NULLS!
Revisiting Null Representation in Modern Columnar Formats

Xinyu Zeng
Tsinghua University
zeng-xy21@mails.tsinghua.edu.cn

Ruijun Meng
Tsinghua University
mrj21@mails.tsinghua.edu.cn

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Wes McKinney
Posit PBC
wes@posit.co

Huanchen Zhang*
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT

Nulls are common in real-world data sets, yet recent research on columnar formats and encodings rarely address Null representations. Popular file formats like Parquet and ORC follow the same design as C-Store from nearly 20 years ago that only stores non-Null values contiguously. But recent formats store both non-Null and Null values, with Nulls being set to a placeholder value. In this work, we analyze each approach's pros and cons under different data distributions, encoding schemes (with different best SIMD ISA), and implementations. We optimize the bottlenecks in the traditional approach using AVX512. We also propose a Null-filling strategy called SmartNull, which can determine the Null values best for compression ratio at encoding time. From our micro-benchmarks, we argue that the optimal Null compression depends on several factors: decoding speed, data distribution, and Null ratio. Our analysis shows that the `Compact` layout performs better when Null ratio is high and the `Placeholder` layout is better when the Null ratio is low or the data is serial-correlated.

ACM Reference Format:
Xinyu Zeng, Ruijun Meng, Andrew Pavlo, Wes McKinney, Huanchen Zhang. 2024. NULLS! Revisiting Null Representation in Modern Columnar Formats. In *20th International Workshop on Data Management on New Hardware (DaMoN '24), June 10, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3662010.3663452

1 INTRODUCTION

Codd first mentioned how to use Null values to represent missing data in a relational database in 1975 [17]. A subsequent paper in 1979 described the semantics of Null propagation through ternary logic for SQL's arithmetic and comparison operations [18]. Every major DBMS and data file format [27, 36] supports Nulls today and they are widely used in real-world applications; a recent survey showed that ~80% of SQL developers encounter Nulls in their databases [34].

Despite the prevalence of Nulls, there has not been a deep investigation into how to best handle them in a modern file format that is designed for analytical workloads processing columnar data.

ACM ISBN 979-8-4007-0667-7/24/06
https://doi.org/10.1145/3662010.3663452

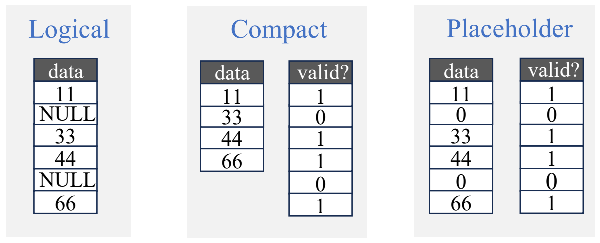


Figure 1: Null Representations – Examples of `Compact` and `Placeholder` representation schemes for a logical data set.

Today's most widely used columnar file formats (i.e., Apache Parquet [7], Apache ORC [6]) follow the same **`Compact`** layout as the seminal C-Store DBMS from the 2000s [13]. For each nullable attribute in a table, C-Store's scheme stores non-Null (fixed-width) values in densely packed contiguous columns. To handle Nulls, the scheme maintains a separate bitmap to record whether the value for an attribute at a given position is Null or not. Storing values in this manner enables better compression and improves query performance. However, because the `Compact` layout does not store Nulls, a tuple's logical position in a table may not match its physical position in the column, hampering random access ability.

An alternative approach is to store the Null values in place. That is, instead of pruning the Nulls out, this scheme uses a default value (e.g., zero, `INT_MIN`) as a placeholder to represent Null for a given tuple. The scheme still maintains a bitmap to indicate whether a position contains Null or not because the placeholder value may collide with a non-null value. Without further compression, this **`Placeholder`** layout always uses the same amount of storage space whether or not values are Null, but facilitates random access and vectorized execution. Recent systems and formats such as DB2 BLU [32], DuckDB [31], Apache Arrow[1] [4], and BtrBlocks [23] adopt this `Placeholder` layout. Figure 1 shows the difference between `Compact` and `Placeholder` layout.

Many DBMSs use a combination of Parquet and Arrow storage to represent data on disk and in-memory, respectively [5, 9, 10]. However, the different representation of Nulls between `Compact` (Parquet) and `Placeholder` (Arrow) introduces performance overhead. As shown in Figure 2, the time spent on format conversion from Parquet to Arrow, which represents a common deserialization

*Huanchen Zhang is also affiliated with Shanghai Qi Zhi Institute.
[1]The Arrow format does not specify Nulls to be any particular placeholder value, but implementations (C++ and Rust) fill it as zero to make the memory fully initialized.

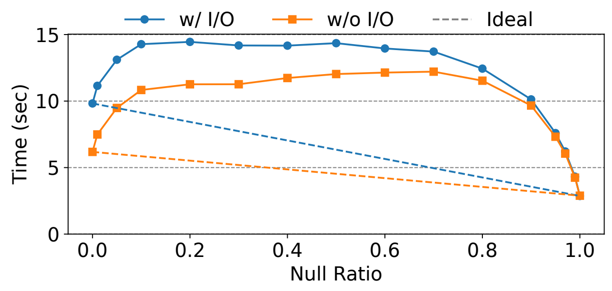
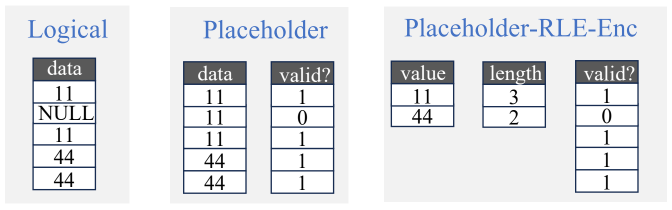
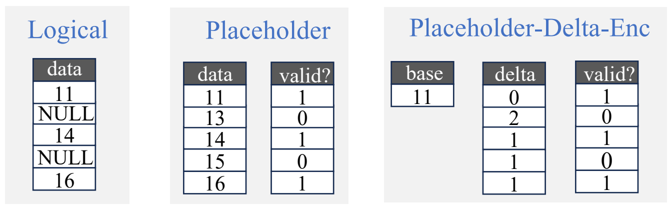


Figure 2: The Cost of Nulls – Converting a Parquet (`Compact`) file with 20 int32 columns, 64Mi rows into Arrow (`Placeholder`).



(a) Run-Length Encoding (RLE)



(b) Delta Encoding

Figure 3: `Placeholder` / Encoding Interactions – Examples of how the `Placeholder` layout interacts with different encoding schemes with better placeholder values. If the compression ratio is on par with `Compact`, a system can directly go for `Placeholder` and thus eliminate the expensive C⇢P Conversion.

operation in today's data science workflow almost doubles with 20% of Nulls without I/O time taking into the account. The dominating overhead in Figure 2 is that one has to scatter the non-Null values into their correct positions according to the Null bitmap — we refer to this operation as **C⇢P Conversion** (also known as `SpaceExpand` in Arrow [8]).

In this paper, we investigate the reasons behind the overhead shown in Figure 2, how to reduce such overhead, and how to improve the design of Null compression in future columnar formats. We first discuss the trade-offs between compression ratio and decoding speed of `Compact` vs. `Placeholder` layouts. We then survey the existing C⇢P Conversion implementations and propose an optimization based on the AVX512 `EXPAND` instruction to accelerate the conversion. To improve the compression ratio of the `Placeholder` layout, we propose a heuristic-based Null-filling strategy.

To evaluate these methods, we implement them in both a custom encoding framework and FastLanes [14], and compare them using our columnar-data microbenchmarks [36]. Our results show that when Null ratio is low, `Placeholder` is 2 – 3× faster than `Compact` while on-par with compression ratio. When Null ratio is high, `Compact` can be more than 2× faster because the cardinality reduction decreases decoding overhead. For serial-correlated data (e.g., timestamps), `Placeholder` never achieves the same compression ratio as `Compact`, but is up to 10× faster to decompress under AVX512. We also show that `Compact` is more than 2× slower for a vectorized execution primitive.

2 BACKGROUND AND RELATED WORK

In this section, we first discuss the background related to Null handling in columnar formats. We then discuss the pros and cons of the `Compact` and the `Placeholder` layouts.

2.1 Columnar Formats

Lightweight Compression: Unlike general-purpose block compression, lightweight compression (a.k.a. encoding) considers data types and other features to achieve a good compression ratio and fast decoding speed. Recent studies suggest that future formats should limit the use of block compression [14, 27, 36]. Representative lightweight encoding schemes include Run-Length Encoding (RLE), Dictionary Encoding, Bitpacking, Delta Encoding, and Frame of Reference (FOR). These schemes are widely used in columnar formats and are often applied in a cascading fashion. For example, the format first encodes a string column using Dictionary Encoding and then encodes the dictionary codes using RLE. After that, the format further applies Bitpacking to the RLE values and lengths.

Null Compression: Abadi et al. first explored the design space of Null compression in columnar formats [13]. All the designs only store non-Null values (i.e., in `Compact` layout), and use a secondary structure to indicate the positions of these non-Null values. Later works further improved the design of such secondary structure [19, 26, 29]. The DBMS can use either (1) a **selection vector (SV)** or ranges[2] if the column is sparse (i.e., Null ratio is high), (2) or a **bitmap (BM)** if the column is dense (i.e., Null ratio is low). RoaringBitmap can change its internal structure adaptively according to the Null ratio [26]. Nevertheless, current open standard columnar formats including DuckDB's memory and storage format, Arrow, and Velox's vector format all adopt the simplest bitmap representation without adaptivity.

BtrBlocks stores the data in the `Placeholder` layout [23]. Although not mentioned in the paper, their implementation takes Nulls into account during data sampling, and selects placeholder values for the Nulls to maximize the run-lengths when applying RLE and to maximize top-value counts for Frequency Encoding. Figure 3a gives an example. BtrBlocks, however, did not study the choices of placeholder values systematically and compare them with `Compact`. As shown in Figure 3b, if we choose the placeholder values wisely for Delta Encoding, the `Placeholder` layout could achieve a comparable or even better compression ratio than the `Compact` layout (Null-filling strategies are discussed in Section 4).

Vectorized Execution: Modern OLAP engines adopt the vectorized processing model, where query operators process batches of tuples (i.e., vectors) at a time [15, 16]. Such batching reduces iteration overhead (i.e., virtual function calls), keeps data in cache, and facilitates SIMD optimizations. Columnar formats such as Parquet

[2]e.g., using two pairs $(1, 3)$, $(5, 8)$ to indicate the range of non-Null positions.

and ORC are not designed for vectorized execution. Recent work urges that the leaf nodes of a query plan (i.e., file formats scanning and decoding) should also be redesigned to seize such opportunity of vectorization and SIMD [14].

In-memory vs. On-Disk Storage Formats: DBMSs use in-memory formats (e.g., Arrow, Velox) to represent intermediate data during query execution and to exchange data between different systems. Such formats emphasize the ability to support random access and vectorized compute primitives. Contrast this with disk-oriented storage formats (e.g., Parquet, ORC) that focus more on achieving high compression rates to reduce disk I/O via lightweight encoding schemes. Although the boundary between the two formats is not well defined (e.g., in-memory formats now include lightweight encodings [3]), for simplicity, we assume the in-memory format is in the plain `Placeholder` layout and the rest of the paper focuses on the design of on-disk storage formats. We will discuss the influence of Null representation on in-memory formats in Section 5.4.

2.2 Null Representations

We next compare the two Null representations (i.e., `Compact` and `Placeholder` layouts) in terms of the compression ratio and decoding speed they can achieve.

Compression Ratio (CR): Under the same encoding schemes, the `Compact` layout is likely to achieve a better CR compared to the `Placeholder` layout. The reason is that both `Compact` and `Placeholder` layouts need to store the Null bitmap, but the `Placeholder` layout has to keep the Null placeholders in the value vector (refer to Figure 3). However, we will show in Section 4 that with smart Null-filling strategies, the `Placeholder` layout could be as space-efficient as the `Compact` layout.

Decoding Speed: Decoding here refers to the process of converting the encoded `Compact`/`Placeholder` layout (e.g., the "`Placeholder`-RLE-Encoded" in Figure 3) to the plain `Placeholder` layout (e.g., the "`Placeholder`" in Figure 3), which is adopted by the in-memory formats for query processing (we will show why `Compact` is not efficient for in-memory query processing in Section 5.4). Despite that decoding the `Compact` layout requires scanning less data compared to decoding the `Placeholder` layout, it must perform the extra C⇢P Conversion to distribute the non-Null values to the final array. The computational overhead of the C⇢P Conversion is substantial, as shown in Figure 2. When the Null ratio is low, the C⇢P Conversion overhead often outweighs the benefit of decoding fewer values, making the overall decoding speed of the `Compact` layout slower than that of the `Placeholder` layout. As we can see, the implementation efficiency of the C⇢P Conversion plays an important role in analyzing the performance trade-offs between the `Compact` and `Placeholder` layouts. We, therefore, present and evaluate different C⇢P Conversion strategies in the next section.

3 COMPACT: C⇢P CONVERSION STRATEGIES

In this section, we first survey the existing C⇢P Conversion algorithms and then propose our own implementation based on AVX512. We use a bitmap (BM) to indicate the Nulls, following the design in existing file/in-memory formats.

```
void ConversionAVX512(const u32* in, u32* out,
  u64 num_values, i16* popcnt, const __mmask16* bm) {
  for (u64 i = 0; i < num_values; i += 512) {
    __m512i bm_512 = _mm512_load_si512(bm);
    _mm512_store_si512(popcnt,
            _mm512_popcnt_epi16(bm_512));
    for (u64 j = 0; j < 32; ++j) {
        __m512i expanded =
      _mm512_maskz_expand_epi32(
                bm[j], _mm512_loadu_si512(in));
      _mm512_store_si512(out, expanded);
      out += 16;
      in += popcnt[j];
    }
    bm += 32;
} }
```

Listing 1: C⇢P Conversion via AVX512 – Assume integers are 32-bit, and vector size is a multiple of 512 values. The remainder can be handled by other implementations.

3.1 Methods

Arrow BitRunReader [8]: To convert `Compact` inputs to a `Placeholder` representation, Apache Arrow uses a stateful BitRunReader. This component provides an iterator interface that returns a pair (pos, len), where pos is the start index of a non-null (set-bit) run and len is the run length. In this way, Arrow requires fewer `memcpy` when the Null ratio is low or high but suffers from excessive `memcpy` calls and branch mispredictions for a medium Null ratio where Nulls are common but non-consecutive (refer to Figure 2).

SIMD BM→SV [1, 12] + Scatter: This method first converts the Null BM into SV using SIMD instructions. The main idea is to pre-allocate a lookup table that maps a byte-long bitmap (with 256 possibilities) to the corresponding indexes of the set bits (i.e., SV). It then scatters the values in the `Compact` layout to their correct positions in the plain `Placeholder` layout according to the obtained SV. The conversion gets faster as the Null ratio increases because the SV used in the final scatter step becomes shorter. Appendix A's Figure 10 provides a high-level overview of this algorithm. We also describe an optimization in Appendix C.

Optimized Scalar BM→SV [2] + Scatter: This implementation uses CPU instructions to count the number of trailing bits (`TZCNT`) and reset lowest set bit (`BLSR`) to speed up the BM→SV conversion. Compared to the previous SIMD method, this implementation does not need to write the result SV to memory and then load it back for the scatter step. It also needs fewer instructions than the SIMD version when the Null ratio is high. We provide a simplified version of this algorithm in Appendix A's Listing 3.

AVX512 Expand: We propose our C⇢P Conversion implementation based on the AVX512 `EXPAND` instruction (illustrated in Figure 11 in Appendix A). For every 512 values in the `Compact` layout, we first perform a SIMD `POPCNT` on the Null BM to compute the number of positions the input pointer should advance after each of the next few (8 for 64-bit values and 16 for 32-bit values) `EXPAND` operations. We then invoke the `EXPAND` operation using the BM as

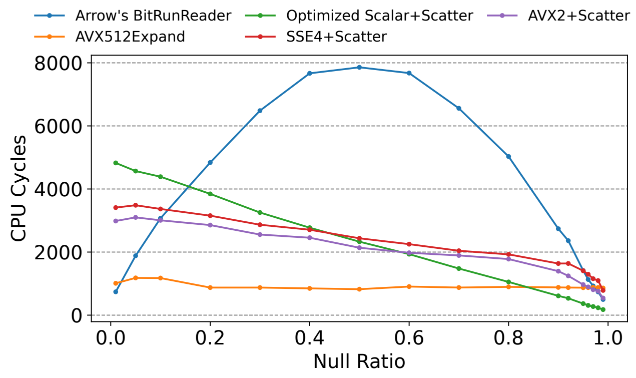


Figure 4: C⇢P Conversion Strategies Comparison – 2048 int32 values

the mask to complete the C⇢P Conversion. Listing 1 shows the simplified code.

3.2 Evaluation

We conduct a simple experiment to evaluate the performance trade-offs of the above C⇢P Conversion algorithms. The decoded result in the `Placeholder` layout contains 2048 32-bit integer values. We vary the Null ratio for the `Compact`-layout input and measure the number of CPU cycles consumed for each algorithm to perform the C⇢P Conversion. We provide a more detailed description of our hardware configuration in Section 5.

The results in Figure 4 shows the results. For a Null ratio smaller than 0.8, AVX512 `EXPAND` outperforms the SV-Scatter-based algorithms because it eliminates the expensive BM⟶SV conversion. In addition, unlike the SV-Scatter-based solutions, the total instruction count for our AVX512 `EXPAND` does not depend on the Null ratio. On the other hand, when the Null ratio is higher than 0.8, the Optimized Scalar approach exhibits advantages over AVX512 `EXPAND` because the number of values to scatter is small. Optimized Scalar also outperforms the SIMD versions (i.e., AVX2+Scatter and SSE4+Scatter) when the Null ratio is high because the BM ⟶ SV does conversion not fully utilize all the SIMD lanes. For example, in Figure 10, only the first three indexes (8 10 15) are effectively used in the conversion step in Figure 10 in Appendix A. Since `EXPAND` is only available in AVX512, in the rest of this paper, we adopt the AVX512 `EXPAND` implementation for the C⇢P Conversion if the encoding schemes also use AVX512 instructions. Otherwise, we choose the best SV-Scatter-based algorithm depending on the Null ratio according to Figure 4.

4 PLACEHOLDER: NULL-FILLING STRATEGIES

We next discuss how the Null-filling strategies affect the performance and space of the `Placeholder` layout. We present a dynamic Null-filling strategy, called SmartNull, that determines the best values for the Nulls at encoding time through a lightweight sampling.

4.1 Methods

Zero: The most widely used strategy is to fill the Null values with zeros (e.g., in Arrow). Other implementations such as DuckDB use the minimum value of the column (e.g., `INT_MIN`). Because most real-world integers have a small value range centered around zero [36], filling with zeros is more likely to result in a better CR under common lightweight encoding schemes.

Random: This strategy fills in the Nulls with random values. The benefit is that it does not need to zero out the buffer as in the Zero strategy. However, random values hurt the efficiency of lightweight encodings.

MostFreq: Null values are replaced by the most frequent non-Null value. This strategy achieves the best CR if the data is skewed, at the expense of a pre-scan to find out the most frequent value.

LastNonNull: Null values are replaced by the last-seen (i.e., nearest previous) non-Null value. This strategy works best with RLE, and it picks the Null value at encoding time without the need for a pre-scan.

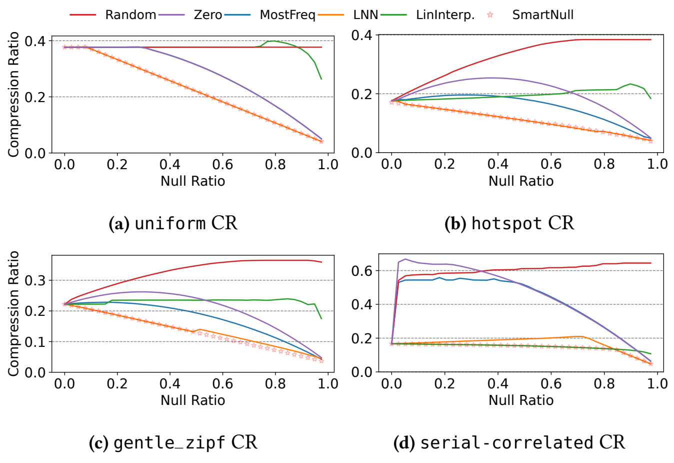
LinearInterpolation: For any two consecutive non-Null values where there are Nulls in-between, this strategy chooses the Null values to make the deltas between adjacent values equal. This strategy works best with a value sequence exhibiting serial correlations (e.g., a timestamp sequence that is best encoded using Delta or LeCo [28]). One can also apply other models here such as linear and polynomial regression.

SmartNull: Because of the trade-offs between different Null-filling strategies, a fixed strategy can hardly be optimal. Modern encoders often sample the data first to collect features (e.g., the number of distinct values, average run length, sortedness) to select the best encoding scheme [21, 23]. We propose to piggyback the above sampling phase to also select the optimal Null-filling strategy. Because each encoding scheme usually favors a particular Null-filling strategy, we only need to consider certain combinations of encoding schemes and Null-filling strategies (e.g., Bitpacking with Zero, RLE with LastNonNull, and Delta with LinearInterpolation). In this way, a DBMS can carry out the Null-filling procedure together with the encoding algorithm, thus reducing the overall compression time.

In our implementation of the SmartNull strategy, we consider Nulls during the encoder sampling phase. For example, when encountering a Null, we do not mark this run ended as if we applied the LastNonNull strategy when estimating the average run length. When estimating the delta bit-width for Delta encoding, we automatically fill in the Nulls using the LinearInterpolation strategy. When estimating the bit-width for Bitpacking, we set the Null-filling strategy to Zero instead. Thus, when a system chooses the encoding scheme that produces the best CR on the sampled data, the candidate encoding schemes are already combined with their best SmartNull strategies.

4.2 Evaluation

To evaluate Null-filling strategies, we built an encoding framework based on the ideas in BtrBlocks. This framework supports Bitpacking, RLE, and Delta encoding schemes. Bitpacking uses the SSE4-optimized implementation from FastPFOR [24]. Our RLE implementation is similar to the one in FastLanes [14] without transposing. Implementation for Delta Encoding is from [25]. We only



(a) uniform CR **(b)** hotspot CR

(c) gentle_zipf CR **(d)** serial-correlated CR

Figure 5: Null-filling Strategies

consider integer data because most strings and floats are mapped to integers via dictionary encoding [23, 35, 36]. The data has the following distributions: uniform, gentle_zipf, hotspot, and serial-correlated. The former three are derived from [36], representing common data distributions in the real world. serial-correlated is from the booksale data set in the SOSD benchmark [22].

We generate 8M values for each data distribution. The unit for encoding is 64K values. We first substitute the Null values according to different Null-filling strategies (SmartNull may not need this phase). We then feed the values to the encoder. The encoder first samples 16 runs of 64 values for each encoding unit and collects statistics on the samples. The encoding selection algorithm then chooses the best encoding scheme according to the estimated CR.

Figure 5 shows the results of the strategies under different data distributions and varying Null ratios. According to Figure 5, LastNonNull is the best across all distributions except serial-correlated. The reason is that, compared with MostFreq which tries to reduce global information redundancy, LastNonNull focuses more on local redundancy reduction, whose pattern can be better captured by encoding algorithms like RLE. Comparing MostFreq and Zero, their CRs are the same under uniform, but MostFreq is better when data is skewed (gentle_zipf and hotspot). For serial-correlated, none of the Null-filling strategies that try to reduce data duplication can work better than LinearInterpolation combined with Delta encoding, for Null ratio $< 80\%$. This indicates that serial-correlated needs special handling regarding Null-filling strategy. SmartNull is the optimal strategy across all the data distributions.

We also briefly describe the encodings SmartNull uses to help understand the results. SmartNull uses Bitpacking for Null ratio $<$ 10% and RLE otherwise under uniform. For hotspot and gentle_zipf, SmartNull chooses RLE for most Null ratios. In the serial-correlated distribution, SmartNull switches from Delta to RLE at the Null ratio of 85%.

5 EXPERIMENTAL EVALUATION

In this section, based on the results from the above analysis, we empirically compare Compact with Placeholder, first in the encoding framework we built in Section 4.2, then in FastLanes. Lastly, we explore the possibility of applying Compact to in-memory format.

All the experiments are conducted on an Intel Ice Lake 8375C with a max CPU frequency of 3.5GHz and a 512 GB DRAM. Per-core L1 cache size is 48 KB. We use g++ 10.2.1 (default in Debian 11) as the compiler. For each data point, we repeat 10 times and take the average.

5.1 Compact vs. Placeholder Comparison

We reuse the encoding framework and data settings in Section 4.2 to compare Compact and Placeholder. We use the Zero Null-filling strategy for Placeholder (w/o SmartNull). We limit the SIMD ISA level for both decoding and C⇢P Conversion up to SSE4 because our encoding framework uses the SSE4 version of FastPFOR. This restriction ensures both C⇢P Conversion and encodings are at the same SIMD level to have a fair comparison. For C⇢P Conversion, we use either Optimized Scalar or SSE4 + scatter based on the Null ratio according to Figure 4. We expand our evaluation to include AVX512 to Section 5.2.

The results in Figure 6 show that across all the data distributions, Placeholder (w/ SmartNull) provides the fastest overall speed without sacrificing too much space. The CR of SmartNull is close to Compact, especially when data is highly skewed (hotspot). Placeholder (w/o SmartNull), however, has even worse CR when Null Ratio is $\sim$ 40% in skewed data sets like gentle_zipf and hotspot. In serial-correlated, only a few Nulls hurt the CR for Placeholder (w/o SmartNull), as they become “outliers” in the data sequence.

Recall that we refer decoding as the process of converting the encoded Compact/Placeholder to the plain Placeholder layout. For decoding speed, however, Placeholder is faster than Compact when Null ratio is below ~0.8. The reason is two-fold: First, C⇢P Conversion itself is costly without the AVX512 EXPAND optimization. Second, C⇢P Conversion is not fused with the decoding of lightweight compression schemes — We do it the same as the current design in Parquet, so it involves extra LOAD and STORE instructions. When Null ratio is higher than ~0.8, the benefit of Compact needs to decode fewer values outweighs the overhead of C⇢P Conversion.

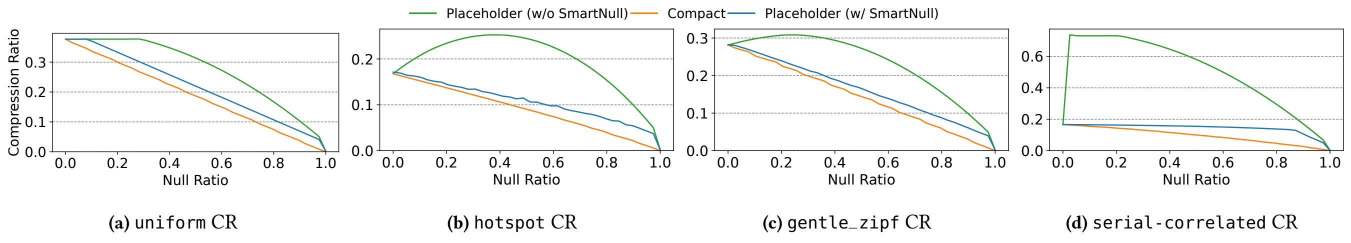
The main takeaway from this experiment is that future file formats should not fix the Null representation as either Compact or Placeholder w/ SmartNull. Instead, the format should dynamically select either Compact or Placeholder based on (1) the data's Null ratio and (2) whether the application prefers better space reduction or decompression speed.

5.2 Comparison under FastLanes with AVX512

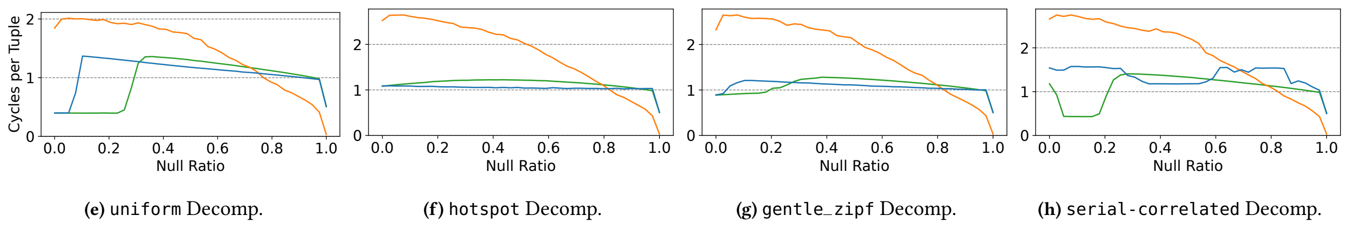
FastLanes (FLS) [14] achieves better data parallelism by transposing tuples to maximize useful work in SIMD operations. With the FLS layout, data can be efficiently decoded on different SIMD ISAs or via scalar code. Since FLS has the fastest decoding speed and is future-proof, in this section, we apply AVX512Expand on C⇢P Conversion, and compare Compact and Placeholder on FastLanes.

FLS assumes that queries rely on set-based relational algebra, and thus, changing the tuple order inside a table does not affect the query result. As we now discuss, this assumption imposes an additional challenge when a system combines FLS with Compact.

First, if Compact happens at the 1024-values vector level (FLS's compression unit), it reduces the number of entries inside the vector, but the transposed design of FLS always requires decoding 1024



(a) `uniform` CR **(b)** `hotspot` CR **(c)** `gentle_zipf` CR **(d)** `serial-correlated` CR



(e) `uniform` Decomp. **(f)** `hotspot` Decomp. **(g)** `gentle_zipf` Decomp. **(h)** `serial-correlated` Decomp.

Figure 6: Comparing `Placeholder` with `Compact` — Compression Ratio (CR) and Decompression Speed

values at a time. So `Compact` at the vector level does not reduce any memory or computation. Second, applying `Compact` at a larger granularity (e.g., reducing 10 vectors to 2 if the Null ratio is 80%) requires recovering the original order of each vector or converting the Null bitmap into a selection vector spanning across multiple vectors. After obtaining this selection vector, we can use a scatter operation to convert `Compact` to `Placeholder`, but it is more costly than register-level AVX512 `Expand` due to the larger access granularity. Thus, in the evaluation, we recover the tuple order in each vector via an inexpensive independent vectorized load. We then use Listing 1 to finish C⇢P Conversion. Appendix B's Figure 12 illustrates this process.

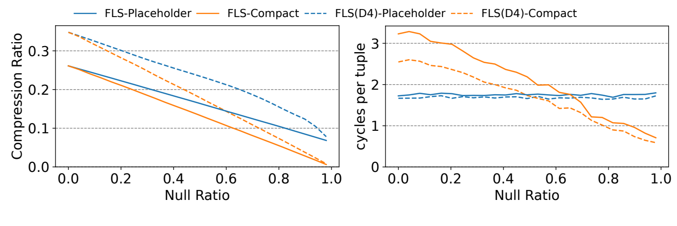
Due to the challenge of the transposed layout, we also adapt D4 [24] for FLS, denoted as FLS(D4). D4 is the delta encoding that does not change tuple order but has a worse CR than FLS. We only substitute the Delta encoding of FLS with D4 and leave RLE and Bitpacking the same as the ones in FLS.

We first measure the performance of FLS and FLS(D4) on `gentle_zipf`. The data is encoded in RLE and LastNonNull strategy. Then we measure performance on `serial-correlated` data, where data is encoded with Delta encoding and LinearInterpolation. C⇢P Conversion happens after the decoding of all vectors, meaning another scan over all the data. The result is from 1024×1K vectors.

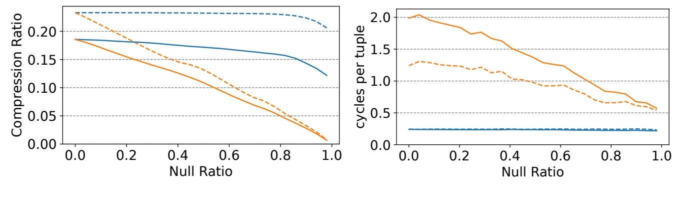
The results in Figure 7 show that under AVX512 the trade-off is similar to the one in Section 5.1. According to Figures 7a and 7b, when Null ratio is low, `Placeholder` is close to `Compact` in terms of CR while maintaining better decompression speed. When Null ratio is high, `Compact` excels in both CR and decompression speed. However, the threshold for `Compact` being faster is lower as C⇢P Conversion is significantly faster. We also notice that under `Compact`, FLS(D4) is faster than FLS, because of the extra transposition cost of FLS-`Compact`. Figures 7c and 7d validate the results in Section 5.1 where `Compact` achieves better CR but is slower.

5.3 Special Value

Another variant of the `Placeholder` layout is to use a special value (**`SpecialVal`**) different from the existing non-Null values to represent Null. This approach obviates the need to store the bitmap. Example systems adopting this scheme include SAP HANA [20],



(a) `gentle_zipf` CR **(b)** `gentle_zipf` Decomp.

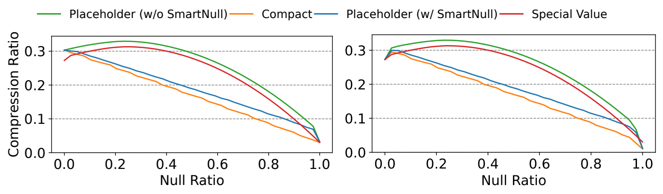


(c) `serial-correlated` CR **(d)** `serial-correlated` Decomp.

Figure 7: `Compact` vs. `Placeholder` (w/ SmartNull) on FastLanes (AVX512)

VoltDB, and MonetDB [11]. Although no open-source storage format uses this approach, we include it in our evaluation. There are several challenges with this approach. Foremost is that `SpecialVal` is often unusable for types with small value domains like `Boolean` or `Int8`; the format still needs to a separate bitmap if all the values in the domain are used. Second, this strategy avoids the storage cost of a separate Null bitmap, but the space of actual data is similar to the Zero-filling placeholder strategy (see Section 4). As such, it is unsuitable for a storage format that cares about compression.

In Figure 8a we rerun the `gentle_zipf` experiments in Figure 6, and also include the size of Null bitmap to the overall compression ratio. The result shows that `SpecialVal` is only better than `Compact` and `Placeholder` (w/ SmartNull) when there are almost no Nulls or all Nulls. It is also possible that the bitmap in `Compact` and `Placeholder` can be compressed using approaches like Roaring [26], which further makes CR of `SpecialVal` uncompetitive with `Compact` and `Placeholder`. Figure 8b shows that when applying Roaring bitmap to `Compact` and `Placeholder`, `SpecialVal` has little to no advantages in CR.



(a) `gentle_zipf` w/o BM Comp. **(b)** `gentle_zipf` w/ BM Comp.

Figure 8: `SpecialVal` Compression Ratio – This measurement is different from Figure 6c because it includes the size of Null bitmap.

```
for each idx in SV:
  if !IsNull(col_1,idx) &&
     !IsNull(col_2,idx) &&
     col_1[idx] < col_2[idx]:
       out_sv[cnt++] = idx
```

(a) Branch

```
for each idx in SV:
  out_sv[cnt] = idx
  cnt += !IsNull(col_1,idx) &
         !IsNull(col_2,idx) &
         col_1[idx] < col_2[idx]
```

(b) Branchless

Listing 2: Two versions of SVPartial with Nulls

5.4 In-Memory Formats

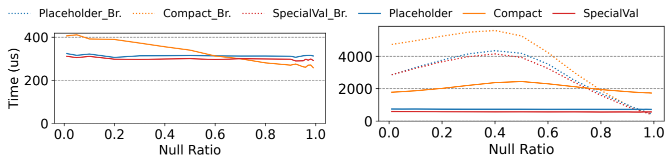
One might argue that using `Compact` in the in-memory format could avoid the overhead depicted in Figure 2 (and save space as well). Abadi et al. [13] executed an aggregation query with disjunctive predicates on `Compact`. All the predicates in their experiments were binary expressions with a column on the left-hand side and a constant on the right-hand side. Although they demonstrated that the performance scales with an increasing Null ratio, they did not consider predicates involving two (or more) columns.

In this section, we evaluate different strategies (including `SpecialVal`) for representing Nulls in an in-memory format within the context of vectorized execution. We follow the filter representation and compute strategy terminologies introduced in Ngom et al. [30]: SV is better for the "Partial" compute strategy that only evaluates the selected tuples, while BM is better for the "Full" strategy where all the tuples are evaluated to benefit SIMD processing. We, therefore, only consider the combinations of SVPartial and BMFull. For the Null indicator, we continue to use BM instead of SV.

We consider a predicate involving two *nullable* columns `col_1 < col_2`. Each column is associated with an SV/BM indicating the still-valid rows after evaluating the previous operators. Because both columns are nullable, the predicate is effectively `col_1 NOT NULL AND col_2 NOT NULL AND col_1 < col_2` during execution.

For (BMFull & `Placeholder`), we first evaluate the predicate on all the tuples using SIMD, and then `AND` the result with the two columns' Null BMs and their selection BMs. For (BMFull & `SpecialVal`), the evaluation of `col_1 != SpecialVal` is fused with that of `col_1 < col_2`. Because (BMFull & `Compact`) requires all the columns be aligned to facilitate SIMD operations, we apply the C⇢P Conversion in this case. We optimize the BMFull primitives as much as possible using AVX512.

We implemented SVPartial in both branching (Listing 2a) and branchless (Listing 2b) [33] ways. Evaluating `col_1 < col_2` first and short-circuit `NOT NULL` is incorrect because this predicate only produces results that are relevant to our evaluation when both values are non-null. Likewise, we do not consider varying selectivity because it does not affect Null representations. For `Compact`, we implemented the rank index in [19] to improve value lookups.



(a) BMFull **(b)** SVPartial

Figure 9: Varing Null representations on Vectorized Primitives – Evaluating $col_1 < col_2$ on different compute strategies, 1Mi rows. "_Br." denotes the branch version. For BMFull the result is the same across different filter selectivities. For SVPartial we only show selectivity=50% for both SV and the predicate.

Figure 9 shows the result. For BMFull, `Placeholder` is faster than `Compact` because `Compact` needs to do C⇢P Conversion first. If Null ratio is high, however, `Compact`'s memory saving can translate into performance gains. In the case of SVPartial, `Compact` is always slower than `Placeholder` regardless of using a branch or branchless implementation. This behavior stems from `Compact`'s non-random accessible format. Despite employing a rank index, its speed is still much slower than `Placeholder`. This result, where we re-evaluate a simple primitive in a modern vectorized setting, refutes the result in the seminal work in [13].

Interestingly, `SpecialVal` has slightly better performance than `Placeholder` with BMFull, despite that `SpecialVal` needs more instructions. The reason is that `Placeholder` has more memory access because of the extra bitmap and the temporary saving of comparison results before AND the Null bitmap. Under SVPartial, `SpecialVal` is faster than `Placeholder`, because by using special value the `IsNull()` operation in Listing 2 is cheaper than using the Null bitmap in `Placeholder`. The result shows that `SpecialVal` has potential performance benefit over `Placeholder` for vectorized primitives. However, it is less adopted than `Placeholder`, partly because of the issue of potential used-up value domain, and that mixing Null representation for query engine may lead to large number of kernel functions (with different combination of Null layout) and increased code size.

6 CONCLUSION

This work analyzes the impact of different Null representations on modern columnar formats. We first analyze the various factors that affect the overall compression ratio and decompression speed, between `Compact` and `Placeholder`. Then we analyze different implementation strategies of C⇢P Conversion for `Compact`, and Null-filling strategies for `Placeholder`. We also propose our implementation of C⇢P Conversion that utilizes AVX512, and a Null-filling strategy SmartNull that can be fused with the sampling phase during encoding. Through a set of experiments, we show that `Placeholder` with SmartNull offers better decompression speed with little space overhead when Null ratio is low, while `Compact` is better when Null ratio is high. We also show that `Compact` is no longer suitable for vectorized execution.

ACKNOWLEDGMENTS

This work was supported (in part) by Shanghai Qi Zhi Institute, VMware Research Grants for Databases, and Google DAPA Research Grants.

REFERENCES

[1] 2018. Iterating over set bits quickly (SIMD edition). https://lemire.me/blog/2018/03/08/iterating-over-set-bits-quickly-simd-edition/.

[2] 2019. Really fast bitset decoding for "average" densities. https://lemire.me/blog/2019/05/03/really-fast-bitset-decoding-for-average-densities/.

[3] 2024. Aligning Velox and Apache Arrow: Towards composable data management. https://engineering.fb.com/2024/02/20/developer-tools/velox-apache-arrow-15-composable-data-management/.

[4] 2024. Apache Arrow. https://arrow.apache.org/.

[5] 2024. Apache Arrow DataFusion. https://github.com/apache/arrow-datafusion/.

[6] 2024. Apache ORC. https://orc.apache.org/.

[7] 2024. Apache Parquet. https://parquet.apache.org/.

[8] 2024. Arrow C++ C->S Conversion. https://github.com/apache/arrow/blob/1eb46f763a73d313466fdc895eae1f35fac37945/cpp/src/arrow/util/spaced.h#L66-L94.

[9] 2024. Dremio. https://www.dremio.com/.

[10] 2024. Influx Data FDAP stack. https://www.influxdata.com/glossary/fdap-stack/.

[11] 2024. MonetDB Data Compression Doc. https://www.monetdb.org/documentation-Dec2023/admin-guide/system-resources/data-compression/.

[12] 2024. Velox's SIMDized BM to SV. https://github.com/facebookincubator/velox/blob/02ca9b0b4f554868b533d2f6526a480ea1e7d035/velox/common/base/SimdUtil-inl.h#L179.

[13] Daniel J Abadi et al. 2007. Column Stores for Wide and Sparse Data.. In *CIDR*, Vol. 2007. 292–297.

[14] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding> 100 Billion Integers per Second with Scalar Code. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2132–2144.

[15] PA Boncz and M Zukowski. 2012. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27.

[16] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[17] E. F. Codd. 1975. Understanding Relations (Installment #6). *FDT Bull. ACM SIGFIDET SIGMOD* 7, 1 (1975), 1–4.

[18] E. F. Codd. 1979. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.* 4, 4 (dec 1979), 397–434. https://doi.org/10.1145/320107.320109

[19] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *Proc. VLDB Endow.* 14, 11 (2021), 2491–2504. https://doi.org/10.14778/3476249.3476297

[20] Gerhard Hill and Andrew Ross. 2009. Reducing outer joins. *VLDB J.* 18, 3 (2009), 599–610. https://doi.org/10.1007/S00778-008-0110-5

[21] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 843–856. https://doi.org/10.1145/3448016.3457283

[22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *CoRR* abs/1911.13014 (2019). arXiv:1911.13014 http://arxiv.org/abs/1911.13014

[23] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[24] D. Lemire and L. Boytsov. 2013. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (May 2013), 1–29. https://doi.org/10.1002/spe.2203

[25] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2014. SIMD Compression and the Intersection of Sorted Integers. *CoRR* abs/1401.6399 (2014). arXiv:1401.6399 http://arxiv.org/abs/1401.6399

[26] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with Roaring. *Software: Practice and Experience* 46, 11 (April 2016), 1547–1569. https://doi.org/10.1002/spe.2402

[27] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.

[28] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1, Article 65 (mar 2024), 28 pages. https://doi.org/10.1145/3639320

[29] Dimitar Mišev, Mikhail Rodionychev, and Peter Baumann. 2023. Performance of Null Handling in Array Databases. In *2023 IEEE International Conference on Big Data (BigData)*. IEEE, 247–254.

[30] Amadou Ngom, Prashanth Menon, Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C Mowry, and Andrew Pavlo. 2021. Filter Representation in Vectorized Query Execution. In *Proceedings of the 17th International Workshop on Data Management on New Hardware*. 1–7.

[31] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1981–1984. https://doi.org/10.1145/3299869.3320212

[32] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[33] Kenneth A. Ross. 2004. Selection conditions in main memory. *ACM Trans. Database Syst.* 29, 1 (mar 2004), 132–161. https://doi.org/10.1145/974750.974755

[34] Etienne Toussaint, Paolo Guagliardo, Leonid Libkin, and Juan Sequeda. 2022. Troubles with Nulls, Views from the Users. *Proc. VLDB Endow.* 15, 11 (2022), 2613–2625. https://doi.org/10.14778/3551793.3551818

[35] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, Alexander Böhm and Tilmann Rabl (Eds.). ACM, 1:1–1:6. https://doi.org/10.1145/3209950.3209952

[36] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment* 17, 2 (2023), 148–161.

APPENDIX

A C⇢P CONVERSION STRATEGIES ILLUSTRATIONS

This section gives a visualization of SIMD BM⟶SV+Scatter in Figure 10, the simplified optimized scalar version of C⇢P Conversion in Listing 3, and visualization of AVX512 `EXPAND` in Figure 11.

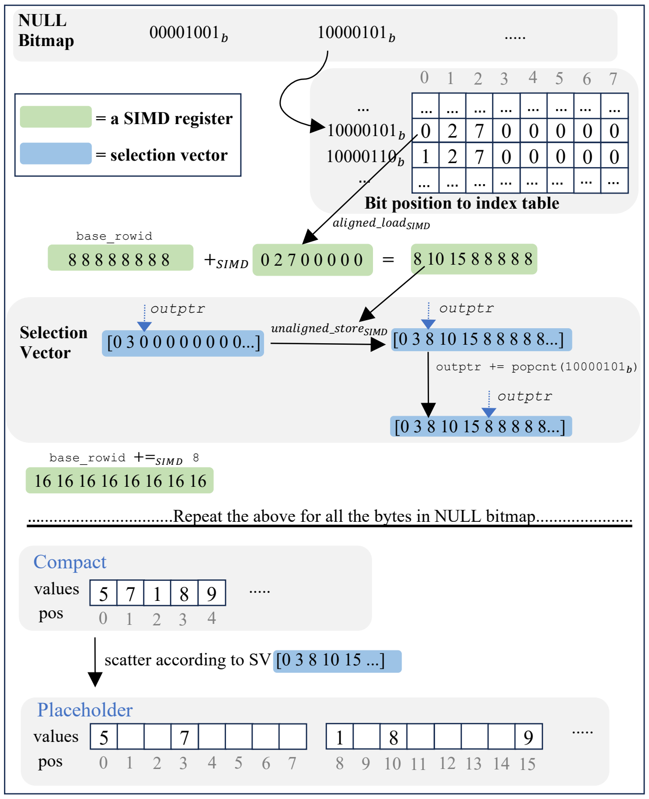


Figure 10: SIMD BM⟶SV+Scatter — Bit position to index table is calculated at compile time and statically allocated. The illustration is for AVX2 and 32-bit integers.

```
void ConversionScalar(const u32* in, u32* out,
                      u64 num_values, const u64* bm) {
  u32 row = 0, idx_in = 0;
  u32 num_words = RoundUp(num_values, 64) / 64;
  for (auto i = 0; i < num_words; ++i) {
    u64 word = bm[i];
    while (word) {
        out[__builtin_ctzll(word) + row] = in[idx_in++];
        word = word & (word - 1); // BLSR
    }
    row += 64;
  }
}
```

Listing 3: C⇢P Conversion via optimized scalar code – Different from other SIMD BM⟶SV implementations, this one requires fewer instructions when Null ratio is high, and eliminates the intermediate `LOAD` and `STORE` instructions of SV values.

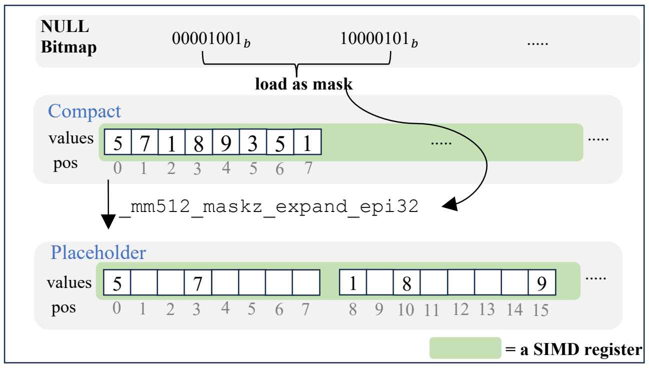


Figure 11: Illustration of AVX512 `EXPAND` — The C⇢P Conversion happens between two vector registers, and can directly operate on bitmap, without the need of BM⟶SV. We omit the `POPCNT` procedure and optimization in Listing 1 here.

B ILLUSTRATIONS OF FASTLANES WITH COMPACT LAYOUT

Here we give the illustration of FastLanes combined with `Compact` in Figure 12.

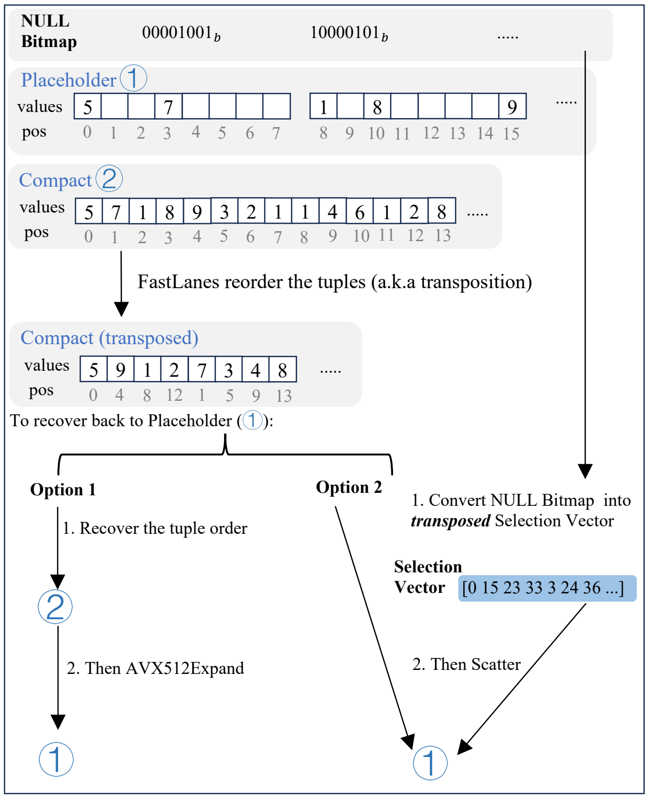

NULL Bitmap
00001001_b
10000101_b
Placeholder 1
Compact 2
values
pos
FastLanes reorder the tuples (a.k.a transposition)
Compact (transposed)
To recover back to Placeholder (1):
Option 1
Option 2
1. Recover the tuple order
2. Then AVX512Expand
1. Convert NULL Bitmap into transposed Selection Vector
Selection Vector [0 15 23 33 3 24 36 ...]
2. Then Scatter

Figure 12: Illustration of FastLanes combined with `Compact` — Option 1 needs extra untransposition of tuple order, but the cost is low as it happens in the L1 cache. The SV values in Option 2 may span across multiple vectors (out of L1 cache size) depending on the Null ratio.

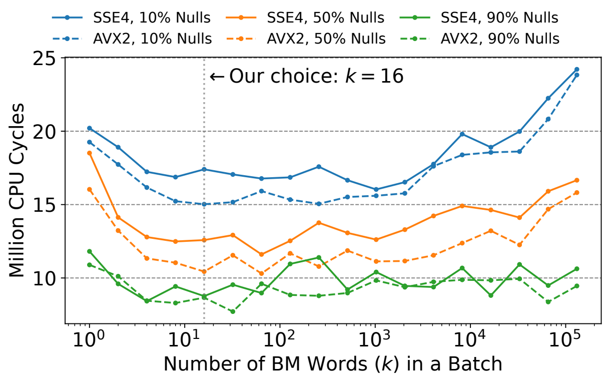


Figure 13: SIMD BM⟶SV+Scatter Varying Batch Size – 8M int32 values

C OPTIMIZATION OF C⇢P CONVERSION

This section describes our optimization when implementing the SIMD BM⟶SV+Scatter algorithm in Figure 10.

For data size that exceeds the cache size (e.g. 8M values in Sections 4.2 and 5.1), converting all the BM to SV first and then scatter introduces a large memory access cost of storing and loading SV values. To address this problem, we fuse these two steps to make the intermediate SV values fit in cache. Different from the scalar version (Listing 3), such fusion in the SIMD version can only be done in batches: for each batch we convert k words of BMs (corresponding to $64k$ values in the `Placeholder` format) into SV, then loop over the SV to scatter the values into the `Placeholder` layout.

To empirically evaluate the best k for different Null ratio, we sweep k on different Null ratio in both the SSE4 and AVX2 implementations. As shown in Figure 13, when the Null ratio is 10%, the performance decreases as k becomes large. This is because there are more non-Null values, resulting in more memory access overhead. For medium to high Null ratio, the performance drop is not that obvious when k is large, but is evident when k decreases from 10 to 1. This is caused by the branch misprediction overhead in the short scatter loop. In our implementation of SIMD BM⟶SV+Scatter, we choose $k = 16$, i.e., processing 1024 values in a batch, which is near optimal for all Null ratios.