BtrBlocks: Efficient Columnar Compression for Data Lakes

MAXIMILIAN KUSCHEWSKI, Technische Universität München, Germany
DAVID SAUERWEIN, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
ADNAN ALHOMSSI, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
VIKTOR LEIS, Technische Universität München, Germany

Analytics is moving to the cloud and data is moving into data lakes. These reside on object storage services like S3 and enable seamless data sharing and system interoperability. To support this, many systems build on open storage formats like Apache Parquet. However, these formats are not optimized for remotely-accessed data lakes and today's high-throughput networks. Inefficient decompression makes scans CPU-bound and thus increases query time and cost. With this work we present BtrBlocks, an open columnar storage format designed for data lakes. BtrBlocks uses a set of lightweight encoding schemes, achieving fast and efficient decompression and high compression ratios.

CCS Concepts: • **Information systems → Column based storage**; **Data compression**.

Additional Key Words and Phrases: data lake, query processing, compression, columnar storage

ACM Reference Format:
Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2, Article 118 (June 2023), 26 pages. https://doi.org/10.1145/3589263

1 INTRODUCTION

Data warehousing is moving to the cloud. Many organizations collect and analyze ever larger datasets, and, increasingly, these are stored in public clouds such as Amazon AWS, Microsoft Azure and Google Cloud. To analyze these datasets, customers use cloud-native data warehousing systems such as Snowflake [29], Databricks [25], Amazon Redshift [34], Microsoft Azure Synapse Analytics [23] or Google BigQuery [47, 48]. Another trend in cloud data warehousing is the disaggregation of storage and compute, where the data is stored on distributed cloud object stores such as S3, and where compute power can be spawned elastically on demand. This architecture was pioneered by BigQuery and Snowflake and even systems that initially started with a horizontally partitioned, shared-nothing design like Redshift are transitioning to disaggregated storage [24].

Data warehouses can become proprietary data traps. Cloud-native data warehousing systems are optimized for analytical queries through vectorized processing [27] or compilation [50], and all systems rely on compressed columnar storage [21], which has become a proven and mature technology. By default, most systems use proprietary storage formats. The big downside of proprietary formats is that they effectively trap the data in one system (or one vendor's ecosystem). Non-SQL

Authors' addresses: Maximilian Kuschewski, maximilian.kuschewski@tum.de, Technische Universität München, Arcisstraße 21, München, Germany, 80333; David Sauerwein, david.sauerwein@fau.de, Friedrich-Alexander-Universität Erlangen-Nürnberg, Schloßplatz 4, Erlangen, Germany, 91054; Adnan Alhomssi, adnan.alhomssi@fau.de, Friedrich-Alexander-Universität Erlangen-Nürnberg, Schloßplatz 4, Erlangen, Germany, 91054; Viktor Leis, leis@in.tum.de, Technische Universität München, Arcisstraße 21, München, Germany, 80333.

2836-6573/2023/6-ART118 $15.00
https://doi.org/10.1145/3589263

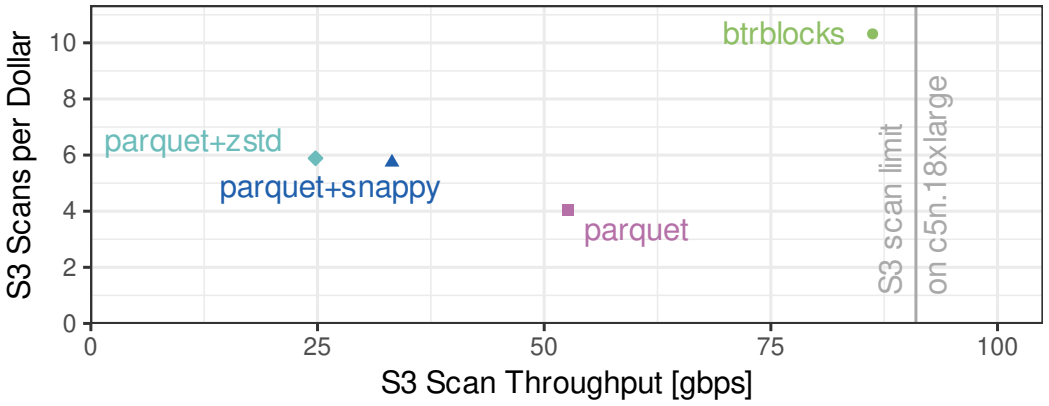


Fig. 1. S3 scan cost and throughput (c5n.18xlarge) on the 5 largest Public BI Benchmark datasets

analytics systems for machine learning or business intelligence often have to first extract the data from the data warehouse, which is not only cumbersome but also inefficient and expensive for large datasets. Often this leads to several unnecessary data copies all residing in the same object store – multiplying storage cost and making data changes difficult.

Data lakes and open storage formats. Data lakes enable interoperability across different analytics applications, including SQL-based data warehousing and complex analytics [60]. They do this by storing data on cloud object stores such as S3, and by relying on open storage formats such as Parquet or ORC that can be accessed by any analytics system. Given that the idea of data lakes is not new, one may wonder why proprietary solutions are still more common than open data lakes. We believe that this is due to two reasons. First, networks used to be slow, making data lake access from object stores relatively slow. Second, compared to their proprietary cousins, Parquet and ORC are neither efficient in terms of scan performance nor compact, which is why they are often combined with general-purpose compression schemes like Snappy [11] or Zstd [12]. While the network bottleneck has been solved with the arrival of cheap 100 Gbit networking instances (e.g., c5n or c6gn in AWS), in this paper we attack the second problem.

BtrBlocks. In this paper, we propose BtrBlocks [ˈbɛtəɹblɒks], an open-source columnar storage format for data lakes. BtrBlocks is designed to minimize overall workload cost through low storage cost *and* fast decompression. To achieve good compression on real-world data, we combine seven existing and one new encoding scheme, all of which offer fast decompression performance and can be used in a cascade (i.e., RLE then Bit-packing). BtrBlocks also includes an algorithm for determining which encoding to use for a particular block of data. Figure 1 compares its scan speed and cost with Parquet, the most common open data lake format. With real-world data from the five largest datasets in the Public BI Benchmark, scans using BtrBlocks are 2.2× faster and 1.8× cheaper due to its superior decompression performance. This makes BtrBlocks highly attractive as an in situ data format for data lakes.

Related Work and Contributions. Much of the existing research on compression focuses on specific encodings for integers [30, 31, 42, 61], while work on compressing strings [26, 39] and floating-point numbers [46] is more sparse. Furthermore, there is a surprising lack of end-to-end designs, i.e., a set of complementary encoding schemes and an algorithm that decides between them. This work consists of the following contributions: (1) A complete compression design for

relational data based on an empirically-selected set of compression schemes that are introduced in Section 2. (2) A sampling-based algorithm for choosing the best compression scheme for any piece of data, discussed in Section 3. (3) A novel floating-point scheme called *Pseudodecimal Encoding*, which we describe in Section 4. (4) An extensive evaluation of BtrBlocks in Section 6 using the Public BI benchmark, a collection of real-world, heterogeneous, and complex business intelligence datasets. BtrBlocks is open source and available at *https://github.com/maxi-k/btrblocks*.

2 BACKGROUND

Outline. In this section, we introduce existing open data lake formats before describing the encodings used in BtrBlocks.

2.1 Existing Open File Formats

Parquet & ORC. Apache Parquet and Apache ORC are open source, column-oriented formats widely supported by modern analytics systems. Like BtrBlocks and most column stores, they apply block-based columnar compression. Both are quite similar, but Parquet is more widely used, which is why we focus on it.

Column encoding in Parquet. Parquet encodes columns using a fixed selection of encoding schemes. The supported encodings are Run-length Encoding (RLE), Dictionary, Bit-packing and variants of Delta Encoding [13]. Which encoding to use is either specified by the user or decided with hard-coded, implementation-specific rules. After encoding chunks of multiple columns, Parquet bundles the results into rowgroups. Multiple rowgroups are combined into a Parquet file, with metadata about each stored in the footer.

Metadata & Statistics. Each Parquet file includes metadata, statistics and lightweight indices. While important for query processing, we believe these are misplaced in the data file. One would like to prune data using statistics and indices *before* accessing a file through a high-latency network. We thus follow a different approach by decoupling compression from the rest of the file format: BtrBlocks only produces blocks of compressed data with a configurable size. Metadata, statistics and indices are completely orthogonal and may be added on top or tracked separately.

Additional general-purpose compression. The set of available encoding schemes in Parquet is small and the rules it uses to choose per-column encoding schemes are simplistic. For example, the default C++ implementation simply tries dictionary compression and leaves the data uncompressed if the dictionary grows too large [3, 54]. As a result, the achieved compression ratios are low in practice. To remedy this, encoded Parquet columns are often compressed again with a general-purpose, heavyweight compression scheme. The scheme is configurable [20] and the set of available options includes Snappy, Brotli, Gzip, Zstd, LZ4, LZO and BZip2. We show results for Zstd and Snappy, which provide two different trade-offs between compression effectiveness and decompression speed. LZ4 [14] behaved very similar to Snappy in our experiments.

A better way to compress. We found that general-purpose schemes on top of simple encodings are quite inefficient to decompress and thus refrain from using them. Instead, BtrBlocks expands on the selection of lightweight encodings Parquet offers. Additionally, it substantially improves the scheme selection algorithm and allows for applying multiple encoding schemes recursively.

2.2 Compression Schemes Used In BtrBlocks

Combining fast encodings. The idea of BtrBlocks is to combine multiple type-specific efficient encoding schemes that cover different data distributions and therefore achieve a high compression ratio while keeping decompression fast. Table 1 lists the encoding schemes we use in BtrBlocks. BtrBlocks compresses columns of typed data (integers, double floating-point numbers and variable-length strings). Like many existing formats [8, 15, 26, 36, 38, 39, 53], it divides each column into

Table 1. Encoding Schemes used in BtrBlocks

Scheme	Reference	Code	Type
RLE		our	all
One Value		our	all
Dictionary		our	all
Frequency		our	all
SIMD-FastPFOR	[42]	[1]	int
SIMD-FastBP128	[42]	[1]	int
FSST	[26]	[2]	string
Roaring	[43]	[7]	bitmap
Pseudodecimal	Section 4	our	float

fixed-size blocks with a default size of 64,000 entries. Compressing blocks individually allows BtrBlocks to react to changing data distributions by adapting the compression scheme to the data in each block. Blocks also facilitate parallelizing compression and decompression. BtrBlocks is based on a number of existing encoding schemes, which we briefly describe below.

RLE & One Value. *Run-length Encoding (RLE)* is a ubiquitous technique that compresses runs of equal values. Instead of storing the run $\{42, 42, 42\}$, for example, we store $(42, 3)$. *One Value* is a specialization for columns with only one unique value per block.

Dictionary. Another simple but effective scheme is *Dictionary Encoding*, which replaces distinct values in the input with (shorter) codes. A lookup structure (the dictionary) maps each code to the original distinct value. The data structure used for implementing the dictionary is determined by the encoded type, e.g., an array for fixed-size values and a string pool with offsets for variable size values. In some lightweight formats [36], dictionaries are often the only way of compressing strings.

Frequency. Skewed distributions, where some values are much more common than the rest, are not uncommon in real-world datasets. DB2 BLU [53] proposed a *Frequency Encoding* that uses several code lengths based on data frequency. For example, a one bit code can represent the two most frequent values, a three bit code the next eight most frequent values, and so on [53]. In BtrBlocks, we adapt Frequency Encoding based on our analysis of real-world data [17]: Often, a column only has one dominant frequent value, with the next most frequent values occurring exponentially less often. We optimize for this case by only storing (1) the top value, (2) a bitmap marking which values are the top value and (3) the exception values which are not the top value.

FOR & Bit-packing. For integers, *Frame of Reference (FOR)* encodes each value as a delta to a chosen base value. For example, instead of storing $\{105, 101, 113\}$, we can choose the base 100 and store $\{5, 1, 13\}$ instead. This can be useful in combination with *Bit-packing*, which truncates unnecessary leading bits. After applying FOR to our example sequence, we can bit-pack $\{5, 1, 13\}$ using 4 bits for each value instead of 8 bits. However, the basic FOR scheme does not work well with outliers: adding 118 to the example sequence would require us to use at least 5 bits for each value. *Patched FOR* (PFOR) thus stores these outliers as exceptions and keeps the smaller bitwidth for the rest of the values [61]. *SIMD-FastPFOR* and *SIMD-FastBP128* build on this idea and specialize the algorithms and layout for SIMD [42]. We use these existing high-performance implementations in BtrBlocks.

FSST. A large portion of real-world data is stored as strings [33, 49]. *Fast Static Symbol Table (FSST)* is a lightweight compression scheme for strings [26]. It replaces frequently occurring substrings of

up to 8 bytes with 1 byte codes. These codes are tracked in a fixed-size 255 entry dictionary: the symbol table. The symbol table is immutable and used for an entire block of strings. Decompression is simple and therefore fast: FSST uses codes from the compressed input as an index into the symbol table and copies the substring to the output. Compression is more involved because FSST needs to find a good symbol table first. BtrBlocks either uses FSST to compress strings from the input directly or applies it to a dictionary when beneficial.

NULL Storage Using Roaring Bitmaps. BtrBlocks stores `NULL` values for each column using a *Roaring Bitmap* [43]. The idea behind Roaring is to use different data structures depending on the local density of bits. This makes it highly efficient for many data distributions [57]. BtrBlocks uses Roaring Bitmaps through an open source C++ library that is optimized for modern hardware [7, 44]. Besides tracking `NULL` values, we also use Roaring Bitmaps to track exceptions for internal encoding schemes like Frequency Encoding.

Cascading Compression. With *FOR + Bit-packing*, we mentioned the idea of compressing the output of an encoding with another encoding to further reduce space. This concept has been named *Cascading Compression* [30]. Damme et al. [31] classify several encoding schemes into *logical* and *physical* compression schemes and study how well they combine. They develop a gray-box cost model for integer compression to tackle the problem of choosing good schemes for a given dataset. However, they limit themselves to integer columns and combinations of at most two algorithms (single-level cascade). We present a more generic approach that handles multi-level cascades and includes doubles and strings as well. Additionally, our scheme selection algorithm avoids cost models and opts for an easily-extendible sampling-based approach.

3 SCHEME SELECTION & COMPRESSION

Scheme selection algorithms. In Section 2.2, we presented encoding schemes for different data types. The effectiveness of these encodings differs strongly depending on the data distribution. Given a set of encodings, we therefore need an algorithm for deciding which encoding is most effective for a particular data block. Simple, static heuristics as used by Parquet – such as always encoding strings with dictionaries and always bit-packing integers – are not capable of exploiting the full compression potential of a particular dataset. Another approach would be to rely on data statistics. For formats like Data Blocks [36] a small number of statistics such as min, max and unique count are sufficient to select among a small set of simple encodings (FOR, dictionary, single value). However, for more complex encodings, simple statistics are not enough, and a general solution would require to exhaustively compress the data with each encoding. Even for a moderate number of encodings, this would be prohibitively slow – even without taking cascading into account, which could increase the search space exponentially.

Challenges. A better approach for encoding selection is to use sampling. For this to work well, the sample must capture the dataset characteristics relevant for compression. Random sampling, for example, may not work well for detecting whether RLE is effective. Simply taking the first k tuples, on the other hand, would result in a very biased sample. Another challenge for the scheme selection algorithm is to take cascading into account, i.e., it must decide whether to encode already encoded data again.

Solution overview. In BtrBlocks, we test each encoding scheme on a sample and select the scheme that performs best. As Section 3.1 describes, our sampling algorithm tries to find a compromise between preserving the locality of neighboring tuples and accurately representing the entire data range. Section 3.2 describes how BtrBlocks integrates cascading with our sample-based scheme selection recursively. Given a block of data to compress, each recursion level executes the following steps:

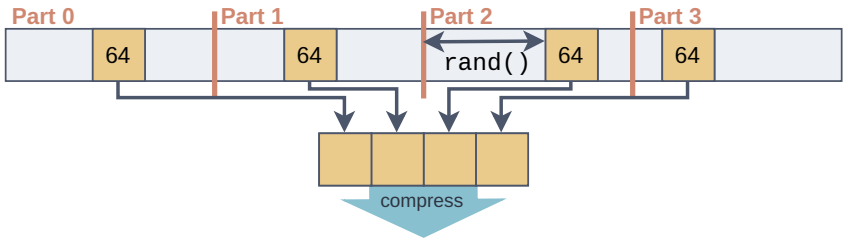


Fig. 2. Choosing a random sample from a column block

(1) Collect simple statistics about the block.
(2) Based on these statistics, filter non-viable encoding schemes.
(3) For each viable scheme, estimate the compression ratio using a sample from the data.
(4) Pick the scheme with the highest observed compression ratio and compress the entire block with it.
(5) If the output of the compression is in a compressible format, then repeat from step 1.

3.1 Estimating Compression Ratio with Samples

Choosing samples. To select the best scheme for each block, the sample has to be representative of the data. The main trade-off is between preserving spatial locality in the data while still capturing the distribution of unique values across the input. At the same time, samples have to stay small to keep scheme selection overhead low. As Figure 2 illustrates, we propose to select multiple small `runs` from random positions in non overlapping `parts` of the data. For a chunk size of 64,000 values, we use 10 runs of 64 values each, resulting in a sample size of 1% of the data. We have found this method to yield a good compromise between compression speed and estimation quality, and evaluate this in detail in Section 6.3.

Estimating compression ratio. BtrBlocks first collects statistics like *min*, *max*, unique count and average run length in a single pass. Based on these statistics, it then applies heuristics to exclude nonviable schemes: It excludes RLE, for example, if the average run length is < 2 and Frequency Encoding if $\geq 50\%$ of values are unique. BtrBlocks then compresses the sample with each viable encoding scheme to estimate the compression ratio of each scheme.

Performance. We evaluated the performance of this method for sampling and compression ratio estimation on real-world data. Our selection algorithm uses only 1.2% of the total compression time while accurately estimating which compression scheme is best.

3.2 Cascading

Recursive application of schemes. After selecting a compression algorithm, the output (or some part of it) may be compressed using a different scheme. This is illustrated in Figure 3, with `recursion points` denoting an additional possible compression step. The `scheme` used for the additional step is again selected with our compression ratio estimation algorithm. The maximum number of recursions is a parameter of the compression algorithm, with the default value set to 3. Once this recursion depth is reached, BtrBlocks leaves the data uncompressed.

Cascading compression: An example. Taking an input of doubles {3.5, 3.5, 18, 18, 3.5, 3.5}, for example, the sampling algorithm may determine that *RLE* is a good choice. This produces two outputs: A value array of doubles {3.5, 18, 3.5} and a run length array {2, 2, 2}. BtrBlocks will decide to compress the run length array using *One Value* using the statistics. The value array is also subject to a cascading compression step. Assuming the estimation algorithm chooses *Dictionary*

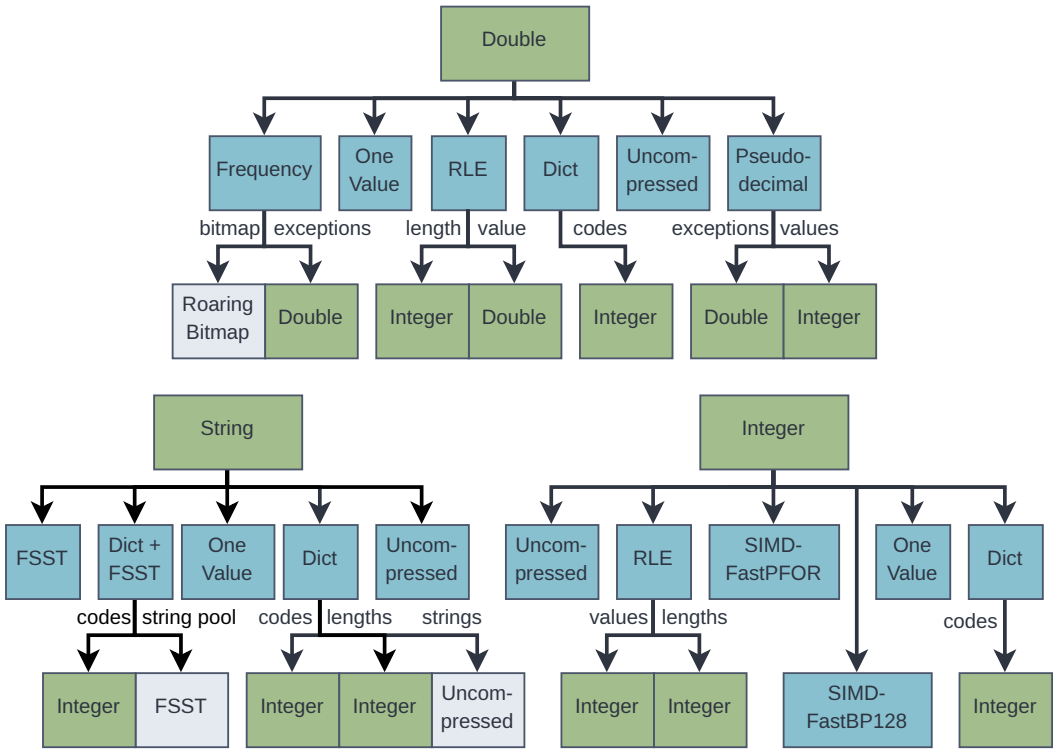


Fig. 3. Encoding scheme decision trees that we apply recursively

Encoding, this will yield a code array {0, 1, 0} and a dictionary {3.5, 18}. As the maximum recursion depth is not yet reached, BtrBlocks may decide to apply *FastBP128* to the code array in a final step. Decompression works analogously, with each scheme storing what scheme it cascaded into and applying the decompression algorithms in reverse order.

Code example. Listing 1 shows a crosscut of the entire cascading compression algorithm for integers using RLE as an example. The RLE `ratio estimation` method stops early if the scheme is not feasible, otherwise it uses the sampling algorithm. The displayed part of the RLE `compress` method shows the recursive calls to the scheme picking algorithm. In this case, there are two recursive calls: One for the values list and one for the run lengths. The `scheme picking` algorithm simply tests all schemes if the maximum recursion depth is not yet reached.

The encoding scheme pool. The result is a generic, extensible framework for cascading compression that draws from a pool of arbitrary encoding schemes. The scheme pool strongly affects the overall behavior of BtrBlocks: With more schemes, compression becomes slower because more samples have to be evaluated, but the compression ratio increases. Adding more heavyweight schemes may also increase the compression ratio but slows down decompression. We have chosen the set of schemes in BtrBlocks based on our analysis of the diverse set of columns in the Public BI datasets. To build up the encoding scheme pool BtrBlocks uses, we iteratively (1) found columns where its compression ratio was worse than heavyweight schemes like Bzip2, (2) analyzed patterns in the data, (3) added schemes that fit those patterns well and (4) pruned schemes that did not improve compression enough or slowed down decompression. The result is the list of `schemes` shown in Figure 3.

```
struct RLEData { u8 val_scheme, cnt_scheme, data[] }

double RLE::estimateRatio (Stats& s)
    if(s.average_run_length < 2) return 0
    return estimateFromSamples(stats)

u32 RLE::compress (u32* src, u32 cnt, u8* dst, u8 recur)
    RLEData& res = *((RLEStructure*)dst)
    vector<u32> values, counts
    ... // <- RLE algorithm, writing to vectors
    // cascading compression for values:
    Scheme val_casc = pickScheme(values.data(), cnt, res.data, recur - 1)
    res.values_scheme = val_casc.scheme_code()
    u8* cnt_dst = res.data+val_casc.compress()
    // cascading compression for counts:
    Scheme cnt_casc = pickScheme(counts.data(), cnt, cnt_dst, recur - 1)
    res.counts_scheme = cnt_casc.scheme_code()
    return cnt_dest+cnt_casc.compress() - dst

Scheme pickScheme (u32* src, u32 cnt, u8* dst, u8 recur)
    if (!recur) return UNCOMPRESSED
    auto stats = genStats(src, cnt)
    auto scheme = UNCOMPRESSED; double min_cf = -1
    for (auto& sc : pool)
      double est = sc.estimateRatio(stats)
      if (est != 0 && est > min_cf)
        min_cf = est
        scheme = sc
    return scheme
```

Listing 1. Pseudocode of the scheme picking algorithm and RLE as an example for an implemented scheme

4 PSEUDODECIMAL ENCODING

Floating-point numbers in relational data. Prior research on floating-point compression in relational databases is very sparse. The lack of interest in floating-point compression schemes has a historic reason: Relational systems usually represent real numbers as `Decimal` or `Numeric`, which can physically be stored as integers. However, this is changing with the move to data lakes and the subsequent integration with non-relational systems: Tableau's internal analytical DBMS, for example, encodes all real numbers as floating-point numbers [56], and machine-learning systems rely on floating-point numbers virtually exclusively.

Pseudodecimal Encoding. While some encoding schemes shown in Figure 3 are applicable to all data types, the two bit-packing techniques and FSST are not effective for floating-point numbers. We thus introduce *Pseudodecimal Encoding*, a compression scheme specifically designed for binary floating-point numbers. We establish the basic idea, the encoding logic and the integration into BtrBlocks in this section, before describing efficient decompression in Section 5. We evaluate the scheme both separately and as part of BtrBlocks as a whole in Section 6.5.

```
const unsigned max_exp = 22, exp_exception = 23;
const double frac10[] = {1.0, 0.1, 0.01, 0.001,...};
struct Decimal { int digits, exp; double patch };

Decimal encode_single(const double input)
  int exp; int digits; bool neg = input < 0
  double dbl = neg ? -input : input
  if (input == -0.0 && std::signbit(input))
      goto patch // -0.0 is exception
  // Attempt conversion
  for (exp = 0; exp <= max_exp; exp++)
    double cd = dbl / frac10[exp]
    digits = round(cd)
    double orig = ((double) digits) * frac10[exp]
    if (orig == dbl) goto success
  patch: // return exception in exponent, patch
    return {0, exp_exception, input}
  success: // return decimal; patch is ignored
    return {(int) digits, exp, 0}
```

Listing 2. Pseudodecimal Compression algorithm

4.1 Compressing Floating-Point Numbers

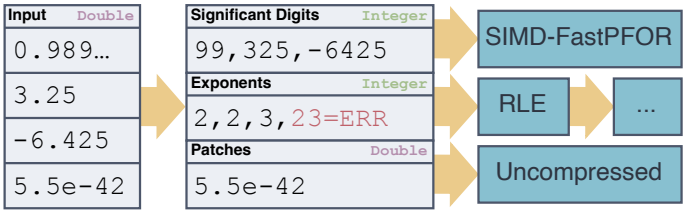
Challenges. Pseudodecimal Encoding sprung from our analysis of the Public BI Benchmark. We found that double-precision floating-point numbers are frequently used where fixed-precision numbers would be sufficient (and indeed better suited). A common example is storing monetary prices such as $3.25 or $0.99 as floating-point numbers. While such values may appear to be highly compressible, there are two problems: First, their physical IEEE 754 representation (1 sign, 11 exponent and 52 mantissa bits) means that standard techniques such as FOR+Bit-packing are not effective. This is because the most significant bits storing the exponent differ strongly even for numbers that are numerically fairly close (e.g., 3.25 and 0.99). Second, some decimal numbers such as 0.99 cannot be represented precisely in binary; the actual value stored is 0.98999..., which results in periodic and hard-to-compress mantissas like `0xfae147ae147ae`. In a lossless compression scheme such as BtrBlocks, decompression has to yield a bitwise-identical output to avoid changing semantics.

Floating-point numbers as integer tuples. As the name suggests, Pseudodecimal Encoding uses a *decimal* representation for encoding doubles. It does this using two integers: *significant digits with sign* and *exponent*. For example, 3.25 becomes (+325, 2), as in 325×10^{-2}. But what happens to a double such as 0.9899 . . . ?, which is 0.99 stored as a double? Intuitively, one would have to store two integers (98999 . . . , 17) to be able to restore the precise double value later. Surprisingly, storing (99, 2) suffices; this effectively compresses the floating-point value `0x3fefae147ae147ae`, to a pair of integers *(*`0x63, 0x2`*)*. Thus, the compression value of Pseudodecimal Encoding is twofold: First, it strips apart IEEE 754 floating-point values into integers that are more easily compressible. Second, it generates a compact decimal representation for hard-to-compress doubles, which is often what users wanted to store in the first place. To do this it has to find a compact decimal representation as we describe next.

Encoding Algorithm. The Pseudodecimal Encoding algorithm determines the compact decimal representation by testing all powers of 10 and checking whether any of them correctly multiply the double to an integer value. Listing 2 shows this algorithm adapted for encoding a single double instead of an entire block like in BtrBlocks. We store the inverse powers of 10 in a static table to avoid recomputing them for every number[1]. The overloaded double number ±0.0 creates an issue because we encode the sign together with the number as an integer. Thus, the algorithm handles negative zero, as well as other special floating-point numbers like $\pm Inf$ and $\pm NaN$, as exceptions. It stores these exceptions separately as patches, together with doubles that it cannot encode as integers, such as 5.5×10^{-42}. We limit the number of bits used for the digits and the exponent to 32 and 5, respectively. These properties ensure that the encoding produces bitwise-identical results.

4.2 Pseudodecimal Encoding in BtrBlocks

Cascading to integer encoding schemes. Pseudodecimal Encoding converts a column of floating-point numbers to two integer columns and a small column of exceptions. BtrBlocks may encode these columns again using cascading compression:



The depicted choices for the cascading compression are examples and not fixed; BtrBlocks chooses the schemes using its sampling algorithm as described earlier.

When to choose Pseudodecimal Encoding. There is data for which Pseudodecimal Encoding is ill-fitted, like columns with many exception values: Pseudodecimal Encoding slightly increases the compression ratio, but decompression is slow because of the many exception values. We thus disable the scheme for columns that have more than 50% non-encodable exception values. Similarly, columns with few unique values usually compress almost as well with dictionaries, which have a much higher decompression speed. In the context of BtrBlocks, we thus choose to exclude Pseudodecimal Encoding for columns with less than 10% unique values.

5 FAST DECOMPRESSION

Decompression speed is vital. Renting compute nodes is one of the main sources of cost in cloud data analytics [41]. Saving cost is therefore best done by reducing the rental time of those nodes. Considering a compression technique, we can do this by (1) reducing network load time with a good compression ratio and (2) reducing compute time with fast decompression. After achieving a good compression ratio with our cascading compression algorithm, we thus turn our attention to decompression throughput.

Improving decompression speed. As Table 1 shows, BtrBlocks uses existing highly-optimized (SIMD) implementations of SIMD-FastPFOR, SIMD-FastBP128, FSST and Roaring. In this section, we describe fast implementations of the other encodings. All presented performance numbers pertain to the Public BI Benchmark datasets discussed in Section 6.1. We measure the performance

[1]Conceptually, it might be more intuitive to divide by powers of 10, but multiplication is slightly faster than division during decompression.

```
void decodeRLEAVX (int *dst, int *runlen,
                   int *values, int runcnt)
  // dst must have >= 32 additional bytes
  for (int run = 0; run < runcnt; run++)
    int *target = dst + runlen[run]
    __m256i vals = _mm256_set1_epi32(values[run])
    for (; dst < target; dst += 8)
      _mm256_storeu_si256(dst, vals)
    dst = target // SIMD may have overflowed
```

```
void decodeDictAVX (int *dst, const int *codes, const int *values, int cnt)
  int idx = 0 // not shown: 4x loop unroll
  if (cnt >= 8)
    while (idx < cnt-7)
      __m256i codes = _mm256_loadu_si256(codes)
      __m256i values = _mm256_i32gather_epi32(values, codes, 4)
      _mm256_storeu_si256(dst, values)
      dst += 8; codes += 8; idx += 8
  for (;idx < cnt; idx++)
    *dst++ = values[*codes++]
```

Listing 3. Vectorized RLE and Dictionary decompression

improvements "end-to-end", meaning for an improved encoding scheme B that is part of the cascade $A - B - C$, we measure the resulting speedup in decompression across the entire cascade $A - B - C$.

Run Length Encoding. The standard RLE decompression algorithm replicates the value of a length-N run N times to the output. To vectorize RLE using AVX2, we perform 8 (4) simultaneous replications for integer (double) runs. However, run lengths are often not divisible by 8 (4), which we would need to handle in an expensive branch. We instead opt for writing behind the end of the output buffer in this case. The buffer length is corrected afterwards as shown on the last line of Listing 3 **(top)**. This gains an average of 76% end-to-end decompression performance for blocks that use RLE at some point in their cascade. Integer columns even decompress 128% faster on average because RLE is commonly chosen by the scheme selection algorithm. String dictionaries often use RLE to compress the code sequence and thus also gain 78% performance on average. Doubles gain 14% performance on average.

Dictionaries for fixed-size data. The standard decompression algorithm for dictionaries simply scans the code sequence and replaces each code with its value from the dictionary. We can copy 8 integer dictionary entries simultaneously using 8×32 = 256 bit AVX2 vector instructions, as shown in Listing 3 **(bottom)**. Double decoding works analogously with 4 entries. We also manually unroll the loop 4 times for both data types. For any blocks that use Dictionary Encoding in the cascade, we saw an end-to-end speedup of 18% for integer decompression and 8% for double decompression.

String Dictionaries. We avoid copying strings during decompression. Instead, BtrBlocks replaces each code with the string length and the offset (≈ pointer) of the uncompressed string. Offset and length form a fixed-size 64 bit tuple, so we can use the same vectorized algorithm we use for double dictionary decompression. Just by avoiding the string copy, we saw a speedup of more than 10× for

some low-cardinality columns. We additionally vectorize dictionary decompression, which yields another 13% end-to-end speedup.

Fusing RLE and Dictionary decompression. The scheme selection algorithm often compresses the (integer) code sequence of a dictionary with RLE. It is thus worth optimizing for this case specifically. The standard implementation generated by the cascading algorithm first decodes runs of dictionary codes into an intermediate array and then looks those up in the dictionary. We can fuse these operations and get rid of the intermediate array, instead doing the dictionary lookup first and directly writing runs of *(offset, size)* pairs. BtrBlocks does this in the vectorized manner discussed previously, but only applies the technique if the average run length is greater than 3 as we have found it to have a negative impact otherwise. This increases the end-to-end decompression performance for string columns using RLE by another 7%.

FSST. FSST exposes an API for decompressing a single string, taking the encoded string offset and length as an argument [19]. We can use this API to compress an entire block by simply calling it in a loop for each string in the input data. This, however, moves CPU time out of FSSTs optimized decompression loop and into edge-case detection. We can avoid this overhead by passing the offset of the first encoded string and the sum of all string lengths to the decompression API instead. In microbenchmarks, this yielded a reduction of 50 instructions per string, independent of string length. Additionally, we can forgo storing the offsets and lengths of compressed strings; storing uncompressed string lengths suffices.

Pseudodecimal. We implemented the decompression algorithm of our novel double encoding scheme using vector instructions. To reconstruct a double, the decompression simply multiplies the significant digits of each value with the respective exponent. This can be easily vectorized (`_mm256_cvtepi32_pd`, `_mm256_mul_pd`), producing blocks of 4×64 bit doubles at once. However, exception values that could not be encoded during compression complicate matters: As explained in Section 4, Pseudodecimal Encoding stores these exceptions separately as patches. The decompression algorithm thus first checks for exceptions in each vectorization block using a Roaring Bitmap. If there are none, it proceeds with vectorized decompression. Otherwise, it falls back to a scalar implementation for the current block and inserts any patch values into the output.

6 EVALUATION

Test setup. We execute all experiments on a c5n.18xlarge AWS EC2 instance. Previous work suggests that c5n is a good instance for analytics in the cloud, primarily because of its 100 Gbps networking [41]. It runs an Intel Xen Platinum 8000 series (Skylake-SP) CPU with 36×3.5 GHz cores (72 threads), offers the AVX2 and AVX512 instruction sets and has 192 GiB of memory. Code is compiled with GCC 10.3.1 on Amazon Linux 2, kernel version 5.10. We use the TBB library [16] for parallelization and disable hyperthreading. Our benchmarks allocate and touch all memory beforehand to avoid page faults. We repeat all measurements and average the results to minimize the effects of caching and CPU frequency ramp-up.

Parquet test setup. For generating Parquet files, we tested both the Apache Arrow (pyarrow 9.0.0) and the Apache Spark (pyspark 3.3.0) libraries. The only parameter change we made was setting the rowgroup size in Apache Arrow to 2^{17} because we found that to be fastest. We implemented the actual benchmarks consuming the generated Parquet files with the Arrow C++ library. This library offers a high-level API based on Arrow constructs and a low-level API that uses Parquet directly. The high-level interface was significantly slower in our tests, so we chose the low-level API in all tests. We parallelized decompression over both rowgroups and columns.

Table 2. Public BI Benchmark (PBI) and TPC-H: Comparison of data types by volume (**sh**) and compression ratio (**cr**)

datatype	String				Double				Integer				Combined	
dataset	PBI		TPC-H		PBI		TPC-H		PBI		TPC-H		PBI	TPC-H
metric	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	sh [%]	cr [×]	cr [×]	cr [×]
Binary	71.5	—	61.7	—	14.4	—	19.5	—	14.1	—	18.7	—	—	—
Parquet	51.0	7.10	64.1	1.63	36.6	1.99	14.0	2.35	12.5	5.73	21.9	1.45	3.37	1.69
+LZ4	39.8	12.05	46.8	3.65	44.6	2.16	18.4	2.94	15.6	6.07	34.8	1.49	4.72	2.77
+Snappy	39.3	12.23	45.0	3.92	44.9	2.15	19.2	2.91	15.7	6.05	35.8	1.49	4.79	2.85
+Zstd	33.6	**17.13**	40.0	**5.27**	50.1	2.30	23.3	2.87	16.3	**6.97**	36.7	1.74	**6.05**	3.41
BtrBlocks	43.6	11.32	54.9	4.26	41.9	**2.36**	16.2	**4.58**	14.5	6.70	28.9	**2.46**	5.28	**3.79**
Average		10.14		3.29		1.99		2.78		5.42		1.60	4.20	2.90

6.1 Real-World Datasets

Synthetic data. Analytical benchmarks such as TPC-H and TPC-DS have proven useful for evaluating both traditional and cloud-native query engines [55]. However, it is also well-known that their data generation algorithms do not necessarily produce realistic data distributions [33, 40, 56]. Assumptions like complete data normalization, uniform and independent distributions, or most of the data being integers do not reflect typical real-world data – particularly in data lakes. We therefore argue that compression algorithms should be evaluated using real-world rather than synthetic datasets.

The Public BI Benchmark. The large real-world collection of datasets we chose to focus on is the *Public BI Benchmark* [33]. It contains datasets derived from the 46 largest Tableau Public workbooks at the time of creation [56]. We thus expect its contents to be more representative of what one might find in today's large data lakes: Data skew, denormalized tables, misused data types (e.g. proliferation of strings) and non-uniform `NULL` representations resulting from the variety of heterogeneous data sources. Additionally, Tableau stores decimal values as floating-point numbers – a data type which we found to be frequently underrepresented in compression literature [56] and which is becoming more important due to the proliferation of machine learning. To get a better understanding of the Public BI Benchmark and its effect on compression performance, we first take a closer look at its datasets.

Public BI vs. TPC-H. Table 2 outlines the differences between a real-world dataset and a generated dataset by comparing the Public BI datasets with TPC-H data. We do this for each data type separately. Because TPC-H can be generated on different scale factors, we use the relative data volume of each data type as a metric instead of an absolute amount. In addition to the *uncompressed* format, for which we use our in-memory columnar binary representation, we convert each dataset to Parquet using multiple compression schemes, as well as BtrBlocks. We then reexamine the data volume of each data type in the compressed formats, yielding a compression ratio per data type and dataset. In the following, we describe our observations about the differences between the Public BI Benchmark and TPC-H in more detail.

Public BI vs. TPC-H: Strings. As Table 2 shows, the Public BI Benchmark consists of 71.5% strings, compared to TPC-H with 61.7%. Additionally, many strings in the Public BI Benchmark are structured, like URLs and product identifiers with common prefixes. In contrast, the largest

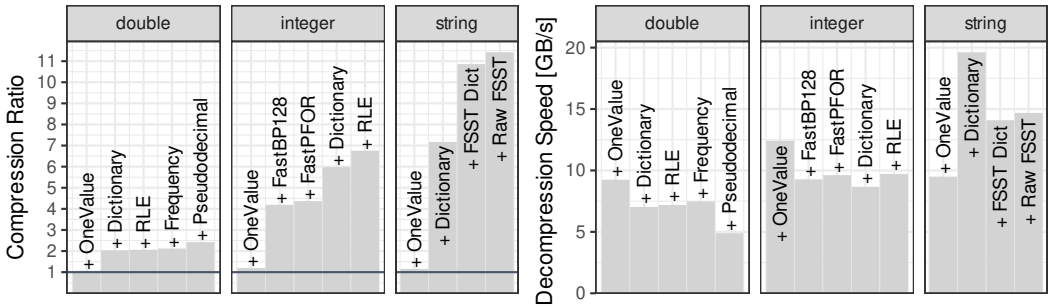


Fig. 4. Compression ratio and decompression throughput changes when successively enabling techniques, by data type

strings in TPC-H – the comment columns – are random samples from a pool of test data. This has a large impact on compression performance: Where the average compression ratio for strings in the Public BI Benchmark is 10.2× across all measured formats, it is only 3.3× in TPC-H.

Public BI vs. TPC-H: Doubles. Doubles make up 19.5% of the data volume in TPC-H, but only 14.3% in the Public BI Benchmark. Across all tested compression schemes, doubles compress with a ratio of 1.99 in the Public BI Benchmark and 2.78 in TPC-H on average. The most likely reason for this are the numeric ranges: Double columns in TPC-H usually contain price data from one size range. They are thus better suited for compression, especially with Pseudodecimal Encoding introduced in Section 4.

Public BI vs. TPC-H: Integers. TPC-H consists of 18.7% integers by volume, compared to the Public BI Benchmark with 14.1%. Integers from Public BI datasets compress with an average factor of 5.4 across all measured formats, and only 1.6 in TPC-H. This effect stems mainly from the unrealistically normalized data TPC-H contains: Most integers are unique keys or foreign keys, and few columns contain runs or repeating patterns. In contrast, the Public BI Benchmark contains denormalized tables where joins result in runs and repeating patterns. This is clearly visible, for example, in samples from the largest two Public BI datasets [9, 10]. Extreme cases like the all-zero integer column "RealEstate1/New Build?" shown in Table 4 are also missing from TPC-H.

Adapting for evaluation. We use a subset of the datasets included in the Public BI Benchmark and adapt them to our use case. From each dataset, we only use the largest table: for example, we only use `TrainsUK1_Table4` from the `TrainsUK1` dataset. We do this because the tables in each dataset are often derived from each other and thus very similar; using only one table per dataset prevents over-representing the data mix of larger datasets. Due to their negligible sizes, we also exclude the datasets `IUBLibrary`, `IGlocations` and `Hatred1` as well as the date and timestamp columns (which can be represented as integers). This adapted subset of the Public BI Benchmark totals 119.5 GB of binary data when loaded into memory. It contains 43 tables, each containing between 6 and 519 columns, 57 on average. Overall, there are 2451 columns with diverse data shapes and distributions. Even though this makes the Public BI Benchmark much more suitable for designing and evaluating a compression scheme, we also perform experiments with TPC-H so that BtrBlocks can be more easily compared with related work.

6.2 The Compression Scheme Pool

Measuring the impact of individual techniques. The list of encoding techniques BtrBlocks tests with each cascading step forms a trade-off between compression ratio and decompression speed. We evaluated this by successively adding techniques to the pool and measuring the resulting

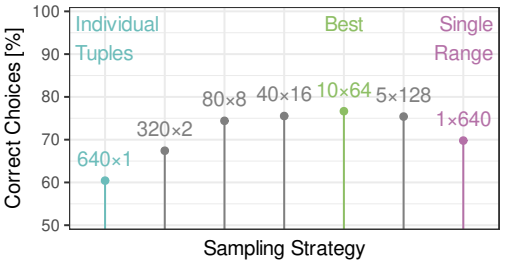


Fig. 5. Correct scheme choices per strategy (N = 640)

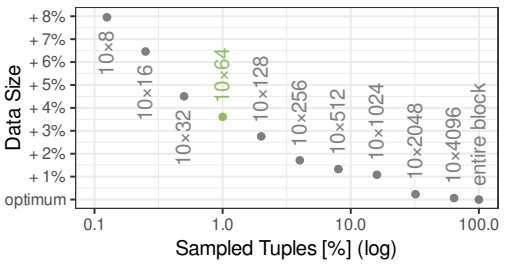


Fig. 6. Public BI compressed size for different sample sizes

compression ratio and decompression speed. Figure 4 shows one sequence of technique additions for each data type. For this experiment, we use a single thread for decompression to avoid measuring noise created through concurrency.

Impact on compression ratio. For doubles, Dictionary Encoding and Pseudodecimal Encoding have the largest impact with a 95% and 20% respective improvement. Still, as expected, doubles are inherently less compressible than integers and strings. We achieve the best average compression ratio on strings, where Dictionary Encoding yields the largest improvement (7×). Using FSST to compress an existing dictionary improves the compression ratio by another 51%. FSST applied to raw data slightly improves compression ratio and decompression speed. One Value barely increases the average compression ratio, but has a large impact on some columns both in compression ratio and speed (cf., Table 4).

Impact on decompression speed. One Value is also fastest in terms of decompression for doubles and integers, yielding an average respective throughput of 8.9 and 11.8 GB/s. For string decompression, Dictionary Encoding increases throughput from 9.4 GB/s to 19.6 GB/s. This is because in BtrBlocks, Dictionary Encoding only decompresses the code sequence into pointers to the dictionary contents and can forgo copying strings.

6.3 Sampling Algorithm

Sampling research questions. Accurately estimating the compression ratio for different schemes requires choosing a good sampling algorithm. We do this by answering two research questions:

(1) Given a fixed sample size, what is the best sampling strategy?
(2) How does sample size relate to scheme selection accuracy?

We score sampling strategies based on the percentage of correctly selected schemes, which we compute as follows: We compress the first block (64k tuples) of every column in the Public BI Benchmark using every scheme, including cascades, and determine the scheme with the best compression ratio: the optimal scheme. We do the same again for each sampling strategy, compressing the sample instead of the entire block. If a sampling strategy chooses the optimal scheme or a scheme at most 2% worse than the optimal, we consider the scheme choice to be correct[2].

Best strategy for a fixed sample size. Figure 5 shows the percentage of correctly selected schemes for different sampling strategies that always sample 640 tuples (≙1% of a 64k block). It includes extreme cases like sampling `random individual tuples` or choosing a `single tuple range`, which perform worst. The main takeaway is that sampling multiple small chunks across the entire block improves accuracy compared to other strategies, though there is little difference between

[2]Allowing almost-optimal schemes filters cases where two scheme cascades compress the same data almost equally well, e.g., Dict→RLE vs RLE→Dict.

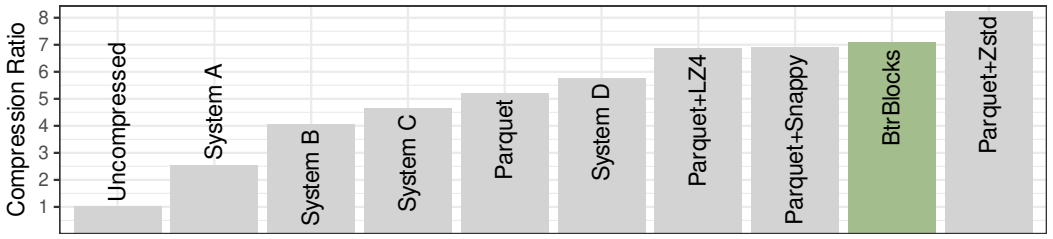


Fig. 7. Public BI compression ratios for proprietary column stores (A-D), Parquet and BtrBlocks

strategies that choose chunks of ≥ 16 tuples. This confirms the intuition that the sample needs to capture both data locality and data distribution across the entire block.

Impact of sample size. We now want to evaluate the impact of the overall sample size on compression ratio. Figure 6 shows the loss in compression ratio compared to the best possible cascade for different sample sizes. Larger samples yield a better compression ratio at the cost of exponentially growing CPU overhead.

Sampling in BtrBlocks. For `BtrBlocks`, we thus choose to sample 10×64 tuples $\hat{=}$ 1% of each block by default. This uses 1.2% of CPU time during compression and results in 77% correct scheme choices. With these choices, BtrBlocks compresses only 3.3% worse than the optimum on average.

6.4 Compression

Compression ratio. We designed BtrBlocks with relational data in mind, e.g., storing aligned columns that form tuples. We thus compared its compression ratio with four relational column stores on the Public BI datasets. These systems base their compression on internal proprietary formats. To show as complete a picture as possible, we also added the most popular open source format, Apache Parquet, to the comparison. Parquet provides different built-in high-level compression options. Figure 7 displays the combined results, showing that BtrBlocks beats every format except the heavyweight Zstd compression on Parquet.

Compression speed. Starting from a CSV file, compression with both Parquet and BtrBlocks consists of two steps: (1) Convert the CSV file to an in-memory format and (2) convert the in-memory format to the compressed final form. Our single-threaded compression speed results are similar to Parquet, both beginning from CSV and the in-memory format:

	From CSV	From binary	Compression Factor
BtrBlocks	**38.2 MB/s**	**75.3 MB/s**	7.06 ×
Parquet+Snappy	38.0 MB/s	41.9 MB/s	6.88 ×
Parquet+Zstd	37.3 MB/s	41.0 MB/s	**8.24** ×

6.5 Pseudodecimal Encoding

Evaluation outside of BtrBlocks. Pseudodecimal Encoding is a novel double compression scheme we designed based on our observations about data in the Public BI Benchmark. To assess its effectiveness, we want to measure its compression factor outside of BtrBlocks. However, similar to FOR, Pseudodecimal Encoding does not usually reduce data size on its own; instead, it prepares the data for compression with another scheme like Bit-packing or RLE. This makes Pseudodecimal Encoding a good fit for the cascading compression applied by BtrBlocks, but it also complicates

Table 3. Compression Ratios of Pseudodecimal (PDE), other double schemes (Public BI, large double columns)

Column	FPC	Gorilla	Chimp	$Chimp_{128}$	**PDE**
CommonGov./10	1.2	1.1	1.5	**1.9**	1.8
CommonGov./26	15.1	48.0	28.0	6.9	**75.0**
CommonGov./30	6.4	7.0	7.6	5.0	**7.8**
CommonGov./31	9.3	14.3	13.3	5.6	**23.4**
CommonGov./40	14.3	38.0	25.0	6.7	**54.6**
Arade/4	.95	1.1	1.2	1.6	**1.9**
NYC/29	1.5	2.1	**2.5**	1.7	1.0
CMSProvider/1	1.5	1.7	1.8	**2.4**	1.6
CMSProvider/9	2.7	2.3	3.4	2.4	**6.6**
CMSProvider/25	.98	.98	1.1	**1.2**	1.0
Medicare/1	1.2	1.4	1.5	**2.0**	1.5
Medicare/9	2.6	2.3	3.4	2.3	**6.3**

a standalone evaluation because the compression cascade conflates measurements from all used schemes. To remove this effect, the following evaluation of Pseudodecimal Encoding applies a fixed two-level cascade: We first compress data using Pseudodecimal Encoding and then *always* compress the output with FastBP128.

Comparing to existing double schemes. We first compare Pseudodecimal Encoding to the well-known existing double compression schemes FPC [28] and Gorilla [51], and the recently proposed Chimp and $Chimp_{128}$ [46]. Table 3 shows the compression ratio of these schemes on the largest non-trivial (e.g., more than one value) Public BI double columns. Pseudodecimal Encoding (PDE) does not compress columns with high-precision values well, like the longitude coordinate values in *NYC/29*. However, it often outperforms other schemes on columns with less precision, like the abundant pricing data columns.

Effectiveness inside BtrBlocks. In order to provide a benefit as part of the scheme pool in BtrBlocks, Pseudodecimal Encoding also has to outperform general purpose schemes like Dictionary Encoding and RLE. We compare these schemes by again applying a fixed two-level cascade, where the output of each scheme is always compressed with FastBP128. We also include non-cascading FastBP128 to check our reasoning that Bit-packing (BP) should rarely be effective on IEEE 754 floating point values:

Column	BP	Dict.	RLE	**PDE**
Gov./10	.99	1.6	1.0	**1.8**
Gov./26	60.9	4.4	**187**	75.0
Gov./30	4.7	2.9	6.9	**7.8**
Gov./31	12.2	4.5	15	**23.4**
Gov./40	38.1	4.4	**91.5**	54.6
Arade/4	.99	1.3	.96	**1.9**

Column	BP	Dict.	RLE	**PDE**
NYC/29	1.1	**2.5**	1.6	1.0
CMS./1	.99	**1.6**	1.5	**1.6**
CMS./9	1.0	5.6	.99	**6.6**
CMS./25	.99	.88	.97	**1.0**
Medi./1	.99	**1.6**	1.2	1.5
Medi./9	1.0	5.4	.99	**6.3**

Pseudodecimal Encoding (PDE) loses on columns with large runs or few unique values, where RLE and Dictionary Encoding are best. But there are columns where Pseudodecimal Encoding offers a significant benefit over other schemes. We thus believe it to be a valuable addition to the BtrBlocks encoding scheme pool.

Table 4. Compression ratios and decompression throughput on a random Public BI Benchmark column sample

ID	Type ↓	Size [MB]	Decomp. Speed Btr [GB/s]	Decomp. Speed Zstd [GB/s]	Compr. Ratio Btr [×]	Compr. Ratio Zstd [×]	Scheme (Root)	Value Example
1	string	34.0	**70.2**	10.4	1,862.6	**3,068.1**	Dictionary	null,null
2	string	59.9	**20.3**	3.7	240.5	**418.7**	Dictionary	“”,””,””
3	string	24.7	**33.6**	3.5	1,262	**1,598.5**	Dictionary	All Residential
4	string	94.2	**30.8**	2.4	**5,048.8**	2,504.1	OneValue	CABLE,CABLE,…
5	string	89.9	**15.0**	1.6	8.0	**13.6**	Dict+FSST	01 BRONX,04 BRONX
6	string	149.2	**17.1**	1.4	5.2	**7.9**	Dict+FSST	5777 E MAYO BLVD
7	string	77.9	**12.4**	1.3	5.2	**6.6**	Dict+FSST	null,BETHESDA,ATHENS
8	string	77.9	**11.8**	1.2	5.1	**7.7**	Dict+FSST	null,PHOENIX,RALEIGH
9	string	22.7	**2.7**	0.9	10.4	**28.5**	Dictionary	Maceió,Curitiba,Curitiba
10	integer	74.5	**26.6**	3.1	**13,055.7**	1,653.5	OneValue	0,0,0,…
11	integer	33.0	**7.1**	1.0	**2.4**	2.2	FastPFOR	26994,18930,7691
12	integer	28.8	**3.5**	0.8	3.0	**3.5**	FastPFOR	2704302,3547304,1200203
13	integer	28.8	**4.0**	0.8	3.0	**3.5**	FastPFOR	2704302,3547304,1200203
14	double	22.2	**5.8**	2.0	11.5	**14.0**	Dictionary	0,0,0
15	double	22.2	**2.3**	1.7	4.4	**5.9**	Frequency	83.2833,3.05,9.5999
16	double	107.6	**11.0**	1.3	4.6	**6.8**	Dictionary	0,0,0
17	double	22.2	**3.1**	0.7	**2.7**	2.4	Pseudodec.	0,0,0
18	double	24.9	**4.3**	0.7	1.3	**1.7**	Dictionary	null,null,null

$\bowtie_{ID}$

ID	Dataset/Column	ID	Dataset/Column	ID	Dataset/Column
1	SalariesFrance/LIBDOM1	7	Provider/[…]city	13	Eixo/cod_ibge_da_ue
2	MulheresMil/pcd	8	PanCreactomy1/[…]CITY	14	Telco/CHARGD_SMS_P3
3	Redfin2/property_type	9	Uberlandia/municipio[…]	15	Telco/RECHRG[…]
4	Motos/Medio	10	RealEstate1/New Build?	16	Motos/InversionQ
5	NYC/Community Board	11	Medicare1/[…]SUPPLY	17	Telco/TOTAL_MINS_P1
6	PanCreactomy1/[…]STREET	12	Uberlandia/cod_ibge[…]	18	Redfin4/[…]price[…]

6.6 Decompression

Open source formats. We compared our compression ratio with proprietary systems in Section 6.4. However, these systems do not allow us to introspect compression and decompression time independent of other system parts. In the following, we thus focus on the widely used open source formats Parquet and ORC. We described our Parquet configuration at the beginning of Section 6.
ORC test setup. We generated Apache ORC files using the Apache Arrow library (pyarrow 9.0.0). Using default settings, ORC files tended to grow large, preventing parallelism. We thus changed the `dictionary_key_size_threshold` parameter from the default (0) to the default of Apache Hive (0.8). We changed the LZ4 compression strategy from the default (`SPEED`) to `COMPRESSION` for the same reason. Changing the stripe size – the equivalent of the rowgroup size for Parquet – did not change the performance in our multithreaded tests, so we kept the default value. The

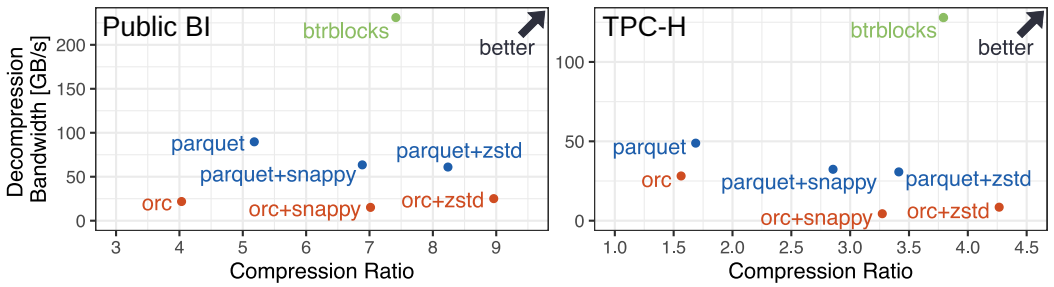


Fig. 8. Compression ratio vs. in-memory decompression bandwidth on the Public BI Benchmark (left) and TPC-H (right) for Parquet, ORC and BtrBlocks on c5n.18xlarge

actual benchmarks use the ORC C++ library, which cannot read files directly from memory. For a fair comparison, we implemented a custom variant of `orc::InputStream` that reads directly out of an in-memory buffer. Like with Parquet, we parallelized by both stripes and columns.

In-memory Public BI decompression throughput. Figure 8 (top) shows our results for the datasets we selected from the Public BI Benchmark as described in Section 6.1. We plot the compression ratio against decompression throughput (e.g., uncompressed size / decompression time) for Parquet, ORC and BtrBlocks. While Zstd compression is better with both Parquet and ORC in terms of compression ratio, **BtrBlocks** is superior in terms of decompression speed. It decompressed 2.6×, 3.6× and 3.8× faster than Parquet, Parquet+Snappy and Parquet+Zstd on average, respectively.

Decompression of Parquet vs. ORC. Interestingly, every **Parquet** variant performs better than its **ORC** counterpart in terms of decompression speed: Uncompressed ORC is 4.1× slower to decode than uncompressed Parquet as measured on the Public BI Benchmark. For Snappy and Zstd, the respective factors are 4.2× and 2.4×. The difference in compression ratio for the compressed variants of both formats is at most 8%, even though ORC without compression is 28% larger than uncompressed Parquet.

Per-column performance. Table 4 facilitates more low-level insights on how the compression ratio and decompression speed of BtrBlocks compare to Parquet+Zstd. It shows metrics for a random sample of Public BI columns and lists the encoding scheme that BtrBlocks used for the first cascading step of the first block. BtrBlocks outperforms Parquet+Zstd in terms of compression speed, and comes close in terms of compression factor. The table also shows a sample from the first 20 entries of each column [17], which may not be representative of the data distribution in the entire column. This illustrates the necessity of a well-crafted sampling algorithm for deciding on encoding schemes.

In-memory TPC-H decompression throughput. We performed another decompression experiment with data from TPC-H and show the results in Figure 8 (bottom). The average decompression throughput of all schemes is less on TPC-H because it compresses worse. Still, **BtrBlocks** decompresses 2.6×, 3.9× and 4.2× faster than Parquet, Parquet+Snappy and Parquet+Zstd, respectively.

6.7 End-to-End Cloud Cost Evaluation

Is Parquet decompression fast enough? Slow decompression in network scans translates to a higher query execution time and thus higher query costs. Looking at Figure 8, however, every Parquet variant achieves an in-memory decompression throughput of over 50 GB/s. With the

Table 5. S3 Scan Cost on the largest 5 Public BI workbooks

Format	S3 T_u [GB/s]	S3 T_c [Gbit/s]	Scan cost [$]	**Normalized Cost [×]**
BtrBlocks	**174.6**	**86.2**	**0.97**	**1.00**
Parquet	56.1	52.6	2.47	2.61
+Snappy	77.6	33.2	1.74	1.84
+Zstd	78.6	24.8	1.70	1.77

100 Gbit ≙ 12.5 GB/s networking of c5n.18xlarge, it seems like network bandwidth is the bottleneck and scans cannot benefit from faster decompression. This, however, is a false conclusion stemming from the definition of decompression throughput.

Decompression throughput and network bandwidth. Decompression throughput is usually measured using the uncompressed data size, e.g., $T_u = \frac{\text{uncompressed size}}{\text{decompression time}}$. This is the metric that Figure 8 shows and it is the relevant metric for the data consumer. But when loading data over a network, decompression throughput has to be higher than the network throughput *in terms of compressed data size*. Otherwise, the network bandwidth is not yet fully exploited and decompression is CPU bound. We thus introduce another metric for decompression throughput, $T_c = \frac{\text{compressed data size}}{\text{decompression time}}$, essentially dividing T_u by the compression factor. We will see how this impacts the scan cost in our end to end cloud cost evaluation.

Measuring end-to-end cost. Because what matters in the end for analytical processing in the cloud is cost, we explicitly evaluate the cost savings BtrBlocks brings. For scans from S3, this cost consists of two parts:

- We need an EC2 compute instance to load data to, which has an hourly rate of $3.89 in the case of our test instance c5n.18xlarge [4, 18].
- Every 1,000 GET requests to S3 cost $0.0004; the amount of data returned by each request is irrelevant.

Thus, to compute the cost of a scan, it suffices to count the number of requests and measure the scan duration. The S3 performance guidelines recommend fetching 8 MB or 16 MB chunks per request for maximum throughput [5]; we chose 16 MB chunks for this experiment. Consequently, one S3 chunk consists of multiple BtrBlocks blocks that add up to 16 MB or slightly less. Parquet data is generated by Apache Spark, which splits it into multiple files by default. We have no control over the size of these files, but they usually range from 5.5 to 24 MB. Some of the datasets from the Public BI Benchmark are too small to get a useful throughput measurement for, so we exclude tables that have a CSV file size of less than 6 GB.

End-to-end cost test setup. Our benchmark uses the S3 C++ SDK [6] to load compressed chunks of various formats from S3 and then decompresses them in-memory like a query processing engine might. We implement our own memory pool on top of the abstractions provided by the S3 SDK in order to measure the raw decompression speed without the inefficient stream implementations the SDK provides by default. We map threads to chunks returned by S3 one-to-one because this turned out to be the most efficient technique. The requests themselves are issued asynchronously and then added to a global work queue to achieve maximum throughput.

Loading individual columns. OLAP queries rarely read entire tables, but instead select individual columns across one or many tables. Our first experiment thus loads individual columns from S3 and decompresses them. We choose the columns using random queries from the five largest Public BI datasets, e.g., our benchmark only fetches columns that a given query scans. We find

that BtrBlocks scans are 9× cheaper than the compressed Parquet variants and 20× cheaper than uncompressed Parquet, on average. We also measured the cost for loading columns from all 22 TPC-H queries. In TPC-H, Parquet is 5.5×, Parquet with Snappy 3.6× and Parquet with Zstd 2.8× more expensive than BtrBlocks on average.

Cost comparability. However, we do not think this experiment represents the contributions of BtrBlocks particularly well because a different factor causes the high performance difference we measured. Parquet bundles multiple columns into one file and stores column offsets in a metadata footer at the end of the file. Thus, to load a single column in Parquet, a client has to perform three separate but dependent requests to S3: fetch the metadata length, fetch the metadata, fetch the partial file containing the column [54]. The alternative is loading the entire file and then decompressing the column locally, which we often found to be faster. In contrast, the BtrBlocks S3 metadata implementation uses one file per column and bundles metadata for the entire table in a separate file. But metadata handling is not an issue we are trying to address with BtrBlocks; in fact, we argued that metadata is orthogonal and should be handled separately in Section 2. We thus perform a different experiment for comparing the scan cost with Parquet.

Loading entire datasets. Instead of loading individual columns, we now load entire datasets from S3 and measure the combined compute instance and request cost. For this experiment, we can forgo loading metadata and just load whole files instead. The measured difference in cost can thus be entirely attributed to the superior decompression speed and efficiency of BtrBlocks. As with the previous experiment, we are using the five largest datasets from the Public BI Benchmark. We load each dataset 10,000 times and average the measured cost and throughput to get rid of network effects.

Cost of loading full datasets. Table 5 shows that BtrBlocks loads these datasets 2.6× cheaper than uncompressed Parquet and 1.8× cheaper than Parquet with Zstd/Snappy on average. No Parquet-based format can exploit the network bandwidth; BtrBlocks almost does at $T_c = 86$ Gbps, which is close to the throughput our S3 client achieves with uncompressed data at 91 Gbps. This reaffirms the importance of T_c as a measurement of decompression throughput when loading data over the network; Figure 1 further illustrates this point. Considering this benchmark does not include any CPU time for query processing, we can expect the cost difference in an actual OLAP system to be even higher.

6.8 Result Discussion

Is BtrBlocks only fast because of SIMD? Section 5 describes low-level decompression optimizations that BtrBlocks includes, most of which use SIMD and often improve performance substantially. One might deduce that BtrBlocks decompresses so much faster than existing formats solely because of these low-level optimizations, not because of its high-level design. If this were the case, we could simply improve the implementation of Parquet instead of designing a new format. We checked this by implementing scalar versions of every decompression algorithm in BtrBlocks. Running the experiments from Section 6.6 again, in-memory decompression is slowed down by 17%. This, however, is still 2.3× faster than the fastest Parquet variant. We conclude that substantially improving Parquet requires more than low-level optimizations such as SIMD.

Update the standard or create a new format? Yet, improving existing widespread formats such as Parquet is more desirable than creating a new data format: For users, there would be no costly data migration, no breaking changes and an improvement in decompression performance just by updating a library version. Unfortunately, our experiments indicate that low-level improvements are not enough, and integrating larger parts of BtrBlocks – such as new encodings and cascading compression – into Parquet will cause version incompatibilities. Such a "Parquet v3" would not share much with the original besides the name, with no actual benefit to existing users of Parquet.

Instead, we have open-sourced BtrBlocks and hope that compatible improvements will find their way into Parquet, while also building a new format based on BtrBlocks that is independent of Parquet.

7 RELATED WORK

Columnar Compression. There is a large body of work on columnar compression in databases [21, 22, 61]. Below, we discuss a selection that relates most closely to BtrBlocks.

SQL Server. With the introduction of column store indexes, SQL Server offers an optional column-based storage layout [38]. It divides data into aligned rowgroups, each of which contains segments of columns. With column store indexes, SQL Server also adds columnar compression. The system compresses each column segment individually in three steps: (1) encode everything as integers, (2) reorder rows inside each rowgroup and (3) compress each column. During the encoding step, SQL Server translates strings to integers using Dictionary Encoding. In more recent work, it optimizes the resulting dictionaries further by keeping short strings instead of translating them to 32 bit integers [37]. Numeric types are encoded as integers by finding the smallest common exponent in each segment and multiplying with it. For integer types, SQL server strips common leading zeros in each segment and then applies FOR encoding to reduce data range. After encoding, the system reorders rows inside each rowgroup to optimize for encoding using *RLE*. Finally, it compresses either using *RLE* or *Bit-packing*. How exactly SQL Server chooses which scheme to use is not published. From the evaluation using Microsoft-internal datasets, this compression technique achieves a weighted average compression factor of 5.1×.

DB2 BLU. Like SQL Server did with column store indexes, IBM added a column-based storage layout to DB2 with DB2 BLU [53]. Unlike SQL Server, BLU stores multiple columns segments together on a single fixed-size page. Column segments are encoded using the previously mentioned Frequency Encoding. Additionally, each page may be compressed again using local dictionaries and offset-coding based on the local data distribution. As with the compression schemes used in SQL Server, DB2 BLU aims to allow for query processing on compressed data, like early filtering on range queries. However, due to the bitwise encoding schemes used, point access is more involved and requires unpacking tuples first. Like BtrBlocks, DB2 BLU uses bitmaps to indicate `NULL` values.

SIMD decompression and selective scans. There is a large body of work discussing the use of SIMD and SIMD-optimized data layouts to speed up decompression and column scans. Polychroniou et al. [52] implement SIMD-optimized versions of common data structure operations and compare them against their scalar counterparts. Joint work by SAP and Intel focuses on fast predicate evaluation and decompression in column stores using SSE and AVX2 [58, 59]. Vertical BitWeaving [45] and ByteSlice [32] propose separating the bits of multiple values in a radix-like fashion, such that the k-th bits of these values reside adjacently in memory, thus enabling short-circuited predicate evaluation. Motivated by the observation that predicates often act on multiple columns simultaneously, Johnson et al. [35] propose storing multiple columns together in a bank, such that the resulting compressed partial tuples fit into a word. Using a custom-designed algebra on these packed words facilitates bandwidth- and cache-friendly computation.

Compressed data processing in BtrBlocks. Most of these academic papers, as well as SQL Server and DB2 BLU, facilitate some kind of partial query processing directly on the compressed data. This makes sense in proprietary systems where processing and storage are tightly integrated. We believe that open formats, in contrast, should optimize for raw decompression speed first: This way, systems can expect speed improvements without having to build their query processing around a single format. Note that BtrBlocks can, in principle, also support processing compressed data if the used schemes support it.

HyPer Data Blocks. The in-memory HTAP system HyPer introduced *Data Blocks* to reduce the memory footprint of cold data. Because HyPer targets both OLTP and OLAP, Data Blocks has to preserve fast point access [36]. As such, it only uses lightweight encoding schemes that keep the data byte-addressable: *One Value*, *Ordered Dictionary Encoding* and *Truncation*. After splitting the data into blocks, HyPer decides which scheme is optimal based on the statistics collected about that block. Truncation is a specialized version of FOR Encoding where the frame of reference is the *min* value of each block. Ordered Dictionary Encoding is feasible because blocks are immutable and do not need fast updates. HyPer chooses the dictionary code size based on the amount of unique values, and ordering the dictionary allows it to evaluate range predicates on compressed data. To further increase the processing speed on compressed data, every block also contains an SMA (small materialized aggregate) and a lightweight index that improves point-access performance. The authors report compression factors of up to 5×.

SAP BRPFC. With *Block-Based Re-Pair Front-Coding (BRPFC)*, SAP introduced a new compression scheme for string dictionaries [39]. This work is motivated by an internal analysis that showed the string pools required by Dictionary Encoding make up 28% of SAP HANAs total memory footprint. The system already uses block-based *Front-Coding* to compress dictionaries. Given sorted input strings, this encoding replaces the common prefix of subsequent strings with the length of the prefix. For example, `{SIGMM, SIGMOBILE, SIGMOD}` compresses to `{SIGMM, (4)OBILE, (5)D}`. HANA further improves this technique by adding Re-Pair compression, which replaces substrings in the data with shorter codes using a dynamically generated grammar for each block. They apply the resulting algorithm to blocks of data that fit in the cache to increase compression speed. Additionally, the authors designed a SIMD-based decompression algorithm to improve access latency. However, decompression is still too slow for our use case: Based on the reported access latency, one can calculate a sequential decompression throughput of ≤100 MB/s [26]. This decompression performance is not sufficient for our use case, which is why we did not include a similar compression technique in BtrBlocks.

Latency on data lakes. BRPFC optimizes for per-string access latency because this is an important metric in an in-memory database like HANA. As a data format that targets data lakes, BtrBlocks does not profit from this: Access latency matters little when fetching large chunks of data over a high-latency network. We thus chose to optimize throughput and decompression speed instead.

8 CONCLUSION

We introduced BtrBlocks, an open columnar compression format for data lakes. By analyzing the Public BI Benchmark, a collection of real-world datasets, we selected a pool of fast encoding schemes for this use case. Additionally, we introduced *Pseudodecimal Encoding*, a novel compression scheme for floating-point numbers. Using our sample-based compression scheme selection algorithm and our generic framework for cascading compression, we showed that, compared to existing data lake formats, BtrBlocks achieves a high compression factor, competitive compression speed and superior decompression performance. BtrBlocks is open source and available at *https://github.com/maxi-k/btrblocks*.

ACKNOWLEDGMENTS

Funded/Co-funded by the European Union (ERC, CODAC, 101041375). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] October 11, 2022. https://github.com/lemire/FastPFor.
[2] October 11, 2022. https://github.com/cwida/fsst.
[3] October 14, 2022. https://github.com/apache/arrow/blob/883580883aab748fe94336cbed844f09e015178f/cpp/src/parquet/column_writer.cc#L1376.
[4] October 14, 2022. https://aws.amazon.com/ec2/pricing/on-demand/.
[5] October 14, 2022. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance-guidelines.html.
[6] October 14, 2022. https://aws.amazon.com/sdk-for-cpp/.
[7] October 4, 2022. https://github.com/RoaringBitmap/CRoaring.
[8] October 4, 2022. https://orc.apache.org/specification/ORCv1.
[9] October 6, 2022. https://github.com/cwida/public_bi_benchmark/blob/master/benchmark/CommonGovernment/samples/CommonGovernment_1.sample.csv.
[10] October 6, 2022. https://github.com/cwida/public_bi_benchmark/blob/master/benchmark/Generico/samples/Generico_1.sample.csv.
[11] September 20, 2022. https://github.com/google/snappy.
[12] September 20, 2022. https://github.com/facebook/zstd.
[13] September 20, 2022. https://parquet.apache.org/docs/file-format/data-pages/encodings/.
[14] September 20, 2022. https://github.com/lz4/lz4.
[15] September 20, 2022. https://parquet.apache.org/.
[16] September 21, 2022. https://oneapi-src.github.io/oneTBB/.
[17] September 24, 2022. https://github.com/cwida/public_bi_benchmark.
[18] September 24, 2022. https://aws.amazon.com/ec2/instance-types/c5/.
[19] September 27, 2022. https://github.com/cwida/fsst/blob/master/fsst.h#L144.
[20] September 29, 2022. https://arrow.apache.org/docs/cpp/api/utilities.html?highlight=lz4#compression.
[21] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280.
[22] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*. ACM, 671–682.
[23] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216.
[24] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD*. 2205–2217.
[25] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD*. 2326–2339.
[26] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *PVLDB* 13, 11 (2020), 2649–2661.
[27] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. 225–237.
[28] Martin Burtscher and Paruj Ratanaworabhan. 2007. High Throughput Compression of Double-Precision Floating-Point Data. In *DCC*. IEEE Computer Society, 293–302.
[29] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.
[30] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey. In *EDBT*. 72–83.
[31] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44, 3 (2019), 9:1–9:46.
[32] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *SIGMOD Conference*. ACM, 31–46.

[33] Bogdan Ghita, Diego G. Tomé, and Peter A. Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *CIDR*.
[34] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD Conference*. ACM, 1917–1923.
[35] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. 2008. Row-wise parallel predicate evaluation. *Proc. VLDB Endow.* 1, 1 (2008), 622–634.
[36] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326.
[37] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. 2013. Enhancements to SQL server column stores. In *SIGMOD*. 1159–1168.
[38] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL server column store indexes. In *SIGMOD*. 1177–1184.
[39] Robert Lasch, Ismail Oukid, Roman Dementiev, Norman May, Süleyman Sirri Demirsoy, and Kai-Uwe Sattler. 2019. Fast & Strong: The Case of Compressed String Dictionaries on Modern CPUs. In *DaMoN*. 4:1–4:10.
[40] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.
[41] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *PVLDB* 14, 9 (2021), 1606–1612.
[42] Daniel Lemire and Leonid Boytsov. 2012. Decoding billions of integers per second through vectorization. *CoRR* abs/1209.2137 (2012). arXiv:1209.2137 http://arxiv.org/abs/1209.2137
[43] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with Roaring. *CoRR* abs/1603.06549 (2016).
[44] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi Yan Kai. 2017. Roaring Bitmaps: Implementation of an Optimized Software Library. *CoRR* abs/1709.07821 (2017).
[45] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *SIGMOD Conference*. ACM, 289–300.
[46] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *PVLDB* 15, 11 (2022), 3058–3070.
[47] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1 (2010), 330–339.
[48] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *PVLDB* 13, 12 (2020), 3461–3472.
[49] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*. 283–294.
[50] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.
[51] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827.
[52] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *DaMoN*. ACM, 9:1–9:6.
[53] 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. *PVLDB* 6, 11 (2013), 1080–1091.
[54] Alice Rey, Michael Freitag, and Thomas Neumann. 2023. Seamless Integration of Parquet Files into Data Processing. In *BTW (LNI, Vol. P-331)*. Gesellschaft für Informatik e.V., 235–258.
[55] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *PVLDB* 16, 6 (2023), 1413–1425.
[56] 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 *DBTest*. 1:1–1:6.
[57] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *SIGMOD*. 993–1008.
[58] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates. In *ADMS@VLDB*. 1–12.

[59] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (2009), 385–394.
[60] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*.
[61] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. 59.

Received October 2022; revised January 2023; accepted February 2023