(8,0,1,2,3,4,5,6);
const IDX2 = _mm256_setr_epi32(0,8,2,8,1,4,3,6);
const IDX3 = _mm256_setr_epi32(8,8,8,8,0,1,2,3);
const INV = _mm256_setr_epi32(3, 2, 1, 0, 7, 6, 5, 4);
function CUMSUM32(b)
    bp = _mm256_permutex2var_epi32(b, IDX, ZERO);
    s1 = _mm256_hadd_epi32(b, bp);
    s2 = _mm256_permutex2var_epi32(s1, IDX2, ZERO);
    s3 = _mm256_hadd_epi32(s1, s2);
    s4 = _mm256_permute2x128_si256(s3, IDX3, ZERO);
    result = _mm256_add_epi32(s3, s4);
    return _mm256_permutevar8x32_epi32(result, INV);
end function
const SHIFT16 = _mm256_set1_epi64x(16);
const MASK16 = _mm256_set1_epi32(0xffff);
const INV16 = _mm256_setr_epi16(0xF0E, 0xD0C, 0xB0A,
    0x908, 0x706, 0x504, 0x302, 0x100, 0xF0E, 0xD0C, 0xB0A, 0x908,
    0x706, 0x504, 0x302, 0x100);
function CUMSUM16(b)
    bp = _mm256_bslli_epi128(current, 2);
    s1 = _mm256_hadd_epi16(current, bp);
    s2 = _mm256_sllv_epi64(s1, SHIFT16);
    s3 = _mm256_hadd_epi16(s1, s2);
    s4 = _mm256_and_si256(s3, MASK16);
    result = _mm256_hadd_epi16(s3, s4);
    return _mm256_shuffle_epi8(result, INV16);
end function
```



$b =$
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
$bp = shift256(b, 32)$
b_6 b_5 b_4 b_3 b_2 b_1 b_0 0
$s1 = hadd(b, bp)$
b_{56} b_{34} b_{67} b_{45} b_{12} b_0 b_{23} b_{01}
$s2 = permute(s1)$
b_{34} b_{12} b_{45} b_{23} 0 b_0 0 b_{01}
$s3 = hadd(s1, s2)$
b_{14} b_{25} b_{36} b_{47} b_0 b_{01} b_{02} b_{03}
$s4 = permute(s3)$
b_0 b_{01} b_{02} b_{03} 0 0 0 0
$result = add\_32(s3, s4)$
b_{04} b_{05} b_{06} b_{07} b_0 b_{01} b_{02} b_{03}
$reorder(result)$
b_{07} b_{06} b_{05} b_{04} b_{03} b_{02} b_{01} b_0

Figure 4: Use hadd to compute 32-bit Cumulative Sum

With the unpack and cumsum operations described above, it is now straight-forward to implement decode and filter for delta encoded data, which is shown in Algorithm 2 and Algorithm 3. We describe the 32 bit version here, and the 16 bit version can be implemented in a similar manner.

The variable latest in Algorithm 2 tracks the latest number we have computed so far, and is initialized to 0. Line 4 unpacks the bit-packed entry into SIMD words, and line 5 computes the cumulative sum on it. Line 6 adds latest to the cumulative result, obtaining the decoded value, and line 7 updates latest with the last entry .

Algorithm 2 decode for Delta Encoded 32 bit Integer

```
1:  function DECODE(stream)
2:      latest = 0;
3:      while stream.hasNext do
4:          word = UNPACK(stream.next);
5:          cumsum = CUMSUM(word);
6:          decoded = _mm256_add_epi32(cumsum, latest);
7:          latest = _mm256_extract_epi32(decoded, 7);
8:          OUTPUT(decoded);
9:      end while
10: end function
```

filter uses the output from decode, and utilize SIMD comparison operations to execute predicate on decoded entries. The result is a dense bitmap and can efficiently be used in future operations.

Algorithm 3 filter for Delta Encoded 32 bit Integer

```
function FILTER(stream, predicate)
    latest = 0;
    while stream.hasNext do
        decoded = DECODE(stream.next);
        if predicate.op == EQUAL then
            scanRes = _mm256_cmp_epi32_mask(decoded,
                predicate.val, _MM_CMPINT_EQ);
        else if predicate.op == LESS then
            scanRes = _mm256_cmp_epi32_mask(decoded,
                predicate.val, _MM_CMPINT_LT);
        end if
        OUTPUT(scanRes)
    end while
end function
```

To the best of our knowledge, no previous vectorized algorithm has been proposed for standard delta encoding. Lemire et al. [13] propose a vectorized variation of delta encoding for SIMD using SSE4 instructions. Instead of computing delta between adjacent numbers, Lemire's algorithm computes and stores delta between number pairs whose index are differ by 4 (as SSE4 registers can hold 4 32-bit integers). For example, given number $[a_0, a_1, \ldots, a_7]$, it stores $[a_0, a_1, a_2, a_3, a_4 - a_0, a_5 - a_1, a_6 - a_2, a_7 - a_3]$. When performing decoding, it loads every 4 entries into a SSE4 word and performs SIMD add operation to get original data. This variation speeds up decoding at the cost of storage space. In average, delta between these number pairs is four times of delta between adjacent numbers, and cost 2 more bits per entry for storage. When migrating this algorithm to larger SIMD word such as AVX-512, the extra space cost can be up to 4 bits per entry.

5 EXPERIMENTS

We use an experiment platform equipped with 2 Intel(R) Xeon(R) Silver 4116 CPUs@2.10GHz, and 190G memory. SIMD codes are compiled using GCC 5.4.0, with `-O3` flag. Software platforms used in the experiment include JDK Version 1.8.0_152, Scala Version 2.12.4, Apache Parquet version 1.9.0.

5.1 Microbenchmarks

In this section we evaluate SBoost's filter/decode algorithm performance on in-memory data, with single thread.

Data Filtering on Bit-Packed Integer

Figure 5 shows the experimental result of SBoost's `filter` operation on bit-packed encoded integers. In Figure 5a we compare SBoost with Willhalm's SIMDScan algorithm [13, 24], rewritten in AVX-512, and Apache's Parquet implementation, rewritten in C++. We can see that both algorithms outperform Parquet's highly optimized scalar algorithm by over one order of magnitude. Moreover, SBoost outperforms SIMDScan by another one order of magnitude on smaller entry sizes.

SBoost achieves higher efficiency on smaller entry size primarily due to higher parallelization. While SIMDScan uses one 32-bit lane for each bit-packed entry, SBoost can fit more than one entry in each 32-bit lane and compare them in parallel, thus achieves higher throughput for smaller entry sizes. SBoost is able to achieve up to 12x performance compared to SIMDScan (over 18 billion numbers per second). When entry size increases to over 22 bits one 64-bit lane can only accommodate at most 2 entries, which is the same as SIMDScan. Consequently, the throughput drops to the same level as SIMDScan.

We also compare SBoost to BitWeaving-H [15], rewritten in AVX-512. As mentioned before, BitWeaving-H does not use tightly bit-packed encoding. Instead, it uses an encoding scheme that trades storage space for efficient processing. Data in BitWeaving-H is stored in 64-bit lanes, with one bit reserved between each entry. We show that SBoost outperforms BitWeaving-H by 10~25% for small entry sizes, again due to higher parallelization. In Figure 5b, we show that SBoost outperforms BitWeaving-H by 10~25% for small entry sizes, again due to higher parallelization. In Figure 5c, we demonstrate the space usage to storing 1 billion numbers in tightly bit-packed format and in BitWeaving-H format. For smaller entries, SBoost is faster than BitWeaving-H. For larger entries, SBoost achieves similar performance as BitWeaving-H, but use much less space (at most 30% space saving).

We see that SBoost does not only outperform previous algorithms on tightly bit-packed integers, it also achieves same or better performance than BitWeaving-H. This shows using SBoost with tightly bit-packed integers is the best choice for both data filtering speed and storage efficiency.

Next, we propose a instruction modification that is able to further improve the efficiency of this algorithm, and may inspire future research. As is described in Section 4.1, our algorithm needs some extra pre-processing step to align data to 64-bit lanes, due to the limitation in arithmetic operation of Intel CPU. This step does not only costs extra CPU cycles, but also limits the number of entry we can process in parallel. For example, with entry size equals 13, we can fit 39 entries into 512-bit lanes, but only 32 entries in eight 64-bit lanes.

To study the impact of this limitation, we implement a software AVX-512 add/sub instruction and test its performance. We also evaluate the throughput if this 512-bit arithmetic instruction is supported and it takes the same cycles as 64-bit arithmetic operation, by counting the number of instructions executed. We notice that when using our software implementation of 512-bit arithmetic operations, throughput decreases to around 50~70% of SBoost due to the extra effort we employ to manually handle cross-lane carry bits. However, if this instruction is supported by hardware, we can gain another 15~20% performance improvement compared to SBoost, which is 20x to SIMDScan, and nearly 2x to BitWeaving-H. This shows that our algorithm has potential to further improve throughput and we explore the possibility of using dedicated hardware for a hardware implementation of this instruction to verify this in the future.

Finally, we conduct a performance evaluation on `filter` for dictionary encoded data. Previously we mention that for `filter` on dictionary-bit-packed encoded data, we use a order preserving dictionary and rewrite query to convert the operation into a `filter` on bit-packed encoded integers. We omit the result here for succinctness as it is identical to what is shown in Figure 5a. As a summary, SBoost is able to achieve nearly two orders of magnitude throughput comparing to Parquet, and can filter up to 18 billion bit-packed entries per second.

Data Filtering on Run-Length Encoded Integer

Next, we report our experimental result of SBoost's `filter` performance on run-length encoded integer. We vary both value field size and run-length field size, and report the result in Figure 6. Based on analysis on a real-world public dataset collection containing over 15,000 columns we have seen that over 99% of the datasets have an average run-length of less than 2^{10}, and thus focus our study on small entry sizes. It can be seen that while changing field size makes no difference to Parquet, SBoost again benefits much when dealing with small entries.

With a run-length field size of 5, SBoost achieves in average 20x and at most 40x throughput compared to Parquet, and can process in average 2 billion entries per second. When a larger run-length field size (15) is used, SBoost performance degrades due to less entries can be processed in parallel. Even though, it still achieves an average throughput of 1 billion entries per second.

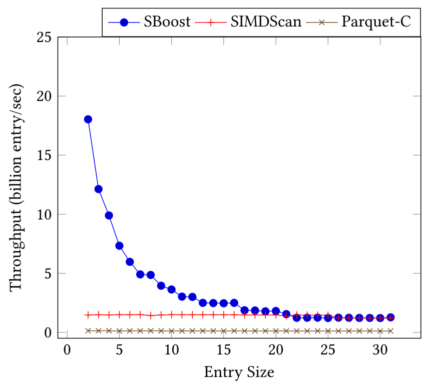
Even with extremely large run-length field size(26), which means only 8 to 16 entry can fit in a AVX-512 word, SBoost still manages to process 0.5 billion entries per second, which provides a lower bound of the algorithm's throughput.

Decoding Delta Encoded Integer

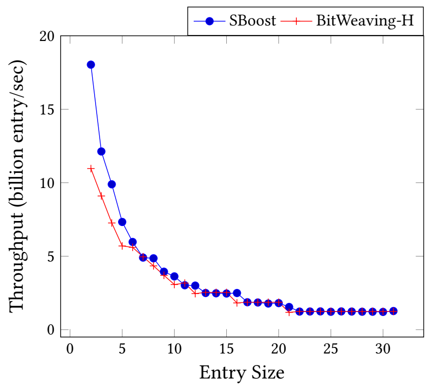
We report our experiment on SBoost's decode algorithm for Delta encoding. We compare our algorithm with the following methods.

- Scalar Decoding algorithm, which extract entries and computes the cumulative sum entry by entry.
- Lemire's vectorized variation of delta encoding [13], rewritten using AVX2. Parquet uses a similar encoding format as is used in this algorithm.

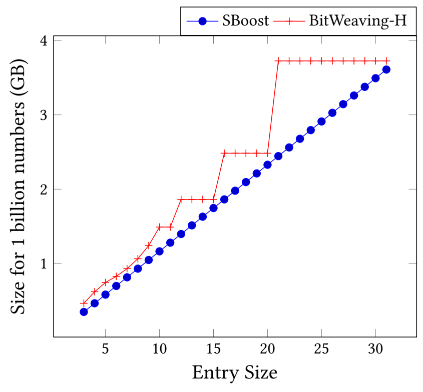
In Figure 7 we compare the throughput of these algorithms. Not surprisingly, Lemire's algorithm performs best as it only execute a single add instruction for each 8 numbers, and reaches a throughput of around 1.5 billion numbers per second. However, SBoost also manages to maintain a performance of 1 billion numbers per second, while they both outperform the scalar method by one order of



(a) filter Performance

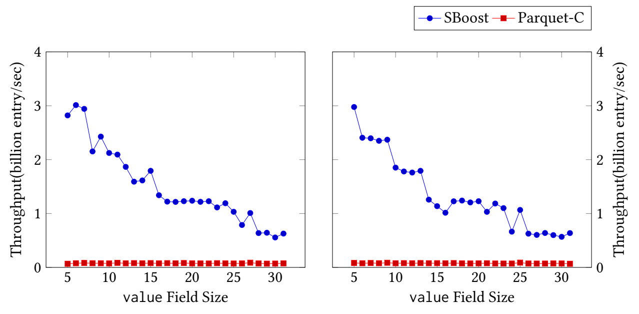


(b) Performance vs. BitWeaving-H

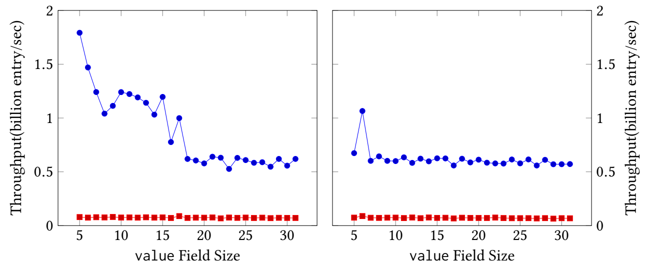


(c) Space Saving vs. BitWeaving-H

Figure 5: SBoost Performance on Bit-Packed Data



(a) Run-length Field size 5 **(b) Run-length Field size 7**



(c) Run-length Field size 15 **(d) Run-length Field size 26**

Figure 6: SBoost Performance on Run-Length Encoded Data

magnitude. This shows that if one need to process standard delta encoding or save storage space, SBoost is still a good choice.

Overall, we show that SBoost's algorithms achieves a similar or better performance comparing to previous state-of-art results, especially on small entry sizes, and improves space utilization by using standard (tight) encoding. SBoost also has obvious advantages comparing to widely used open-source implementations, and exhibits great potential in speeding up database queries.

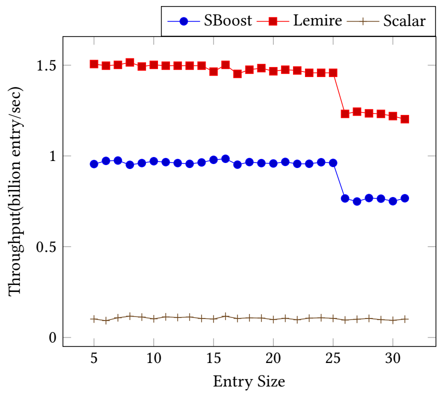


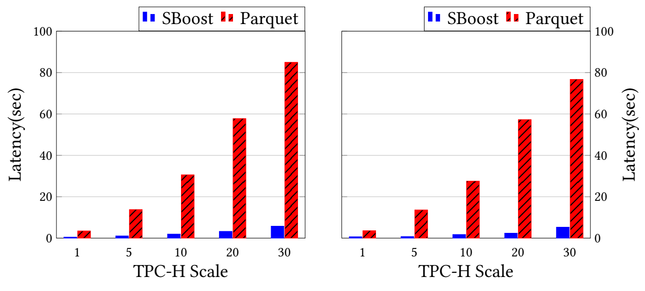
Figure 7: SBoost Performance on Delta Encoded Data

5.2 Boosting JVM-based Columnar Stores

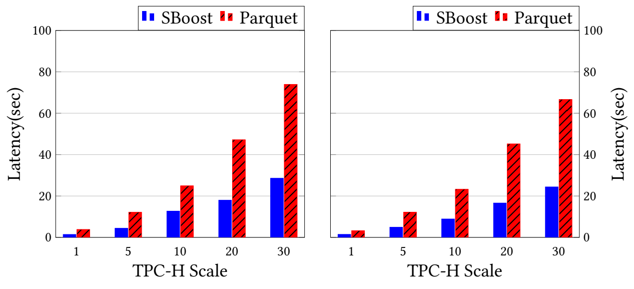
In this section, we demonstrate our experiment results of using SBoost to speed up data filtering/decoding in Apache's Java implementation of Parquet, and further improve query efficiency. We hand-craft a simple query engine, and execute TPC-H queries against both SBoost and Parquet. SBoost utilizes JNI to invoke SIMD algorithms for columns with supported data type and encoding, while retreating to Parquet's default implementation for columns that are not supported.

As SBoost aims at improving table filtering /decoding speed, we choose Q1 and Q6 from TPC-H queries as they only involves select/project operators. We use the TPC-H data generator to generate test datasets with scale varied from 1 to 30, and read files from both disk and memory (ramdisk) – simulating executions in both OLAP data stores and in-memory data stores.

We encode string columns `shipdate`, `line_status`, and double columns `extend_price`, `discount`, `tax` with dictionary-bit-packed encoding using a order-preserving dictionary, and integer column `quantity` with bit-packed encoding. We use SBoost `filter` to execute predicates on `shipdate`, and `quantity`, and use decode to extract `line_status`.



(a) TPC-H Q1 Disk **(b) TPC-H Q1 RAM**



(c) TPC-H Q6 Disk **(d) TPC-H Q6 RAM**

Figure 8: Accelerating Queries in Parquet

The experiment results are shown in Figure 8. For execution against files stored in both physical disk and in ram disk (simulating a in-memory database), we observe similar results. In Q1, the only predicate is on `shipdate` column, which can be executed efficiently with SBoost. In addition, `quantity` can benefit from SBoost's decode function. As a result, SBoost is one order of magnitude faster than Parquet's default implementation. For Q6, there are four columns involved in predicate execution, of which only two (`quantity` and `shipdate`) can be speed up using SBoost. The projected columns are all of double type thus do not benefit from SBoost. Even with these limitations, SBoost uses only 45% of Parquet's execution time.

In addition, we notice that the time difference between in-disk and in-memory execution, which is caused by I/O latency, is relatively small (5% ~ 10% of total time). As our experiment platform has large memory capacity, data files stored on hard disk can be efficiently read into page cache upon first access, making later operations equivalent to in-memory operations. This also shows that CPU computation, rather than disk IO, becomes the critical performance bottleneck for these queries, which further justify the effectiveness of our approach.

Overall, we believe this result clearly demonstrate SBoost's potential application in both disk-based OLAP and in-memory databases.

5.3 Scalability

In this section, we study the scalability of SBoost algorithms. It is straight forward to parallelize algorithms we introduce in this paper for bit-packed encoding, run-length encoding, and dictionary encoding. We simply split the input/output into multiple slices and process each slice with one thread.

For delta encoding, we use a two-pass method. In the first pass, we split input and output into slices as described above, and compute cumulative sum in each slice using the delta-decoding algorithm described before. In the second pass, for each slice, we add to it the sum of last elements from all slices before it. As in this phase, data in each slice has been decoded to 16-bit or 32-bit lanes, the add operation can be done efficiently using `_mm512_add_epi16` and `_mm512_add_epi32`.

Figure 9 shows the performance of bit-packed `filter` algorithm using multithreading. Run-length `filter` and dictionary `filter`, which are based on the same algorithm, exhibit similar patterns.

It can be noticed that multithreading does benefit the algorithm. Using 16 threads generally brings 4x~5x throughput comparing to single thread in all cases. However, we also notice that using more than 16 threads does not bring further benefit. For entry size of 3, adding more threads causes throughput to drop around 10%. For all other entry sizes, throughput stalls at some plateaus.

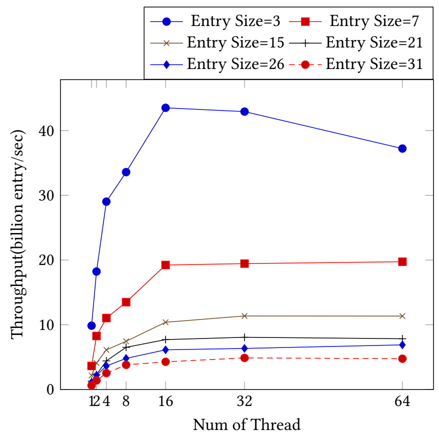


Figure 9: Scalability of Bit-packed `filter`

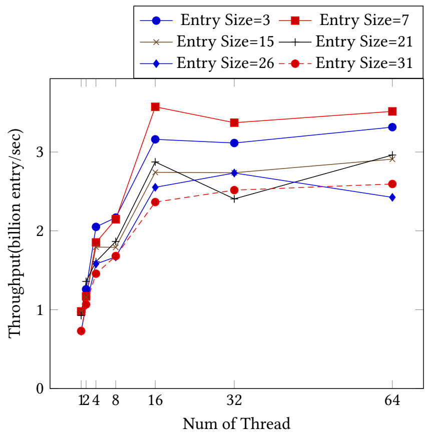


Figure 10: Scalability of Delta decode

The multi-threaded Delta decode algorithm, as is demonstrated in Figure 10, exhibits a similar pattern. Using more threads helps in the beginning, but no longer has obvious effect after using more than 16 threads.

This result is likely caused by hardware limitation. Our hardware platform is equipped with two CPUs, each with 12 cores. As the decoding process is highly CPU intensive, when the number of threads exceed available cores of each socket, system performance will not benefit from more threads.

Nevertheless, this demonstrates that our algorithms scale reasonably well within hardware limit and can make full utilization of available cores.

6 CONCLUSION

Hardware acceleration plays an important role in database research. Among all possible methods, SIMD has exhibited great potential, with advantages such as direct memory access and fused control flow. In this paper, we introduce novel SIMD algorithms for prevalent encoding schemes that support predicate execution directly on encoded data. Our algorithms work on standard encodings, requiring no additional storage space or special file format, yet providing lightening processing speed. Our data filter algorithm for bit-packed encoded integer and dictionary-bit-packed encoded integer / string can process over 18 billions numbers per second. Our algorithm for delta encoded integers and run-length encoded integers also achieves a throughput of over 1 billion numbers per second. We implement these algorithms and build a columnar data store SBoost based on Apache's Parquet. Our experimental results demonstrate that the new algorithms outperform their counterparts by at least one order of magnitude. It reduces query time by up to 90% for in-memory queries.

In the future, we plan to extend this work in several directions. We observe several limitations due to lacking SIMD instruction support, and are interested in developing accelerators of our algorithms to further improve efficiency. Furthermore, we would like to utilize our bit-packed data filtering algorithm for faster table joins and aggregations directly on encoded data.

ACKNOWLEDGEMENTS

This research is supported by the CERES Center for Unstoppable Computing and gifts from FutureWei.

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[2] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 283–296, New York, NY, USA, 2009. ACM.

[3] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 271–282, New York, NY, USA, 2001. ACM.

[4] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, aug 2008.

[5] Jeffrey Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 1–1, New York, NY, USA, 2009. ACM.

[6] Erlingsson, Ulfar and Manasse, Mark and McSherry, Frank. A cool and practical alternative to traditional hash tables. In *7th Workshop on Distributed Data and Structures (WDAS'06)*, Santa Clara, CA, January 2006.

[7] Apache Foundation. Apache Parquet. https://parquet.apache.org/, 2018.

[8] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *Proc. VLDB Endow.*, 3(1-2):1382–1393, sep 2010.

[9] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *2014 IEEE 30th International Conference on Data Engineering*, pages 484–495, March 2014.

[10] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[11] Balakrishna R. Iyer and David Wilhite. Data compression support in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 695–704, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[12] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *Proc. VLDB Endow.*, 8(6):642–653, feb 2015.

[13] D. Lemire and L. Boytsov. Decoding Billions of Integers Per Second Through Vectorization. *Softw. Pract. Exper.*, 45(1):1–29, jan 2015.

[14] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw. Pract. Exper.*, 46(11):1547–1569, nov 2016.

[15] Yinan Li and Jignesh M. Patel. Bitweaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 289–300, New York, NY, USA, 2013. ACM.

[16] Guido Moerkotte, David DeHaan, Norman May, Anisoara Nica, and Alexander Böhm. Exploiting ordered dictionaries to efficiently construct histograms with q-error guarantees in SAP HANA. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 361–372, 2014.

[17] Wojciech Mula, Nathan Kurz, and Daniel Lemire. Faster Population Counts using AVX2 Instructions. *CoRR*, abs/1611.07612, 2016.

[18] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1493–1508, New York, NY, USA, 2015. ACM.

[19] Orestis Polychroniou and Kenneth A. Ross. Vectorized Bloom Filters for Advanced SIMD Processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 6:1–6:6, New York, NY, USA, 2014. ACM.

[20] Gautam Ray, Jayant R. Haritsa, and S Seshadri. Database compression: A performance enhancement tool. 09 2004.

[21] K. A. Ross. Efficient hash probes on modern processors. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1297–1301, April 2007.

[22] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 317–326, New York, NY, USA, 2011. ACM.

[23] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[24] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.*, 2(1):385–394, aug 2009.

[25] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.

[26] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[27] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.