The FastLanes File Format

Azim Afroozeh
Centrum Wiskunde & Informatica
The Netherlands
azim@cwi.nl

Peter Boncz
Centrum Wiskunde & Informatica
The Netherlands
boncz@cwi.nl

ABSTRACT

This paper introduces a new open-source big data file format, called FastLanes. It is designed for modern data-parallel execution (SIMD or GPU), and evolves the features of previous data formats such as Parquet, which are the foundation of data lakes, and which increasingly are used in AI pipelines. It does so by avoiding generic compression methods (e.g. Snappy) in favor of lightweight encodings, that are fully data-parallel. To enhance compression ratio, it cascades encodings using a flexible *expression encoding* mechanism. This mechanism also enables multi-column compression (MCC), enhancing compression by exploiting correlations between columns, a long-time weakness of columnar storage. We contribute a 2-phase algorithm to find encodings expressions during compression.

FastLanes also innovates in its API, providing flexible support for *partial* decompression, facilitating engines to execute queries on compressed data. FastLanes is designed for fine-grained access, at the level of small batches rather than rowgroups; in order to limit the decompression memory footprint to fit CPU and GPU caches.

We contribute an open-source implementation of FastLanes in portable (auto-vectorizing) C++. Our evaluation on a corpus of real-world data shows that FastLanes improves compression ratio over Parquet, while strongly accelerating decompression, making it a win-win over the state-of-the-art.

PVLDB Reference Format:
Azim Afroozeh and Peter Boncz. . PVLDB, 18(11): 4629 - 4643, 2025.
doi:10.14778/3749646.3749718

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/cwida/fastlanes-vldb2025.

1 INTRODUCTION

The data formats Apache Parquet and ORC were designed in 2013, and quite similar designs are used in modern analytical systems that have an own storage format, such as DuckDB and Snowflake [26, 83]. Parquet is now the de-facto standard format for data lakes and "lake houses" [13]. However, we argue that changes in hardware and workloads during the past decade call for a re-design.

In the next decade, workloads of analytical data systems and data lakes will increasingly include AI pipelines that perform training or inference [66]. In terms of hardware that runs these workloads, CPUs have become quite diverse (not only x86, but also ARM and RISC-V) and are evolving mostly in novel instructions (SIMD), while AI pipelines increase the importance of GPU- or even TPU-based data processing. In order to efficiently process data on such hardware, algorithms need to harbor *data parallelism*, and specifically need to consist of massive regular computation patterns with absence of data- and control-dependencies. This imposes constraints on what algorithms a data format should employ, e.g. Snappy is the antithesis of a data-parallel algorithm. Current data formats [67, 102] were not designed with this in mind, and struggle to effectively use SIMD and GPUs for decompressing data.

	Parquet	BtrBlocks	FastLanes
Heavy-Weight Compression methods	yes	no	no
data-parallel: SIMD/GPU-friendly	no	no	yes
cascading Light-Weight Compression	no	yes	yes
Multi-Column Compression methods	no	no	yes
			compression methods
access granularity	1MB chunk	64K rowgroup	1K vector
can return compressed vectors	no	no	yes
			read access API

Figure 1: Feature comparison of big data file formats. BtrBlocks introduced cascading Light-Weight Compression to avoid the Heavy-Weight Compression (e.g. Zstd) used in e.g. Parquet, but its encodings are not data-parallel (SIMD/GPU-friendly). FastLanes is fully data-parallel, can do vector-at-a-time decompression (small footprint), introduces Multi-Column Compression & allows access to compressed vectors.

Further, for efficient query processing *after* decompression, data needs to stay in SIMD-friendly representations during execution. Modern query engines such as DuckDB, Velox and Procella therefore added *compressed execution* capabilities, augmenting vectorized query execution with new compressed vector classes, such as constant-vectors, dictionary-vectors and FSST-vectors [19, 78, 83]. This trend calls for innovation in data format APIs, to directly deliver compressed vectors from a table scan on request of an engine that can handle this, by only partially decompressing data.

This paper describes the **FastLanes** data format, marking its v0.1 release in open source. It is designed to efficiently support modern analytics+AI workloads. Its main contribution is a novel Expression Encoding mechanism, supported by an intricate segmented block layout, enabling flexible cascaded encodings and multi-column compression. This allows it to achieve excellent compression ratios while using only simple and ultra-fast data-parallel encodings.

Outline. In Section 1.1, we describe our core ideas and in Section 1.2 outline the FastLanes design. Section 2 explains Expression Encoding. Our novel segmented layout is detailed in Section 3. We evaluate vs. Parquet, BtrBlocks and DuckDB in Section 4, showing that FastLanes achieves state-of-the-art compression ratios at higher decompression speed. Additional design decisions and related work are covered in resp. Section 6 and Section 5. We conclude in Section 7 and outline future work in Section 8.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749718

1.1 Design Ideas

A columnar layout typically reduces data entropy over row storage, as it concentrates data belonging to the same distribution, making it more compressible. Heavy-Weight Compression **(HWC)** schemes, also referred to as {general-purpose, block-based, type-agnostic} compression schemes, such as `Snappy` [40] and `Zstd` [22], are used by default in Parquet to compress column chunks. While such compression libraries provide good compression ratios, they are typically CPU-intensive, making decompression considerably slower than accessing uncompressed data [2, 107]. In contrast, Light-Weight Compression **(LWC)** schemes such as `FFOR [6]`, `Delta [6]`, `DICT [6]`, `ALP [8]`, `FastLanes-RLE [6]`, and `FSST [16]` are specifically designed for certain data types and encode data by capturing simple compression patterns. Unlike HWC schemes, it is possible to fully data-parallelize LWC decompression, which makes LWC profit from wide SIMD CPU capabilities, accelerating them up to 64x, which can make accessing compressed data even *faster* than uncompressed data [6]. Data-parallelism also helps GPU decoding performance, as it provides independent work and interleaved memory access for all threads in a GPU warp [7]. However, when considering compression ratio, rather than speed, a micro-benchmark on the `Public BI` dataset shows that adding HWC schemes in ORC, on top of LWCs, improves the compression ratio 3x [4] (`ORC` vs. `ORC+Snappy`). This means that using HWCs is necessary in current big data formats.

Cascading LWC schemes. To achieve the same compression ratio as HWCs while maintaining the speed of LWCs, Cascaded Compression, also known as {recursive, composable} compression [3, 5, 33], has been implemented in BtrBlocks [53]. It combines multiple LWCs to capture a wider range of data patterns. To illustrate how this approach can improve the compression ratio, consider the array: `{'Cascading','Cascading','Cascading','Cascading','Cascading', 'Cascading','Compression','Compression'}`. This array exhibits two patterns: repeated values and low entropy, which are well-suited for {`RLE`, `DICT`} encodings. However, applying only `DICT` or `RLE` captures just one of these patterns. By first applying `DICT`, the array is transformed into codes `{0,0,0,0,0,0,1,1}` and dictionary `{'Cascading', 'Compression'}`. Then, applying `RLE` turns the codes into `{{0,6},{1,2}}`, achieving better compression.

Multi-Column Compression (MCC) is a new category of compression schemes that takes multiple columns into account, with the key idea that correlation between two columns can be used to infer one column from another, thereby achieving a higher compression ratio [38, 68, 70, 91]. Note that compressed columnar formats store columns independently of each other, missing out on this opportunity – leading to the phenomenon that some tables with strongly correlated columns can be more compact in the row-oriented RC format than in Parquet. A simple example is when two columns are completely identical. Our MCC also includes schemes that split one column into multiple sub-columns, which can be encoded individually. For example, string values composed of names and numbers like "Compression101" could be separated into two columns: one for strings and one for integers. This enables further compression e.g., by applying integer-based specific encoding (such as `DELTA` or `FOR`) on the suffix [37, 50].

Vectorized decoding carries over the efficient properties of vectorized execution [17] when applied to decoding compressed data. When a vectorized table scan decompresses a vector, the (compact) compressed data in RAM is decompressed into an uncompressed vector, which is a small array of e.g., 1024 values that fits into the CPU's L1 cache and is immediately processed by the query pipeline, typically without spilling to RAM. As such, decompression occurs only when the data arrives in the CPU for query processing, keeping it small while in transport, reducing memory, network, and disk bandwidth consumption [107]. Reading while decompressing FastLanes data was found faster than reading uncompressed data (memcpy) [3], because of the reduced bandwidth needs plus ultra-fast auto-vectorized decoding kernels (e.g., decoding 60 values in 1 CPU cycle). The BtrBlocks format not only relies on older encodings that are not data-parallel, but also performs decompression on the full rowgroup level, imposing a large memory footprint. However, allowing fine-grained read access is to avoid overwhelming L1 CPU caches, and even more pressingly, GPU cache and register space [7].

Compressed Execution LWCs (encodings) are more than just a technique to compress data; they capture patterns that can also be used later to optimize query execution on this data. The simplest example is constant encoding, which can tell the query engine that all operations on this column could to be done once instead of on all the values in the column. Modern systems like Procella [20], Velox [78], and DuckDB [83] support *compressed vectors*, where data is both randomly accessible yet still might be encoded in e.g., `DICT`, `FOR` or `FSST`. For example, the `l_tax` column in the TPC-H benchmark is a `decimal(18,2)`, which many systems would implement as a `int64` because it can represent numbers of up to 18 digits, where internally the decimals are multiplied by 100 (due to decimal scale 2). Now suppose, the actual data just contains values between `0.01` and `0.08` (i.e., integers 1-8). Applying LWC, would typically compress such a column using `FOR` and bit-packing (`BP`) in 3 bits per value (`FOR` base 1; and differences 0-7). Whereas legacy systems would decompress this column in their scan into its SQL type `decimal(18,2)` (i.e., `int64`), a system like DuckDB can decompress it into a `int8` (byte). This allows to use 8x thinner SIMD lanes for decompression, accelerating decoding 8x; and also creates better chances for exploiting SIMD in the query e.g. for comparisons or subsequent arithmetic operations, and further, reduces memory pressure e.g. when the column is materialized in a join hash table.

1.2 Designing The FastLanes File Format

FastLanes is a project initiated at CWI, designed as a foundation for next-generation big data formats. In the first paper on FastLanes [6], we focused on significantly improving data decoding performance over the state-of-the-art by introducing a 1024-bit interleaved and Unified Transposed Layout, enabling data-parallel decoding even with scalar code. In the second paper, we demonstrated that data-parallelized layouts are essential to fully exploit GPU parallelism [7]. Additionally, we designed and implemented ALP [8], a new vectorized and data-parallel encoding for floating-point data.

In this paper, we introduce *Expression Encoding* and design, implement and evaluate the **FastLanes file format**, while addressing the following research questions:

(1) how can Expression Encoding be supported in a file format?

(2) what is a good pool of encodings to use in expressions?
(3) what encoding algorithm can find good expressions quickly?
(4) what forms of Multi-Column Compression can be integrated in Expression Encoding, and what does this add?
(5) how effective is FastLanes compared to the state of the art in terms of compression ratio and en/decoding speed?

For the FastLanes file format we designed and implemented a novel *Segmented Page Layout*, that allows to store data encoded with any arbitrary nested encoding expressions, while providing an efficient vectorized API to decode the data. This API allows for the fetching and decoding of arbitrary vectors in case of random access or sequences of vectors in cases of full vectorized scans. The segmented layout stores similar parts of encoded data, encoded by an expression, in a single location, along with additional metadata for the encoded data; essentially pointers to this encoded data at the granularity of a vector. Having all encoded data of the same type in one place makes it ideal for recursive compression due to shared types and data semantics. This also enables fine-grained access to the encoded data, at the vector granularity. As a consequence, decoders can perform advanced predicate pushdown, e.g., the base of FOR encoding, representing minimum values, can be evaluated first to skip individual vectors in range queries. When reading, data in the expression chain is decoded-bottom-up, but not necessarily fully until the end of the chain. This allows modern query engines to choose to partially decode data, yielding *compressed vectors* that query engines can exploit for compressed execution.

Our main contributions are:

- An open-source, high-quality implementation of FastLanes (v0.1) in C++ with absolutely zero code dependencies.
- The design of Expression Encoding, a novel compression model providing a unified approach to cascaded encoding, MCC, compressed execution and vectorized decoding.
- A novel segmented layout to store any arbitrary expression-encoded data, enabling the query engine to interpret underlying encoded data and apply further optimizations.
- The design of a Two-phase Expression Detection algorithm that identifies the optimal expression among a wide pool of possible encoding expressions.
- An evaluation against other file formats on the Public BI dataset, demonstrating that FastLanes achieves faster decompression and better compression ratios.

2 EXPRESSION ENCODING

In this section we touch on the first four research questions, which all revolve around the design of our Expression Encoding framework. We first explain the pool of operators that serve as the building blocks of Expression Encoding, including operators that perform Multi-Column Compression. We then describe how to serialize an expression and its operators within a file format and how to interpret a serialized expression during decoding at execution time. Finally, we outline the process for identifying a good encoding expression from a potentially infinite domain space, as operators can be combined in any order.

Table 1: FastLanes operators, and the encoded data held by each. An operator is a vector of 1024 values in an executable encoded layout. Encoding operators can be exploited for compressed execution. Take FFOR as an example, which keeps the base separated from the bit-packed data. In the case of simple query predicates such as addition, decoding can be delayed, and the value can be added only to the base. Multiple of these operators can be combined in a chain, forming Encoding Expressions. For instance, FFOR can be combined with DICT to build dictionary encoding with bit-packed codes.

ID	Operator	Encoded Layout
0	FFOR	Bitpacked-data, Base, Bit-width
1	PATCH	Data, Exceptions, Exception Positions
2	DELTA	Deltas, Bases
3	ALP	Data
4	ALP_RD	Left side data, Right side data
6	DICT	Dictionary, Codes
7	Transpose	Transposed Data
8	Cast	Data
9	FRLE	RLE-values, Length
10	CROSS RLE	RLE-values, Indexes
11	FSST	Symbol Table, Compressed Strings
12	FSST12	Symbol Table, Compressed Strings
13	CONSTANT	Single Value
14	EQUALITY	Data or Pointer to data
15	EXTERNAL DICT	Dictionary, Pointer to another column

2.1 Expression Operators

Expression Encoding is similar to white-box compression models [37] or cascaded encodings [53] in that it combines different primitives to achieve better compression. However, operators in FastLanes Expression Encoding are neither simple functions with single tasks, as in white-box compression models, nor entire LWCs, as in cascaded encoding. An operator in FastLanes is a data structure that stores data in a compressed format and transforms it to the next format during decoding. These transformations come from breaking down LWCs into parts that are both reusable and efficient.

For example, the DICT operator in FastLanes maintains a pointer to a dictionary and a vector of associated codes, replacing the codes with actual values only if needed. Furthermore, to support vectorized execution and leverage data-parallel layouts, such as a Unified Transposed Layout or 1024-interleaved layout, each operator holds only 1024 values at a time (a vector). All operations on data are performed in a tight loop over these 1024 values, with consistent work patterns that enable compilers to auto-vectorize [6].

The operators used in FastLanes are summarized in Table 1 and are explained as follows:

FFOR. The FFOR operator stores data in a FOR vector, consisting of a base and a vector of bit-packed data – it is Fused with bit-packing. This fusion eliminates a SIMD store and load instruction between the addition resp. subtraction and bit-[unp]packing loop, improving performance. Unlike BtrBlocks and DuckDB, we use only FFOR and avoid a separate bit-packing operator, since the performance of FFOR decoding is almost identical to bit-unpacking.

PATCH. The PATCH operator, inspired by Patched Encoding [107], addresses the vulnerability of encodings such as FFOR and ALP to

outliers, by keeping outliers separate from the main vector and reintegrating them during full decompression. We do not fuse patching with encoding operators like FFOR, as having a separate PATCH operator allows us to apply the patching mechanism to enhance any other LWC or operator. For example, if a vector is 95 percent constant, we can still use constant encoding while storing exceptions separately. In FastLanes, we implement only one variation of the possible options for patching, namely {LinkedList (LL_PATCH) [107], SelectionVector (SL_PATCH) [3],Bitmap (BM_PATCH) [3]}: SelectionVector patching, as SelectionVector is the only patching technique capable of being data-parallelized on a GPU [46]. SelectionVector patching uses a separate array to store the positions of exceptions.

DELTA. The DELTA operator works only on integers and stores delta values in the Unified Transposed Layout [6], which breaks data dependencies among values, accelerating the decoding of delta encoding using scalar code that auto-vectorizes. The implementation comes from our previous work on data-parallelized encoding [6]. Unlike BtrBlocks, which completely avoids delta encoding, we argue that delta encoding is crucial for future file formats, particularly for encoding (mostly) sorted data and, more importantly, for encoding offset arrays that are always sorted and are necessary to represent any variable-size data (strings).

ALP. The ALP operator, used specifically for the DOUBLE and FLOAT data type, keeps data in an ALP-encoded format and utilizes our own ALP [8], which significantly improves previous DOUBLE schemes in both speed and compression ratio. ALP is designed for *vectorized execution* and uses an enhanced version of PseudoDecimals [53] to encode doubles as integers if they originated as decimals. Its high speed is due to our implementation in scalar code that auto-vectorizes, using building blocks provided by our FastLanes library [6], and an efficient two-stage compression algorithm that first samples rowgroups and then vectors.

ALP_RD. is used to compress high precision values, by separating the front bits of a float/double from the rest. These front bits are then compressed using primitives designed for the INTEGER data type and, during decoding, are reassembled with the rest of the double using the Glue operator. The difference from the original ALP paper is that we split ALP in its two schemes in FastLanes. Since both schemes map floating-point values to integers, this allows to cascade compression of these integers with Expression Encoding.

Glue. The Glue operator combines two sources of bit-packed data, used to merge the front bits and tail bits in ALP_RD encoding or in one-to-many mappings from MCC schemes.

DICT. The DICT operator stores data in a dictionary-encoded format, consisting of a reference to a dictionary and a vector of codes. We support compressed dictionaries, using either Cast and FSST; because dictionaries must allow random-access (note that e.g., ALP and FFOR store data bit-packed, which does not allow random-access). We chose this approach because otherwise dictionary decoding would become rather block-based: access to dictionary-encoded data would then require to fully decode the dictionary first.

We also support a special **Shuffle Dictionary**, used only for fixed-size data types. It contains the eight most repeated values and uses the SIMD shuffle instruction for decoding, as the dictionary can be loaded into a single register. This dictionary is now only used to encode front bits in ALP_RD.

EXTERNAL-DICT. This operator enables us to use "external codes", i.e. the codes from a different column, with a different dictionary. This is useful to support column correlations with a one-to-one mapping, where the codes of the two columns are the same, but their dictionaries are different.[1]

Transpose. The Transpose operator is applied only during encoding, so the decoded data remains in the Unified Transposed Layout (UTL) after decoding. FastLanes provides a shareable *selection vector*: an array of 1024 integers containing the permutation of the UTL, which vectorized query engines can put in front of vectors decoded by FastLanes to recover the original ingested tuple order. The FastLanes decoder can also be requested to restore this order, performing a gather operation on this selection vector.

Cast. The Cast operator keeps values of a column in a different type from what is specified in the schema, to a type which simplifies encoding and query execution. We employ Cast in three scenarios: STRING to INTEGER, allowing query engines to benefit from the SIMD-friendly, fixed-size properties of integers; DOUBLE to INTEGER, enabling the use of the richer INTEGER encoding pool and allowing query engines to operate on integers instead of floating-point data types; and INTEGER to a narrower INTEGER type (e.g., 64-bit to 8-bit). The Cast is a useful end-point for compressed execution. **RLE.** The

RLE operator stores data in the FastLanes-RLE [6] compressed format, which consists of two vectors: one for repeated values and another for indexes pointing to these repeated values. For full decoding, the RLE values are placed in their correct positions using the indexes. Note that FastLanes-RLE maps RLE to dictionary encoding and applies delta encoding to the indexes. This enables the use of a Unified Transposed Layout to break data dependencies among values, accelerating the decoding of RLE encoding with scalar code that auto-vectorizes.

FSST. The FSST operator compresses a vector of STRING data using FSST [16], a lightweight compression scheme with decompression and compression speeds comparable to, or better than, the best speed-optimized compression methods, such as LZ4. FSST uses a static symbol table (stored in the rowgroup header) that enables random access to individual compressed strings, allowing for query processing directly on compressed data.

FSST12 is an alternative version of FSST<200c> that uses 12-bit instead of 8-bit codes [25], allowing it to encode up to 4,096 symbols (each up to 8 bytes long). The larger dictionary allows FSST12 to obtain better compression ratios than FSST on distributions with more entropy; but comes at the cost of a large CPU cache footprint. For instance, JSON and XML benefit more from FSST12.

Cross RLE. The motivation behind this operator is that our data-parallel RLE is very fast but introduces a 128-byte overhead per vector. For a rowgroup of size 64×1024 with very few RLE values, this overhead becomes significant (8KB). To address this issue, we

[1] We also tried the opposite idea: sharing a dictionary between columns with different codes – however in our tests this did not improve compression ratio significantly.

introduce the Cross RLE operator, which applies classical run-length encoding across an entire rowgroup. The main challenge for Cross RLE is efficiently supporting vectorized decoding on the Unified Transposed Layout. To overcome this, we implement the decoding in two steps: first, we traverse the RLE lengths to identify the initial value belonging to the vector currently being decoded, and then we proceed with standard RLE decoding. An additional mapping is performed to correctly decompress into unified transposed layout, but this step is only needed at the boundaries of RLE stretches, adding low overhead.

2.2 FastLanes Expression Notation

To store and represent expressions we use a modified form of *Reverse Polish Notation (RPN)*, that separates operators and operands into two distinct RPN-style (postfix order) sub-expressions:

(1) **Operators**: Stored as integers, with each operator assigned a unique ID.
(2) **Operands**: Stored as integers representing either a column or a segment within a data page in *FastLanes* (Segments are discussed in greater detail in Section 3).

Whereas standard RPN requires (string) parsing to tokenize operators and operands, our approach directly stores operators and operands as integers. This design aims to minimize the overhead of interpretation at runtime, while also using little space for expressions. Each operator is uniquely identified based on its type. For example, the `FFOR` operator has distinct IDs for each data type it supports, further reducing the need to interpret the operator's data type. Thus, `FFOR_UINT8` is the version of `FFOR` that operates on 8-bit data, different from `FFOR_UINT16`, for 16-bit data.

Decoding interpretation consists of initializing a chain of physical expressions, by reading the operator and operand arrays from a column descriptor inside the rowgroup file-footer. These physical expressions are initialized by (i) looking up function pointers from operand IDs, and (ii) binding parameters by looking up values and offsets in the column descriptor, which contains encoding parameters such as e.g. the bit-width for bit[un]packing in `FFOR`, as well as segment descriptions that point to raw bytes in the rowgroup. Execution of [en/de]coding then calls these functions in the physical expressions one after the other.

2.3 Expression Detection

Expression Encoding enables a file format to encode data using any combination of operators. This flexibility introduces a challenge in identifying suitable expressions that achieve both fast decompression and high compression ratios for a given table, as the search space is infinite. We address this challenge with a two-phase approach for expression selection: a *rule-based* phase for detecting relationships between columns and determining the appropriate type for each column, followed by a *sample-based* encoding phase that selects an expression from a predetermined pool of expressions.

Rule-Based Operator Selector. The process of operator selection or expression creation begins by applying rules in the order they are defined. The FastLanes v0.1 rule set consists of the following:

1. Constant: We first identify constant columns where all values are identical. These columns are represented by an expression with only one operator, `CONSTANT`. The constant value is not stored directly; instead, the Min-Max information in the rowgroup footer is used to retain this value. This decision allows the reader to use this column in a query without fetching any data.

2. Equality: We check for equality columns where the values in two columns are (almost) completely identical row-by-row. In this case, the second column is encoded with an expression consisting of only one operator, `EQUAL`, and as operand a {`column-id`}.

3. String as Numerical: It is not uncommon for database users to select a string type as a fallback data type for a column that contains mostly numerical data [36]. Converting these strings to numerical types, such as integers or doubles, improves compression efficiency, as numerical data compresses better than strings (as shown in Section 4), and simplifies processing since numerical data is fixed-size, making it suited for SIMD instructions. This rule first detects such columns and then selects the appropriate numerical type. A `CAST` operator is added to the expression to retrieve the original type if necessary. We currently only apply this rule when **all** values in a rowgroup are numerical, though in the future we

Expression (where $X \in \{08, 16, 32\}$)	Count	Popularity (%)
string columns		
CROSS_RLE_STR	210	9.17%
FSST_DICT_STR_FFOR_SLPATCH_UX	191	8.34%
FSST_DICT_STR_FFOR_UX	166	7.25%
CONSTANT_STR	81	3.54%
EXTERNAL_FSST_DICT_STR_UX	41	1.79%
FSST_DELTA_SLPATCH	33	1.44%
FSST_DELTA	21	0.92%
FSST12_DICT_STR_FFOR_SLPATCH_UX	8	0.35%
FSST12_DELTA_SLPATCH	7	0.31%
RLE_STR_SLPATCH_UX	2	0.09%
FSST12_DELTA	2	0.09%
RLE_STR_UX	1	0.04%
numeric columns		
CONSTANT_INTEGER	334	14.59%
FFOR_SLPATCH_INTEGER	227	9.92%
DICT_INTEGER_FFOR_SLPATCH_UX	210	9.17%
DICT_INTEGER_FFOR_UX	190	8.30%
CROSS_RLE_INTEGER	121	5.29%
FFOR_INTEGER	47	2.05%
EXTERNAL_DICT_INTEGER_UX	24	1.05%
RLE_INTEGER_UX	15	0.66%
RLE_INTEGER_SLPATCH_UX	3	0.13%
floating-point columns		
ALP_DBL	87	3.80%
DICT_DBL_FFOR_SLPATCH_UX	38	1.66%
RLE_DBL_UX	30	1.31%
DICT_DBL_FFOR_UX	28	1.22%
CONSTANT_DBL	18	0.79%
ALP_RD_DBL	17	0.74%
EXTERNAL_DICT_DBL_UX	12	0.52%
CROSS_RLE_DBL	2	0.09%
RLE_DBL_SLPATCH_UX	1	0.04%
correlated columns		
EQUALITY	56	2.45%

Table 2: The FastLanes v0.1 Expression Pool: Sorted by Category & Popularity in being chosen in encoding the Public BI benchmark. For string columns, run-length encoding with long stretches dominates (`CROSS_RLE_STR`), and second most effective are FSST-compressed dictionary encoding with exceptions (`FSST_DICT_STR_FFOR_SLPATCH_UX`). This pool was chosen by exhaustive testing of a wide spectrum of expressions encodings on Public BI, and retaining the winners.

could also use the exception operator to also leverage this encoding when only the majority of string values are numerical.

4. Double as Integer: Similar to "String as Numerical," this rule aims to select a more efficient data type when possible, particularly for double columns that consistently have a zero after the decimal point. A `CAST` operator is added to the expression to retrieve the original type if necessary.

5. Narrower Types for Integer: Similar to "String as Numerical" and "Double as Integer," this rule aims to select a more efficient data type when possible, particularly a narrower integer data type. The narrowest integer type is determined based on the number of bits required to represent the maximum value. A `CAST` operator is added to the expression to retrieve the original type if necessary.

6. One-to-One Map: In a one-to-one correlation, a specific value "X" in one column is always associated with a single specific value "Y" in another column. In this case, the first column proceeds to the second phase to determine the best expression, with the caveat that only expressions with a dictionary as the root are considered. For the other column, we store only a reference to the dictionary column, and the expression selection stops here by choosing `EXTERNAL_DICTIONARY`.

Sampling-Based Encoding. After the Rule-Based Operator Selector, three categories of columns – constant, equal, and one-to-one map – are removed from the pipeline for choosing the optimal expression. For the remaining columns, we use a sampling-and-try approach to select the best encoding expression from a limited predefined **expression pool**, shown in Table 2. This pool consists of expressions we derived from trying a large set of combinations of encodings on the Public BI datasets; where we kept those encoding expressions that ended up being the best for some column.

We call our sampling method *three-way*: FastLanes simply takes the first, last and middle vector (each of 1024 values) in the rowgroup and tries compressing this limited data with all encoding expressions in our pool for that datatype. The key intuition behind this sampling strategy is that many tables exhibit locality, while gradually changing the data distribution throughout the rowgroup. Figure 2 shows a benchmark of compression ratio achieved on the Public BI dataset using two sampling strategies and various sample sizes; where 100% compression ratio was achieved by choosing the best encoding expression when using all *all* vectors of the rowgroup (i.e., no sampling). The sequential strategy uses a front-biased strategy, whereas three-way applies a breadth-first binary-search exploration: after sampling the first and last vector, it incrementally probes the middle of the largest unexplored space. The sequential strategy uses vectors with indices 0, 1, 2, 3, 4, ..., 63 to construct samples of ever larger sizes, whereas three-way uses the order {0, 63, 32, 16, 48, ..., 1, 62}. We see that this strategy, after just three vectors (hence: three-way), achieves more than 99% accuracy in terms of compression ratio, which is why we settled for this.

3 FASTLANES FILE FORMAT

The FastLanes file format is a novel columnar format designed to store expression-encoded data, enabling query execution engines to efficiently access individual vector data efficiently. In this section, we come back at the first research question, completing our description of the FastLanes file layout; by outlining the design of its meta-data, that allows to read encoding expressions from the footer and find for an individual column and vector the input data for these expressions, which get stored in multiple adjacent *segments*.

We first explain the file format from a high-level perspective, beginning with the rowgroup, then moving to the column chunk, and finally delving down to the segment, the fundamental building block of the FastLanes file format. We also demonstrate how segments are used to store the encoded data of an expression. Additionally, we provide a detailed example of how a table is stored.

File Format Overview. The FastLanes file format consists of two main components: the footer and the data. The footer, stored in FlatBuffers [41], contains all the necessary information to access, decode, and decrypt the data, along with statistics such as Min-Max values. The data itself is stored in binary format after being expression-encoded. We propose storing the footer metadata separately from the data —for example, in different files or objects in cloud storage —so that query engines can process metadata first, possibly from a cache or catalog that consolidates metadata for many rowgroups, enabling optimizations like projection pushdown and zone map filtering that avoid accessing data files unnecessarily.

Rowgroup. FastLanes first divides a table horizontally into smaller mini-tables called *rowgroups*. Each rowgroup stores records using the PAX layout [9], which keeps the attribute values of each record together in the same file, while the attributes themselves are stored in a DSM (columnar) layout [24, 108]. All decisions in FastLanes, such as expression detection, are made on a per-rowgroup basis, allowing for more fine-grained tuning and adaptability to data, rather than applying a single expression to an entire column. Additionally, we store statistics for each rowgroup, such as Min and Max values, enabling the ability to skip entire rowgroups for range queries. The size of a rowgroup in FastLanes is a fixed number of records, similar to ORC, and is always a multiple of 1024. This design ensures compatibility with data-parallel encodings [6], as these encodings require 1024-value batches to fully leverage SIMD registers or all threads within a GPU warp [7]. In contrast, Parquet uses fixed physical sizes for rowgroups, resulting in a variable number of records.

Rowgroup Descriptor. Each rowgroup in the FastLanes file format is associated with a descriptor in the footer. This descriptor includes the size of the rowgroup (in terms of the number of vectors), along with the size and offset specifying where the rowgroup starts in the binary data and the number of subsequent bytes it occupies. The descriptor enables the query engine to fetch only the relevant bytes

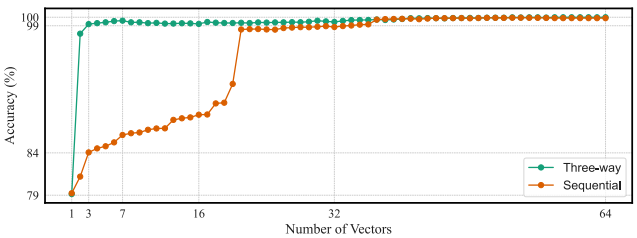

Figure 2: Compression ratio accuracies of Sequential and Three-Way sampling methods. Three-way sampling, vectors at positions 0, 32, and 64 achieve more than 99% accuracy.

```
{   "Rowgroup size in terms of number of vectors": 2,
    "Rowgroup binary size": 26,
    "Rowgroup offset within binary file": 0,
    "Rowgroup ID": "0",
    "Column Descriptors": ["Explained below"]        }
```


Figure 3: Row-group Descriptor in JSON format for the first row-group in Table 7, stored separately from the row-group binary data. This row-group contains two vectors and is located at an offset of 0 bytes in the binary FastLanes file format, with a size of 26 bytes. To retrieve this row-group, 26 bytes starting from offset 0 need to be loaded.


```
{   "type": "STRING",
    "Column offset": 0,
    "Column binary size": 23,
    "Expression": "DICT_FFOR_UINT8"],
    "Column Index": 0,
    "Segments Descriptors": ["Explained below"],     }
```


Figure 4: Column Descriptor for the first column in Table 7, stored within the row-group descriptor depicted above. The Expression for decoding and encoding code ("DICT_FFOR_UINT8") gets mapped to an array of operators and an array of operands, a form a Reverse Polish Notation.


```
{   "Entrypoint offset": 16,
    "Entrypoint binary size": 2,
    "Data offset": 18,
    "Data binary size": 2                            }
```


Figure 5: Segment Descriptor for the segment that stores bases for FFOR. The entry point array is of size 2, as each entry point is a 8-bit unsigned integer. There are two entry points needed, since the row-group size is 2 vectors. The size of the data is 2 bytes: 1 byte for each 8-bits base (1 byte).

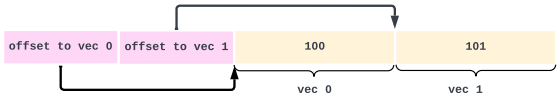

Figure 6: Example segment storing the bases of the `FFOR` expression of the `ID` column with a row-group size of 2 vectors. The segment holds bases for 2 vectors {100, 101}, with an entry point array at the start that points to the base of each vector to enable vectorized decoding. Storing bases in one location allows further compression of these bases, e.g., with `FFOR`. Bases can also be exploited in queries (e.g., ID>101) to skip vectors to provide per-vector min-max stats.

from the binary file, skipping unnecessary rowgroups through zonemap filtering. Additionally, it contains an array of column descriptors, which are explained later. An example of a rowgroup descriptor is shown in Figure 3.

Column Chunk. Within each rowgroup, there is exactly one column chunk per column, storing the data of that column in a columnar format after being expression-encoded. By keeping all data of a column in a contiguous location, this setup enables query engines to perform projection pushdown, allowing them to load only the columns relevant to the query instead of loading all columns.

Column Chunk Descriptor. For every ColumnChunk in a FastLanes rowgroup, there is a corresponding descriptor in the footer. This descriptor includes the type defined by the schema, segment descriptors (explained later), the size and offset indicating where the column starts in the binary data, and the number of subsequent bytes belonging to the column. These details enable the query engine to access relevant columns while skipping unnecessary ones, an optimization known as projection pushdown. The descriptor also includes the column index and statistics, such as the maximum value. An example of a column descriptor is shown in Figure 4.

Segment. A segment stores encoded data of the same nature—data that has the same role in encoding and shares the same type—resulting from encoding a column chunk (multiple vectors) through the expression encoder. It enables fine-grained access to this encoded data (at the granularity of a vector at a time, 1024 values) to support vectorized decoding. Each segment achieves this by storing an additional *entry points* array alongside the data itself, which keeps track of the offset to the start of the data for each vector. Note that the segment stores data from the assigned source consecutively after this entry points array.

Segment Descriptor. For each segment in the file format, there will be a descriptor in the file footer containing two key pieces of information: the entry point offset and size, and the data offset and size. These fields determine the exact location and extent of each segment within the binary data file. An example of segment descriptor is shown in the middle of Figure 5

4 EVALUATION

Hardware. We conducted all experiments on an EC2 instance `i4i_4xlarge`, with Intel Xeon (Ice Lake) CPU, 16 vCPUs and 128 GiB RAM. FastLanes is portable accross multiple operating systems and compilers, and we have previously evaluated its encodings also on Apple and Graviton ARM hardware [6]; however, BtrBlocks depends on x86 intrinsics, which is why we chose this platform.

Data Formats. FastLanes v0.1 is released under an MIT license in our GitHub repository[2]. All experiments are released separately in a dedicated repository[3]. For Parquet, there are several open-source implementations; we use the implementation in DuckDB v1.2 [31], as it employs the latest Parquet encodings [32] and is widely used for writing Parquet files. We compare two variants of Parquet: `Parquet+Snappy`, widely used in practice, and `Parquet+Zstd`, which offers the best compression ratio. For BtrBlocks, we use the original implementation provided by the authors of BtrBlocks [52], run with its default settings at cascading level 2.

We also evaluate DuckDB's native format. Note that DuckDB does not provide any API to directly determine the storage size occupied by a table, making it challenging to accurately measure DuckDB's compression performance. We replicate each sample dataset until the number of samples reaches at least 10 and is a multiple of 1024×120, as DuckDB begins compressing data only when a rowgroup size reaches 1024×120. This setup ensures that the resulting files are sufficiently large, minimizing inaccuracies caused

[2]https://github.com/cwida/FastLanes
[3]https://github.com/cwida/fastlanes-vldb2025

Rowgroup N.	Vector N.	RowID N.	Name	Institute	ID	Alias
0	0	0	Alice	CWI	100	Alice
0	0	1	Alice	CWI	100	Alice
0	0	2	Alice	CWI	100	Alice
0	0	3	Alice	CWI	100	Alice
0	0	4	Bob	CWI	101	Bob
0	0	5	Bob	CWI	101	Bob
0	0	6	Bob	CWI	101	Bob
0	0	7	Bob	CWI	101	Bob
0	1	8	Bob	CWI	101	Bob
0	1	9	Bob	CWI	101	Bob
0	1	10	Bob	CWI	101	Bob
0	1	11	Bob	CWI	101	Bob
0	1	12	Alice	CWI	100	Alice
0	1	13	Alice	CWI	100	Alice
0	1	14	Alice	CWI	100	Alice
0	1	15	Alice	CWI	100	Alice

A table with columns Name, Institute, and ID, and gray inexistent columns including rowgroup, vector, and rowId, indicating which row belongs to which rowgroup and vector. We assign a color to each column, which is also applied to the raw bytes in the FastLanes binary data, depicted at the bottom.

Col Name	Name	Institute	ID	Alias
Type	String	String	Integer	String
Offset	0	23	23	25
Size	23	0	2	0
Expression	DICTIONARY_FFOR_UINT8	CONSTANT	EXTERNAL DICT	EQUALITY
Metadata	---	CWI	---	---
Segments	5	0	0	1

[0, 1]	[1, 8]
[9, 1]	[10, 2]
[12, 2]	[14, 2]
[16, 2]	[18, 2]
[20, 2]	[22, 2]

[24, 1]	[25, 2]

The FastLanes file format footer with required fields: Col Name, Type, Offset (representing the binary offset of this column in the row group), Size (allowing computation of the end of the column chunk), Expression (specifying the encoding expression of this column), Metadata, and Segments—an array of two byte ranges [offset, size]. The first range specifies where the segment 'entry point' array (with one entry point per vector) starts within this column chunk and its size, while the second range specifies where the segment data starts and its size.

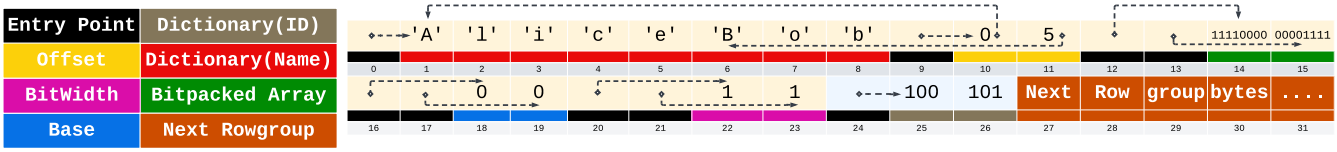

Figure 7: FastLanes file layout for the table in the top left. We reduced the row group size to 2 vectors of size 8 (v0.1 defaults are 64×1024). The FastLanes file footer is shown in the top right, and finally, the actual bytes of the FastLanes file format are shown at the bottom and a color legend bottom-left. Each box represents one byte, tagged below with a color that matches the legend, and a byte position in gray. FastLanes used Dictionary_FFOR_UINT8 encoding for the Name column, as the values come from a small domain. Constant encoding is used for the Institute column, as all values are equal. One-to-One Mapping is selected for the ID column, since it completely correlated with the Name column. Finally, the Alias column is compressed using Equality, as it is a copy of Name. For the first column in the binary format, there are a total of five segments: (1) one for bytes of dictionary values ("Alice" and "Bob"), (2) one for offsets of strings stored in the first segment ("0,5"), (3) one for the bit-packed array, which in this case requires two bytes as each of the 16 values can be represented in 1 bit, (4) one for the base values (0 and 0), and (5) one for bit widths (1 and 1). For Institute there is no segment, as the constant ("CWI") is stored in the metadata. For ID, we store only the dictionary with one segment consisting of values [100, 101].

by DuckDB's storage being allocated in fixed increments of 256 KB, thus allowing a fair evaluation of its compression performance.

Data. We chose data from the PUBLIC_BI [36, 96] benchmark as a basis to design and compare FastLanes against other file formats, and also used it to identify the expressions encodings (see Section 2.3). The PUBLIC_BI dataset is particularly relevant because it captures a wide variety of data distributions, is derived from real-world datasets, and has previously been used in the analysis, design, and evaluation of other encoding schemes such as ALP, FSST, White-Box Compression, C3, Chimp, Chimp128.

We used 36 datasets from PUBLIC_BI, summarized in Table 3, consisting of 2,289 columns. For consistency, we only considered the first table from each dataset to ensure equal-sized samples, as datasets vary in the number of tables. An exception was made for datasets with very few records, such as TrainsUK1, where we used Table 2 instead of Table 1.

Additionally, for datasets with similar schemas, such as Redfin1, Redfin2, Redfin3, and Redfin4, only the first dataset was included to avoid the effect of redundant tables on the results. For all experiments in this section, we selected 65,536 records (64 vectors) from each table to ensure fair benchmarking, as BtrBlocks also uses this number as the rowgroup size. Exceptions were made for the datasets CommonGovernment, Generico, and USCensus, where only 32,768 rows were used, as including 65,536 rows would result in file sizes exceeding 100MB.

Compression ratio. Table 3 summarizes the compression ratios for all evaluated file formats. All ratios are reported relative to FastLanes, where positive values indicate the percentage by which FastLanes compresses data more effectively. As shown, FastLanes clearly outperforms BtrBlocks as well as the commonly used Parquet default Snappy. Remarkably, using LWCs only, FastLanes still edges out Parquet+Zstd by 2% on PUBLIC_BI.

Table 3: Compression ratios of file formats relative to FastLanes on two datasets. FastLanes provides very good compression despite it using only Light-Weight Compression schemes; only Parquet with the slow and coarse-grained Zstd can equal it in compression ratio.

Dataset	CSV	FastLanes	Parquet		BtrBlocks	DuckDB
			Snappy	Zstd	[52]	v1.2
PUBLIC_BI	980.03 MB	172.05 MB	+41.0%	+2.0%	+18.0%	+66.0%
TPC-H	43.6 MB	11.8 MB	+20.8%	−15.3%	+17.5%	+14.0%

We tuned our expression pool and designed our rules while working on the PUBLIC_BI corpus. To verify that our design generalizes, we also included an experiment in which we compressed the first 64×1024 rows of each TPC-H table larger than that (i.e., excluding region and nation). The TPC-H results for FastLanes are also very good, but here Parquet+Zstd compressed 15.3% better. We identify two reasons for this: the TPC-H data (i) is normalized and does not offer MCC opportunities unlike PUBLIC_BI, and (ii) contains some synthetic data patterns (e.g. string columns shaped "String#Integer") that our encoding rules currently do not catch. A future version of the FastLanes encoder could detect this pattern and emit an encoding expression using the GLUE and CAST operators – splitting such columns in DICT- and FOR-compressible parts, respectively. The fact that FastLanes rowgroup metadata contains these expressions means that a FastLanes reader is forward-compatible with writers that emit new such expressions.

Table 4: Decoding/encoding throughput of file formats, based on the number of rowgroups decoded/encoded per second.

File Format	Total Decoding Time (ms)	Decoding (rowgroup/s)	Total Encoding Time (ms)	Encoding (rowgroup/s)
FastLanes	16.32	61.27	81341.63	0.012
Parquet+Snappy	712.55	1.40	5867.07	0.17
Parquet+Zstd	731.45	1.37	6927.52	0.14
BtrBlocks	115.43	8.66	111091.39	0.009
DuckDB	483.45	2.07	20347.82	0.05

Decoding and Encoding Speed. Next, we evaluate the decoding and encoding speed of all file formats. The results are shown in Table 4. FastLanes is faster on all datasets and, on average, is 43 times faster than Parquet+Snappy, 44 times faster than Parquet+ZSTD, 7 times faster than BtrBlocks, and 29 times faster than DuckDB.

Regarding encoding speed, we note that the FastLanes v0.1 code path is extremely basic and no effort whatsoever has been made to make it fast. With all optimization still on the table, we note it is already faster than BtrBlocks.

Random access. Next, we evaluate the random access time of all file formats. To do so, we execute the following query: `"SELECT * FROM read_parquet() LIMIT 1 OFFSET 0"` in DuckDB to retrieve only the first row from a Parquet file. For BtrBlocks, we fully decompress the entire block. Note that neither system offers true random access by row position, and we simulate a random-access workload using the given query in DuckDB. Regarding BtrBlocks, this could in principle be simulated by decoding only the first element of the last recursion. However, this was not possible, as it would require a complete reimplementation of BtrBlocks with a new API that supports this.

Table 5: Random access time comparison across different file formats. The result is presented in terms of the millisecond taken by FastLanes and how many times FastLanes is faster than others. FastLanes achieves the fastest access time.

Table	FastLanes	Parquet		BtrBlocks	DuckDB
Name	V0.1	Snappy	Zstd	[52]	v1.2
Total	0.14053	315.62x	413.66x	813.57x	5.96x

Table 6: Total decoding time of FastLanes, Parquet, and BtrBlocks on Intel Ice Lake using different SIMD compilation flags. AVX512 improves decoding time by nearly 40% on FastLanes, demonstrating the efficiency of data-parallelized encodings. In contrast, the performance gains for Parquet+Zstd, Parquet+Snappy, and BtrBlocks are negligible. BtrBlocks only works on machines with AVX2/AVX512 instruction sets.

ISA	FastLanes		Parquet+Snappy		Parquet+Zstd		BtrBlocks	
	Time(ms)	Speedup	Time(ms)	Speedup	Time(ms)	Speedup	Time(ms)	Speedup
SSE	23.16	-	2110.27	-	2152.29	-	✖	✖
AVX2	18.96	22.10%	2057.16	2.58%	2167.52	-0.70%	113.09	-
AVX512	16.32	41.87%	2070.90	1.90%	2197.58	-2.06%	115.43	-2.07%

The results are shown in Table 5. FastLanes takes 0.14 milliseconds to retrieve the first value from all datasets, making it 315 times faster than Parquet+Snappy, 416 times faster than Parquet+ZSTD, 800 times faster than BtrBlocks, and 5 times faster than DuckDB. DuckDB is the closest to FastLanes in performance. This benchmark demonstrates that block-based compression methods are extremely inefficient for random access, as they require decompressing the entire block to access a single value. In contrast, the vectorized decoding model used in both FastLanes and DuckDB provides a good balance between compression ratio and small enough block sizes to enable efficient tuple retrieval. Overall, the results highlight the inefficiency of block-based compression for random access and the advantage of vectorized decoding in balancing compression effectiveness and retrieval speed.

SIMD. To evaluate in howfar fully data-parallelized encodings improve with SIMD, we benchmark the total decoding time of FastLanes on Intel Ice Lake using three different compilation flags: `-O3`, `-O3 -mavx2`, and `-O3 -mavx512dq`.

The results are shown in Table 6. As observed, AVX512 improves decoding time by nearly 40%. To further emphasize the necessity of next-generation file formats for data-parallel encodings, we repeat the same benchmark for Parquet+Zstd, Parquet+Snappy, and BtrBlocks. The observed performance gain is negligible. While BtrBlocks uses explicit SIMD instructions, it employs non-fully data-parallel layouts, which limit its ability to benefit from AVX512, as clearly shown in Table 6. This benchmark clearly demonstrates the importance of fully data-parallel encodings.

Expression Pool. To measure the effect of each expression in our pool, we conducted an experiment where we measured the impact of each expression encoding included, compared to when it was removed from the pool. The results indicate how much an expression practically improves the compression ratio performance and decompression time of FastLanes[4]

The results are shown in Table 7. `Dictionary` encoding has the most significant impact on the compression ratio, improving it by 40%, followed by `DELTA` decoding, which improves it by 6%. Based

[4]This micro-benchmark was conducted on an Apple MacBook Pro M4.

Table 7: Improvement brought by adding each scheme to the pool compared to having it removed. Usefulness is a mix of impact on compression ratio, decompression speed (and code complexity - but this is harder to quantify).

Expression	Compression Ratio	Decompression Speed
Dictionary	+42.36%	+44.19%
DELTA	+5.92%	-1.91%
Equality	+4.70%	+2.66%
ALP	+4.36%	-7.28%
FSST	+3.86%	+3.84%
Patch	+2.51%	-7.11%
FFOR	+1.30%	+4.48%
One-to-One Map	+1.17%	+9.47%
FSST12	+0.84%	+2.62%
Cast	+0.78%	+10.16%
RLE	+0.69%	+6.65%
CROSS RLE	+0.65%	+16.27%
ALP RD	+0.57%	-2.30%
Frequency	+0.09%	+2.06%
Constant	+0.00%	+2.92%

on these results, we address the following questions, which could serve as guidelines for future file formats, including FastLanes.

Exception Handling. Despite its proven benefits [59, 107], almost all new file formats, such as BtrBlocks, DuckDB's native file format, and Nimble from Meta, avoid supporting any exception handling mechanism. The `PATCH` operator, which handles exceptions, improves the compression ratio by 2.5% in the presence of all schemes, demonstrating its significance. We consider patching a first-class citizen in FastLanes, though we notice that it has considerably slowed down decompression.

FSST. FSST12 and FSST work similarly, with the key difference that FSST12 uses 12-bit symbols, allowing it to capture 16 times more symbols at the cost of 4-bit longer codes. This raises the question: do future file formats need FSST, FSST12, or both?

By looking at the table, FSST improves the compression ratio by almost 4%, which is significant, while FSST12 contributes only 1%, which is still meaningful. Despite not having as much impact as FSST, a detailed analysis shows that FSST12 performs very well on long string columns, making our file format more future-proof for handling long strings. Therefore, we support both FSST12 and FSST in FastLanes.

Frequency. BtrBlocks argues for using frequency encoding, which considers the most commonly used value as a default and stores only values that do not match the most frequent one. In a sense, it is similar to constant encoding with exceptions.

Our analysis, summarized in Table 7, shows that this scheme brings only a 0.05% improvement, which is very insignificant compared to the complexity it adds to the file format. Therefore, in FastLanes, we do not support the Frequency encoding.

MCC. MCC schemes, including Equality with (4.7%, 2.6%), One-to-One Mapping with (1.2%, 9.47%), and Cast with (0.78%, 10.16%), bring an overall improvement of approximately (8%, 20%) to the compression ratio and decoding speed of FastLanes, which is very significant. Therefore, we support MCC schemes as a first-class citizen of FastLanes and continue to explore further improvements to our MCC schemes.

5 RELATED WORK

Today, there are multiple open columnar [1] file formats, including Parquet [12], RCFile [44], ORC [11], BtrBlocks [53], DuckDB [82, 83], Albis [93], Carbon [10], DataBlocks [55], Artus [20], Capacitor [77], LanceDB [30], Bullion [66], and Nimble [66]. These formats have been analyzed and surveyed in [3, 49, 67, 102], and their integration into real systems has also been studied in [86]. In this section, we first review the work already done in the FastLanes project. Then, we review BtrBlocks as the current state-of-the-art file format, the first (and only) to implement cascaded encoding. We also examine studies in the database context, focusing specifically on `Cascaded Encoding` and `Multi-Column Compression (MCC)`, and conclude by comparing the FastLanes file format to both BtrBlocks and Parquet.

5.1 FastLanes

In the initial work on FastLanes, we introduced and implemented fully data-parallel encodings with zero explicit SIMD instructions, including bit-packing, dictionary encoding, FastLanes-RLE, delta encoding, and FastLanes-FOR, which auto-vectorize to match the performance of explicit SIMD implementations [6]. These schemes are the fastest LWCs available now. Further, we developed and implemented ALP [8], another auto-vectorizing encoding for float and double. ALP offers not only very fast decoding but also leading ratios for lossless floating-point compression. In this paper, we *use* these encodings, as well as FSST – and for the first time FSST12 and MCC – to create the FastLanes file format, that can go head-to-head with Parquet. Its novelties lie in an effective operator pool for cascading compression, expression-encoded metadata, and segmented data layout to match it.

5.2 BtrBlocks

BtrBlocks implements cascaded compression through recursion, where an entire column chunk is compressed recursively using multiple lightweight compression schemes (LWCs). Here we highlight some key reasons FastLanes provides advantages over BtrBlocks.

Block-Based Compression: Despite using LWCs that could potentially support vectorized decoding, BtrBlocks' cascading compression reverts to a coarse-grained approach. This is due to the recursive nature of the implementation, which requires an entire rowgroup (64*1024 values) to be fully [de]compressed multiple times for each LWC used in a combination. One could argue that the size 64*1024 could be reduced to one vector of 1024, and that repeating the process of recursive decompression for each vector could support vectorized decoding. However, this approach is not feasible because, for example, the crucial dictionary encoding codepath would then get executed separately for each vector, resulting in a separate dictionary being stored for each vector. This could

lead to dictionaries with potentially repeated values across vectors and significantly worse compression ratios.

No Compressed Execution Support: BtrBlocks always completely decompresses values. While one could extend its cascaded decoding to support compressed execution, this would only provide a limited form by skipping decoding of the final recursion level. This limitation arises from the recursive nature of its implementation, where lower levels of encoding remain opaque and can only be accessed after full decoding.

Not Fully Data-Parallelized: BtrBlocks relies on 128-bit interleaved bit-packing provided by the FastPFOR library. At best, a 128-bit interleaved layout can only use a SIMD register of width 128 bits or only 4 threads of a 32-threaded warp – and there is no BtrBlock GPU decoder yet.

Missing Schemes: The BtrBlocks scheme pool lacks three critical schemes: `ALP`, `Patching`, and `Delta encoding`. `ALP` is a state-of-the-art encoding for floating-point data and an essential LWC scheme to achieve better compression ratios than `Zstd`. `Delta encoding` is crucial in niche domains (e.g., timeseries data) but is also a useful component for encoding offsets needed for string storage.

Hardcoded: Btrblocks implements cascaded compression with hardcoded configurations. For example, in all cascades involving dictionary encoding, `uint32` is always used for codes. In contrast, the FastLanes implementation of cascaded encoding allows an expression of any combination of operators; and even though we limit the space of possibilities with an expression pool, it offers multiple variants for dictionary codes. In FastLanes, dictionary codes are typically further compressed with `FFOR`. Note that in the `FFOR` of FastLanes, the bit-width is a parameter that gets stored in a column segment, and hence it can vary from vector to vector.

Dependencies: the BtrBlocks implementation relies on several external dependencies, which complicates its use in practical systems. We argue that a file format should be implemented with zero external dependencies, following the DuckDB and SQLite implementation model, making it usable as an embeddable library for any query execution engine.

5.3 Encoding/Compression.

Encoding/compression is frequently studied in database systems, with a focus on improving decoding speed [6, 43, 57–61, 80, 88, 90, 94, 95, 97, 99, 106, 107], optimizing encoding/compression selection [15, 51], enhancing compression ratios [50, 69, 73, 84, 87, 101, 105], integrating compression with query execution [2, 14, 29, 45, 47, 50, 98, 100, 103], evaluating predicates on encoded data [34, 63, 64, 81], and GPU encoding/compression [33]. HWC schemes, such as `Zstd` [22], `Snappy` [40], and `LZ4` [21], are the default in most open file formats. Several LWC schemes have also been developed to encode specific data types, such as `DOUBLE` [8, 18, 35, 53, 54, 62, 65, 79, 85], `INTEGER` [3, 6, 39, 59, 60], and `STRING` [16, 74, 104]. Grammar-based compression schemes like `Sequitur` [75], `Re-Pair` [56], and `GLZA` [23] have been proposed to compress data by building a context-free grammar for it. However, these grammar-based schemes are generally unsuitable for file formats due to their slower decompression speeds [16].

Cascaded Encoding. Fang *et al.* [33] propose cascaded encoding, which combines LWC schemes to improve compression ratios. Damme *et al.* classify LWC schemes into logical and physical compression categories and study how well they can be combined [27, 28]. However, their work is limited to integer columns and combinations of at most two LWC schemes. Afroozeh *et al.* [3, 5] propose a Composable Compression Model that decomposes LWC schemes into several efficient functions that can later be used to construct more complex encodings, though this work focuses on decoding speed rather than compression ratios. BtrBlocks [53] implements cascaded encoding recursively, while Nvidia's nvCOMP [76] applies cascaded encoding recursively for GPUs, though it is limited to a single variation of `[DICT, RLE, BITPACK]`.

Multi-Column Compression. `White-box Compression` [37] proposes a conceptual model that represents logical columns in tabular data as an openly defined function over some physically stored columns, allowing the query optimizer to enable optimizations such as improved filter predicate pushdown during query execution. `PIDS` [50] identifies common patterns in string attributes using an unsupervised approach and uses the discovered patterns to split each attribute into sub-attributes. These sub-attributes can then be encoded individually, which enables future engines to push down many query operators to sub-attributes, thereby minimizing I/O and potentially costly comparisons, resulting in faster execution of query operators. `C3` [38] proposes six MCC schemes—`Equality, 1To1Dict, 1toNDict, Numerical, DFOR, SharedDic` – to address a key limitation of column stores relative to row stores, namely that they compress attributes of each record in isolation. `Corra` [68], similar to C3, looks for column correlation for compression and proposes the same compression schemes under different names with the same compression ratio.`Virtual` [91] implements Corra in Python on top of Parquet. `Expression encoding` extends and integrates the concepts of `White-box Compression`, `PIDS`, and `C3` by proposing a unified framework that allows future file formats to fully leverage MCC schemes.

FastLanes vs BtrBlocks vs Parquet: Regarding compression ratio FastLanes typically compresses better, thanks to its support for cascading encodings and MCC. Only the less-often used combination of Parquet+ZSTD is on par with it. Regarding decompression speed, FastLanes is at least an order of magnitude faster, and it can also very quickly provide fine-grained access. Both BtrBlocks and Parquet must decode large blocks of data and hence are coarse-grained. Parquet relies on HWCs and is therefore much slower. BtrBlocks is not fully data-parallel; and this property makes FastLanes also GPU-friendly [7, 46] and in our eyes, more future-proof.

6 DISCUSSION

In this section, we discuss two layout strategies – Unified Transposed Layout within a vector and Segmented Page Layout within a page – as these are fundamental decisions for future file formats.

Unified Transposed Layout. We use the Unified Transposed Layout (UTL) [6] as an option rather than as the default. Although this layout enables complete data-parallelism for `FastLanes-RLE` and

Delta schemes, its effect to permute the order of the tuples in a vector may sometimes not be desirable – though the original order can always be restored, this comes at an overhead. However, we argue that the substantial compression ratios achieved by FastLanes-RLE and DELTA make these schemes essential, making the UTL a valuable option for efficient data-parallelized decoding [6] in contexts where high compression ratios are a priority.

We address a common point of confusion regarding the applicability of the Unified Transposed Layout to variable-sized data, such as String or List. We find it can be used without problem, as variable-sized data are always accompanied by an offset array – a *fixed-size* vector of 1024 values to which we apply the UTL.

Vectorized Page Layout. An alternative to the segmented page layout for storing the result of an expression in a file is the vectorized page layout, where all encoded data of an expression for a vector are stored sequentially in one place. We have chosen the segmented page layout over the vectorized page layout for three main reasons: Supporting structs in a vectorized page layout can lead to filling the cache with unnecessary data during reading, particularly when only subfields of a struct are required. The segmented page layout allows for additional query optimizations by enabling query engines to access relevant data in a single location, such as the bases in FFOR, which effectively serve as vector-based zone maps [71], specifically the minimums of each vector. Collecting data with similar properties in one segment allows for further compression in a single pass. Although we currently avoid compressing these segments, having this option remains beneficial for compression-sensitive workloads where achieving a high compression ratio is a priority.

7 CONCLUSION

Popular big data file formats only partially benefit from the full compression potential of Light-Weight Compression (LWC) schemes [3, 53], missing opportunities for compressed execution, cascaded compression and multi-column compression. The latter two issues affect compression ratio and make the use of Heavy-Weight Compression (HWC) methods necessary, even though these are SIMD and GPU unfriendly. This is why FastLanes introduces Expression Encoding, paired with a intricate segmented page design, that enables fine-grained and efficient decoding of cascading LWC schemes.

With this paper, we release a high-quality open-source C++ implementation of FastLanes v0.1. Designing a data format requires a lot of effort and getting many details right. We think this release is a major contribution.

Our evaluation of FastLanes versus Parquet, BtrBlocks and the DuckDB format shows that HWCs can now be avoided without sacrificing any compression ratio, and very significantly improving decoding speed; while offering efficient fine-grained data access as well as novel opportunities for compressed execution.

8 FUTURE WORK

Schema Evolution. Future file formats should support schema evolution. Parquet currently offers a limited form of schema evolution for changing types within a column [48], allowing only the promotion of a few specific types, rather than broader support for other data types. We believe that Expression Encoding enables file formats to support this feature seamlessly, thanks to the type information included in each expression.

Storing the footer in a separate place allows to include new types and new columns by modifying, resp. generating new expressions in the footer, without having to rewrite existing rowgroup data.

End-to-End Benchmark. The focus of this paper is on benchmarking compression ratios rather than evaluating the extent to which the FastLanes file format accelerates query execution engines. Operators such as [Cast, Constant] proposed in this paper are intended to help query engines execute queries more efficiently. Therefore, we left a full evaluation of FastLanes integrated in a query engine for the future. Our next development step will be a DuckDB reader/writer for Fastlanes.

Encryption. Our proposed expression encoding is perfect match to support encryption in a vectorized manner, contrary to Encryption in Parquet which is block-based. In our vision, encrypting merely is one more operator at the end of an expression, which [en/de]crypts the compressed vector of an expression.

Cascaded Encoding on GPU. The state-of-the-art encoding model on GPU, Tile-based decoding [89], proposes decoding data in small batches, called tiles, within a GPU's shared memory to avoid transferring data back to global memory—a primary bottleneck in GPU performance. Additionally, it supports cascaded encoding limited to FOR, DELTA, and RLE, with both the value and length arrays further bit-packed. Expression Encoding aligns with the concept of decoding a batch of data that fits in shared memory, while offering more cascaded combinations capable of achieving a better compression ratio than Zstd and supporting compressed tiles similar to compressed vectors in DuckDB [83] and Velox [78]. We speculate that a CUDA implementation of FastLanes could bring significantly higher decoding speeds and improved compression ratios to the GPU processing ecosystem. Inital results look promising [7].

Nested data types Nested data types, such as structs, lists, and maps, are widely used and natively supported by open big data file formats like Apache Parquet. Recent work at CWI on real-world JSON datasets [42] suggests that applying LWC schemes to flatten nested data types, which resemble columns, is less effective for compression than using HWC schemes. However, new nesting-specific encodings [72] could significantly reduce this gap.

Machine Learning Data. Several file formats have emerged to address the needs of machine learning data workloads, including Bullion [66], which tackles the complexities of data compliance, optimizes the encoding of long-sequence sparse features, and efficiently manages wide-table projections. Nimble [92], a columnar file format from Meta, is designed for very wide tables commonly found in machine learning training datasets. LanceDB [30], another columnar data format, is optimized for machine learning workloads, offering high-performance random access and efficient handling of complex data types, including images and videos.

Adding support for nested data types and wide and sparse data, that is characteristic of machine learning data workloads, has been part of the FastLanes design process, and implementation efforts are underway.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems*. Vol. 3. Now Publishers Inc., Cambridge, MA, USA. 197–280 pages. https://doi.org/10.1561/1900000014

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[3] A Afroozeh. 2020. *Towards a New File Format for Big Data: SIMD-Friendly Composable Compression*. Master's thesis. centrum wiskunde & informatica. https://homepages.cwi.nl/~boncz/msc/2020-AzimAfroozeh.pdf

[4] Azim Afroozeh. 2024. FastLanes End-to-End Script. https://gist.github.com/azimafroozeh/b5d0dbea44ee7cd6dc39b0c4b0f7ef38 Accessed: 2024-11-29.

[5] Azim Afroozeh and Peter Boncz. 2021. FastLanes: A SIMD-friendly Composable Compression Library. In -. DBDBD, -, –. https://www.wis.ewi.tudelft.nl/assets/DBDBD2021_submissions/DBDBD2021_paper_10.pdf

[6] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (jul 2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[7] Azim Afroozeh and Peter Boncz. 2023. FastLanes on GPU: Analysing Data-Parallelized Compression Schemes. DaMoN workshop.

[8] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless Floating-Point Compression. *Proc. ACM Manag. Data* 1, 4, Article 230 (dec 2023), 26 pages. https://doi.org/10.1145/3626717

[9] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 169–180.

[10] Apache. 2023. *Apache CarbonData*. Apache. https://carbondata.apache.org/.

[11] Apache. 2023. *Apache Orc*. Apache. https://orc.apache.org/.

[12] Apache. 2023. *Apache Parquet*. Apache. http://parquet.apache.org/.

[13] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*. https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[14] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 283–296. https://doi.org/10.1145/1559845.1559877

[15] Martin Boissier and Max Jendruk. 2019. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, Lisbon, Portugal, 674–677. https://doi.org/10.5441/002/edbt.2019.84

[16] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2649–2661. https://doi.org/10.14778/3407790.3407851

[17] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the 2005 Conference on Innovative Data Systems Research (CIDR)*. Very Large Data Base Endowment, Asilomar, CA, USA, 225–237. https://www.cidrdb.org/cidr2005/papers/P19.pdf

[18] Andrea Bruno, Franco Maria Nardini, Giulio Ermanno Pibiri, Roberto Trani, and Rossano Venturini. 2021. TSXor: A Simple Time Series Compression Algorithm. In *String Processing and Information Retrieval: 28th International Symposium, SPIRE 2021, Lille, France, October 4–6, 2021, Proceedings (Lecture Notes in Computer Science)*, Vol. 12944. Springer, Lille, France, 217–223. https://doi.org/10.1007/978-3-030-86692-1_18

[19] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying Serving and Analytical Data at YouTube. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2022–2034. https://doi.org/10.14778/3352063.3352121

[20] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying Serving and Analytical Data at YouTube. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2022–2034. https://doi.org/10.14778/3352063.3352121

[21] Yann Collet. 2014. LZ4 - Extremely fast compression. https://github.com/lz4/lz4 Accesed on: 2023-04-13.

[22] Yann Collet. 2015. Zstandard - Fast real-time compression algorithm. https://github.com/facebook/zstd Accesed on: 2023-04-13.

[23] Kennon J. Conrad and Paul R. Wilson. 2016. Grammatical Ziv-Lempel Compression: Achieving PPM-Class Text Compression Ratios with LZ-Class Decompression Speed. In *Proceedings of the 2016 Data Compression Conference (DCC)*. IEEE Computer Society, Snowbird, UT, USA, 586. https://doi.org/10.1109/DCC.2016.119

[24] George P. Copeland and Setrag N. Khoshafian. 1985. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data* (Austin, Texas, USA) *(SIGMOD '85)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/318898.318923

[25] cwida. 2023. Fast Static Symbol Table (FSST). https://github.com/cwida/fsst. Accessed: 2025-02-25.

[26] Benoit 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 *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[27] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, Venice, Italy, 72–83. https://openproceedings.org/2017/conf/edbt/paper-146.pdf

[28] 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, Article 9 (jun 2019), 46 pages. https://doi.org/10.1145/3323991

[29] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2396–2410. https://doi.org/10.14778/3407790.3407833

[30] LanceDB Developers. 2024. LanceDB: A Modern Vector Database. https://github.com/lancedb/lancedb. Accessed: 2024-11-29.

[31] DuckDB. 2025. Announcing DuckDB 1.20. https://duckdb.org/2025/02/05/announcing-duckdb-120.html Accessed: 2025-02-25.

[32] DuckDB. 2025. Parquet Encodings. https://duckdb.org/2025/01/22/parquet-encodings.html Accessed: 2025-02-25.

[33] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 670–680. https://doi.org/10.14778/1920841.1920927

[34] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelope of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, Melbourne, Victoria, Australia, 31–46.

[35] Nathaniel Fout and Kwan-Liu Ma. 2012. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2295–2304.

[36] Bogdan Ghita. 2019. Public BI Benchmark. https://github.com/cwida/public_bi_benchmark. Accessed on: 2023-04-13.

[37] Bogdan Ghita, Diego G. Tomé, and Peter A. Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *Proceedings of the 2020 Conference on Innovative Data Systems Research (CIDR)*. Very Large Data Base Endowment, Amsterdam, The Netherlands, 23. https://ir.cwi.nl/pub/29515

[38] T Glass. 2023. *C3: Compressing Correlated Columns*. Master's thesis. centrum wiskunde & informatica. https://homepages.cwi.nl/~boncz/msc/2023-ThomasGlas.pdf

[39] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Orlando, FL, USA, 370–379. https://doi.org/10.1109/ICDE.1998.655800

[40] Google. 2014. Snappy - A fast compressor/decompressor. https://github.com/google/snappy Accesed on: 2023-12-04.

[41] Google LLC. 2014. *FlatBuffers: Efficient Cross-Platform Serialization Library*. Google LLC. https://google.github.io/flatbuffers/ Accessed: 2025-06-01.

[42] CWI Database Architectures Group. 2024. RealNest - A Collection of Nested Data from Real-World Datasets. https://github.com/cwida/RealNest

[43] Dirk Habich, Patrick Damme, Annett Ungethüm, and Wolfgang Lehner. 2018. Make Larger Vector Register Sizes New Challenges? Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages.

https://doi.org/10.1145/3209950.3209957

[44] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 2011. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Hannover, Germany, 1199–1208. https://doi.org/10.1109/ICDE.2011.5767933

[45] Linus Heinzl, Ben Hurdelhey, Martin Boissier, Michael Perscheid, and Hasso Plattner. 2021. Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS. In *Proceedings of the 11th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2021)*. ADMS Workshop Organizers, Copenhagen, Denmark, 26–36. https://adms-conf.org/2021-camera-ready/heinzl_adms21.pdf

[46] S Hepkema. 2025. *FastLanes on GPU*. Master's thesis. centrum wiskunde & informatica. https://azimafroozeh.org/assets/master_thesis/sven_thesis.pdf

[47] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. 2017. Compression-Aware In-Memory Query Processing: Vision, System Design and Beyond. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*. Springer, Venice, Italy, 40–56. https://doi.org/10.1007/978-3-319-56111-0_3

[48] Apache Iceberg. 2023. Apache Iceberg Specification - Writer Requirements. https://iceberg.apache.org/spec/#writer-requirements Accessed: 2023-11-01.

[49] Todor Ivanov and Matteo Pergolesi. 2019. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32 (09 2019). https://doi.org/10.1002/cpe.5523

[50] Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, and Aaron J. Elmore. 2020. PIDS: Attribute Decomposition for Improved Compression and Query Performance in Columnar Storage. *Proc. VLDB Endow.* 13, 6 (feb 2020), 925–938. https://doi.org/10.14778/3380750.3380761

[51] 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 *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 843–856. https://doi.org/10.1145/3448016.3457283

[52] Maximilian Kuschewski and contributors. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. https://github.com/maxi-k/btrblocks GitHub repository, accessed on February 18, 2025.

[53] 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 (jun 2023), 26 pages. https://doi.org/10.1145/3589263

[54] DuckDB Labs. 2022. Patas Compression: Variation on Chimp. Accessed on: 2023-04-13.

[55] 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 *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

[56] N.J. Larsson and A. Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732. https://doi.org/10.1109/5.892708

[57] Robert Lasch, Ismail Oukid, Roman Dementiev, Norman May, Suleyman S. Demirsoy, and Kai-Uwe Sattler. 2019. Fast & Strong: The Case of Compressed String Dictionaries on Modern CPUs. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3329785.3329924

[58] Florian Lemaitre, Arthur Hennequin, and Lionel Lacassagne. 2020. How to Speed Connected Component Labeling up with SIMD RLE Algorithms. In *Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing* (San Diego, CA, USA) *(WPMVP'20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/3380479.3380481

[59] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

[60] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD Compression and the Intersection of Sorted Integers. *Softw. Pract. Exper.* 46, 6 (jun 2016), 723–749.

[61] Daniel Lemire and Christoph Rupp. 2017. Upscaledb: Efficient integer-key compression in a key-value store using SIMD instructions. *Information Systems* 66 (2017), 13–23. https://doi.org/10.1016/j.is.2017.01.002

[62] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-based Lossless Floating-Point Compression. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1763–1774.

[63] Yinan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores Using Bit Manipulation Instructions. *Proc. ACM Manag. Data* 1, 2, Article 178 (jun 2023), 26 pages. https://doi.org/10.1145/3589323

[64] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/2463676.2465322

[65] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3058–3070.

[66] Gang Liao, Ye Liu, Jianjun Chen, and Daniel J. Abadi. 2024. Bullion: A Column Store for Machine Learning. arXiv preprint arXiv:2404.08901.

[67] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proc. VLDB Endow.* 16, 11 (aug 2023), 3044–3056. https://doi.org/10.14778/3611479.3611507

[68] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. 2024. Corra: Correlation-Aware Column Compression. arXiv:2403.17229 [cs.DB] https://arxiv.org/abs/2403.17229

[69] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2023. LeCo: Lightweight Compression via Learning Serial Correlations. arXiv:2306.15374 [cs.DB]

[70] Xi Lyu, Andreas Kipf, Pascal Pfeil, Dominik Horn, Jana Giceva, and Tim Kraska. 2023. CorBit: Leveraging Correlations for Compressing Bitmap Indexes. In *Proceedings of the Fifth International Workshop on Applied AI for Database Systems and Applications (AIDB 2023) (CEUR Workshop Proceedings)*, Vol. 3462. CEUR-WS.org, Vancouver, Canada, 1–10.

[71] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, New York, NY, USA, 476–487. http://www.vldb.org/conf/1998/p476.pdf

[72] Ziya Mukhtarov. 2024. *Nested Data-Type Encodings in FastLanes*. Master's thesis. Technical University of Munich. https://homepages.cwi.nl/~boncz/msc/2024-ZiyaMukhtarov.pdf

[73] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, Athens, Greece, 283–294. https://openproceedings.org/EDBT/2014/paper_25.pdf

[74] Bhavik Nagda. 2021. *CHuff: Conditional Huffman String Compression*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[75] C.G. Nevill-Manning and I.H. Witten. 1997. Linear-Time, Incremental Hierarchy Inference for Compression. In *Proceedings of the Data Compression Conference (DCC)*. IEEE Computer Society, Snowbird, UT, USA, 3–11. https://doi.org/10.1109/DCC.1997.581951

[76] NVIDIA. 2023. nvCOMP. https://github.com/NVIDIA/nvcomp. Accessed on: 2023-4-12.

[77] Mosha Pasumansky. 2023. Inside Capacitor, BigQuery's Next-Generation Columnar Storage Format. https://cloud.google.com/blog/products/bigquery/inside-capacitor-bigquerys-next-generation-columnar-storage-format. Accessed: 2023-10-10.

[78] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.

[79] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.

[80] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (Melbourne, VIC, Australia) *(DaMoN'15)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/2771937.2771943

[81] Martin Prammer and Jignesh M. Patel. 2023. Rethinking the Encoding of Integers for Scans on Skewed Data. *Proc. ACM Manag. Data* 1, 4, Article 257 (dec 2023), 27 pages. https://doi.org/10.1145/3626751

[82] Mark Raasveldt. 2022. Lightweight Compression in DuckDB. https://duckdb.org/2022/10/28/lightweight-compression.html. Accesed on: 2023-04-13.

[83] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam, Netherlands, 1981–1984.

[84] Vijayshankar Raman and Garret Swart. 2006. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. Citeseer, VLDB Endowment, Seoul, Korea, 858–869.

[85] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. 2006. Fast Lossless Compression of Scientific Floating-Point Data. In *Data Compression Conference (DCC'06)*. IEEE, IEEE, Snowbird, Utah, USA, 133–142.

[86] Alice Rey. 2024. Seamless Integration of Parquet Files into Data Processing. In *Proceedings of the Workshops of the EDBT/ICDT 2024 Joint Conference*, Vol. 3651. CEUR-WS.org. https://ceur-ws.org/Vol-3651/PhDW-3.pdf

[87] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (sep 1993), 31–39. https://doi.org/10.1145/163090.163096

[88] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2010. Fast Integer Compression Using SIMD Instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware* (Indianapolis, Indiana) *(DaMoN '10)*. Association for Computing Machinery, New York, NY, USA, 34–40. https://doi.org/10.1145/1869389.1869394

[89] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-Based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1390–1403. https://doi.org/10.1145/3514221.3526132

[90] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. 2011. SIMD-Based Decoding of Posting Lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (Glasgow, Scotland, UK) *(CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 317–326. https://doi.org/10.1145/2063576.2063627

[91] Mihail Stoian, Alexander van Renen, Jan Kobiolka, Ping-Lin Kuo, Josif Grabocka, and Andreas Kipf. 2024. Lightweight Correlation-Aware Table Compression. In *NeurIPS 2024 Third Table Representation Learning Workshop*. https://openreview.net/forum?id=z7eIn3aShi

[92] Facebook Incubator Team. 2024. Nimble: A Columnar File Format for Feature Engineering. https://github.com/facebookincubator/nimble. GitHub Repository.

[93] Animesh Kr Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schüpbach, and Bernard Metzler. 2018. Albis: High-Performance File Format for Big Data Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, Boston, MA, USA, 561–574. https://api.semanticscholar.org/CorpusID:51876043

[94] Andrew Trotman and Jimmy Lin. 2016. In Vacuo and In Situ Evaluation of SIMD Codecs. In *Proceedings of the 21st Australasian Document Computing Symposium* (Caulfield, VIC, Australia) *(ADCS '16)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3015022.3015023

[95] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Dirk Habich, and Wolfgang Lehner. 2018. Conflict Detection-Based Run-Length Encoding—AVX-512 CD Instruction Set in Action. In *Proceedings of the 34th IEEE International Conference on Data Engineering Workshops (ICDEW)*. IEEE Computer Society, Paris, France, 96–101. https://doi.org/10.1109/ICDEW.2018.00023

[96] 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 2018 Workshop on Testing Database Systems (DBTest 2018)*. Association for Computing Machinery, Houston, TX, USA, 1–6. https://doi.org/10.1145/3209950.3209952

[97] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. 2017. MILC: Inverted List Compression in Memory. *Proc. VLDB Endow.* 10, 8 (apr 2017), 853–864. https://doi.org/10.14778/3090163.3090164

[98] Richard Michael Grantham Wesley and Pawel Terlecki. 2014. Leveraging Compression in the Tableau Data Engine. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 563–573. https://doi.org/10.1145/2588555.2595639

[99] 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 (aug 2009), 385–394. https://doi.org/10.14778/1687627.1687671

[100] Leon Windheuser, Christoph Anneser, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2024. Adaptive Compression for Databases. In *Proceedings of the 27th International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, Paestum, Italy, 143–149. https://doi.org/10.5441/002/edbt.2019.84

[101] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Proceedings of the 18th International World Wide Web Conference (WWW 2009)*. Association for Computing Machinery, Madrid, Spain, 401–410. https://doi.org/10.1145/1526709.1526764

[102] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (2023), 148–161. https://doi.org/10.14778/3626292.3626298

[103] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1655–1669. https://doi.org/10.1145/3514221.3526130

[104] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1601–1615. https://doi.org/10.1145/3318464.3380583

[105] Jiujing Zhang, Zhitao Shen, Shiyu Yang, Lingkai Meng, Chuan Xiao, Wei Jia, Yue Li, Qinhui Sun, Wenjie Zhang, and Xuemin Lin. 2023. High-Ratio Compression for Machine-Generated Data. *Proc. ACM Manag. Data* 1, 4, Article 245 (dec 2023), 27 pages. https://doi.org/10.1145/3626732

[106] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. 2015. A General SIMD-Based Approach to Accelerating Compression Algorithms. *ACM Trans. Inf. Syst.* 33 (2015), 15:1–15:28. https://api.semanticscholar.org/CorpusID:12175168

[107] Marcin Zukowski, Sándor Heman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Atlanta, GA, USA, 59–59. https://doi.org/10.1109/ICDE.2006.150

[108] Marcin Zukowski, Niels Nes, and Peter Boncz. 2008. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware* (Vancouver, Canada) *(DaMoN '08)*. Association for Computing Machinery, New York, NY, USA, 47–54. https://doi.org/10.1145/1457150.1457160