SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Nested Data-Type Encodings in FastLanes

Ziya Mukhtarov

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Nested Data-Type Encodings in FastLanes

Verschachtelte Datentypen-Kodierungen in FastLanes

Author:	Ziya Mukhtarov
Supervisor:	Prof. Dr. Thomas Neumann
Advisors:	Prof. Dr. Peter Boncz
	Azim Afroozeh
Submission Date:	15.07.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.07.2024 Ziya Mukhtarov

Acknowledgments

This thesis was completed at Centrum Wiskunde & Informatica (CWI) in Amsterdam, Netherlands. I want to thank CWI Database Architectures Group, especially Prof. Dr. Peter Boncz and Azim Afroozeh, for welcoming me and advising me for the duration of this thesis. I am also grateful to Azim for his insights into my various implementations in the FastLanes project and his efforts to maintain a clean codebase. I would also like to thank Prof. Dr. Stefan Manegold for his support in various onboarding activities, such as choosing and accessing CWI SciLens cluster machines, and Laurens Kuiper for giving insights into DuckDB internals and supporting me throughout my DuckDB contribution. Finally, I thank my CWI colleagues Paul Gross, Niclas Haderer, and Raufs Dunamalijevs for their moral support during challenging times and for regularly playing table tennis with me.

Abstract

Nested data types, like structs, lists, and maps, allow users to store data with complex hierarchical structures. Widespread open big data file formats like Apache Parquet, Apache ORC, and DuckDB's database format have built-in support for these types. These file formats compress the data to store it efficiently by reducing its size and apply decompression when the data is accessed. The critical characteristics of compression algorithms are the compression ratio, an indicator of how much size reduction was achieved, and the decompression speed, which shows how fast the compressed data can be decoded to the original form. Parquet and ORC rely on general-purpose compression schemes, which operate on the byte level and can be applied to any data to achieve high compression ratios with unideal decompression speeds. Lightweight compression schemes for exploiting data patterns have been developed to speed up decompression by keeping the algorithms simple and parallelizable while still achieving high compression ratios.

FastLanes is a new file format in development at CWI aiming to address the shortcomings of other open big data file formats. It uses innovative data-parallel compression layouts and combines multiple lightweight encodings that facilitate fast decompression and provide high compression ratios. This thesis analyzes nested data to find common patterns and studies encodings suitable for FastLanes that exploit those patterns. To represent the structure of nested data in real-world scenarios, the RealNest dataset, which contains nested data from various public user-generated datasets, is introduced as a compression corpus for nested data types. The data from RealNest suggests that the existing representations for nested data types can lose critical data patterns that are significant for lightweight compression. Three new lightweight encoding schemes are introduced for the list data type and evaluated against the other FastLanes schemes on the RealNest dataset. The results show a need for special lightweight nested data type encodings that consider the structure of the data in file formats to push the compression ratio higher and reach the compression ratios of general-purpose compression schemes.

Contents

1 Introduction

In the modern landscape of data-intensive applications, efficient storage and swift data retrieval is crucial. Over the years, many special file formats, like Apache Parquet [1], have been developed to deal with extensive data while achieving these goals. However, many of these formats fall short when it comes to keeping up-to-date with the innovations in computer hardware concerning parallelism. In addition to plain multithreading and multiprocessing, modern CPUs support Single Instruction/Multiple Data (SIMD), which facilitates working with more data using a single CPU instruction. However, the CPU is not the only hardware that can be used for data processing: GPUs have recently gained more attention due to their massively parallel compute power and high memory bandwidth [2]. Hence, it is essential to utilize this power of the hardware available to its limits.

Large data file formats support data compression out of the box because it reduces the size of the data that needs to be stored while also speeding up the processing, as less data has to be read back from the storage. If the data is stored on a cloud provider like AWS S3, it can also help to save on cloud provider egress costs. With this in mind, researchers have developed numerous lossless compression algorithms that fall into two categories. *General-Purpose Compression* schemes can compress any data by consuming a byte stream and compressing it by exploiting the patterns at the byte level. These schemes, like Zstd[1], Snappy[2], and LZ4[3], can compress the data well at the cost of compression/decompression speed[4] and are inherently unsuitable for utilizing SIMD or GPUs efficiently [3]. To address the shortcomings of general-purpose compression, *LightWeight Compression (LWC)* schemes have been developed. These often are specialized for a specific data type, like strings or floating-point numbers, and are designed to exploit commonly occurring data patterns. As the name suggests, LWC schemes, like Bit-Packing and Run Length Encoding (RLE), are lightweight, provide extremely fast decompression, and can be parallelized using modern hardware [3, 4]. Since LWC schemes might not offer very high compression ratios, some file formats choose to apply general-purpose compression on top to get manageable data sizes.

While nested data in tables is against the normal forms proposed by Codd [5], they have become a standard feature of big data file formats. These formats emerged in distributed systems at Google and Hadoop clusters, where table joins require expensive data shuffling over a network. Nesting tabular data avoids such joins and ensures that

[1]`https://github.com/facebook/zstd`
[2]`https://github.com/google/snappy`
[3]`https://lz4.org/`
[4]`https://github.com/inikep/lzbench`

the data is processed on the same computer, as the nested data resides in a single file on the computer, where network data placement is typically done at the file level.

Typical nested data type choices are list, struct, or map type to represent complex and hierarchical structures. List type allows users to store more than one value per row, where all the values must be of the same type. With structs, several related columns with possibly different types can be grouped under a single column, given that the subcolumns and their types are known beforehand. Maps consist of key/value pairs similar to structs, but they allow arbitrary keys of the same type, and all the value types must also be the same. These types can be arbitrarily combined to build a more suitable and intuitive schema for representing real-world entities. Modern file formats can help users load and process their data more efficiently by supporting nested data types. Although file formats generally have nested types available as logical types, they are usually unnested and compressed with simple type schemes. This might be due to the lack of research in developing specialized compression schemes for nested types.

FastLanes is a project working towards a next-generation columnar file format. It introduces an innovative data layout for LWC schemes such as Bit-Packing, Delta encoding, and RLE that removes SIMD-unfriendly data dependencies and facilitates incredibly fast decompression on modern CPUs [6]. One of the main goals of FastLanes is to be future-proof as the hardware improves. To achieve this, FastLanes does not use explicit SIMD instructions as it can vary among different CPUs; instead, it only contains scalar code and relies on the compiler to auto-vectorize the code using the best available CPU registers. It has also been shown that the FastLanes layout is suitable for GPU processing and can achieve significant speedups compared to other systems [7].

The FastLanes file format is a work-in-progress project utilizing the FastLanes layout that will be open-sourced upon release. It supports efficiently storing many simple data types, such as integers and strings. Support for the nested data types has been added as a part of this work.

1.1 Research Questions

This work aims to study the usage of nested data types in practice and analyze their exploitable data patterns for compression schemes suitable for FastLanes. The following questions will be answered in this thesis:

1. How is nested data structured in real-world use cases? What is a suitable dataset that can be used to benchmark existing file formats and their nested type encoding schemes?

2. Is there a need for specialized nested data-type encoding schemes in file formats?

3. What are the possible encoding schemes for nested data types that facilitate fast decompression?

4. How much compression improvement can FastLanes get with specialized nested data type encoding schemes?

1.2 Outline

The rest of this thesis is structured as follows. Chapter 2 briefly describes the previous work done in file formats and compression and covers the necessary background information for the following chapters. A new compression dataset for benchmarking nested data compression is introduced in Chapter 3. Chapter 4 introduces new compression schemes designed for nested data types. The new schemes are evaluated on the dataset against other FastLanes compression schemes and file formats in Chapter 5. Finally, Chapter 6 summarizes the work done in this thesis and lists potential improvements as future work.

2 Background

This chapter will first discuss the existing LWC schemes and some common ways of representing nested data types. Then, we will consider some of the existing large data file formats and their supported compression schemes.

2.1 Lightweight Compression / Encoding

There are two main ways of storing tabular data: Row Store and Column Store. In Row Stores, the data is organized by rows, meaning all the values of a row come before the next row. This layout benefits transactional systems as row-level operations such as insert and delete only require reading from/writing to a consecutive disk region. However, data compression becomes more challenging as the exploitable data patterns are often local to a column, but the values of a single column are split across rows. Column Stores addresses this problem by organizing the data in columns, meaning that the values of a single column are directly next to each other before the following column's values. Data locality from a single distribution facilitates data compression and is suitable for analytical systems where data updates are rare. [8] shows that the columnar layout is also CPU-efficient for executing certain queries.

This section briefly describes various LWC schemes for columnar storage. In the computations below, n is the number of values in the original column, and each value requires b bits to be stored uncompressed. Hence, the uncompressed size is $n * b$ bits. The compression ratio is a frequently used indicator of the effectiveness of a compression scheme, which is defined as $\frac{\text{uncompressed size}}{\text{compressed size}}$ - a higher compression ratio means better compression.

2.1.1 Bit-Packing

The idea of bit-packing is using only as many as required to store the values. For instance, if the column type is a 16-bit unsigned integer, but the largest value of the column fits 5 bits, then all column values can be stored using 5 bits per value, significantly reducing the total size. This scheme requires storing the packed bit-width as the metadata and bit-packed values. It is often combined with the other LWC schemes. If values can be bit-packed to m bits per value and assuming 8-bit bit-width metadata, then only $8 + n * m$ bits are required to encode the column. Table 2.1 shows bit-packing applied to an integer column.

Table 2.1: Bit-packing of 4 8-bit integers (4 bytes) to 4 bits per value (2 bytes).

Original Column (Decimal)	6	2	9	7
Original Column (Binary)	00000110	00000010	00001001	00000111
Bit-Packed Column (Binary)	0110	0010	1001	0111

2.1.2 Run Length Encoding

The RLE scheme is most useful for columns where consecutive values often repeat. The idea behind RLE is that rather than storing repeating values multiple times, it is enough to store the repeating value and the number of times it is repeated. These sequences of repeating values are called *run*s. An example application of RLE is shown in Table 2.2. The number of runs needs to be stored in addition to the data. If a column has r runs and assuming both the number of runs and run-lengths fit 32 bits, it can be encoded in $32 + r * (b + 32)$ bits. Both RLE encoding and decoding can be accelerated using SIMD instructions, which is known as *Vectorized RLE* [9].

Table 2.2: RLE encoding of a string column. 8 string values are encoded in 4 runs as length-value pairs.

(a) Original String Column

"A"	"A"	"A"	"B"	"B"	"C"	"A"	"A"

(b) RLE runs

Run Value	"A"	"B"	"C"	"A"
Run Length	3	2	1	2

2.1.3 Dictionary Encoding

Dictionary encoding is effective for any column with low uniqueness. The idea is to map each unique value to a unique integer called *dictionary code* and store this mapping as a dictionary. Then, the encoded data is the sequence of *code*s, which is usually much smaller to store than the original data values. Integer *code*s are bit-packed to save even more space. If a column has u unique values, bit-packed codes are c bits wide, and assuming the number of elements in the dictionary (u) fits 32 bits, then the dictionary requires $32 + u * b$ bits, and the encoded data requires $n * c$ bits. An example application of dictionary encoding is shown in Table 2.3.

2.1.4 Delta Encoding

Suppose an integer column consists of numbers close to each other. In that case, Delta encoding can be applied to store the relative *delta*s between each value and the preceding

Table 2.3: Dictionary encoding of a string column with 8 values. Codes can be bit-packed to 2 bits as the largest code is 2.

(a) Original String Column

"A"	"A"	"A"	"B"	"B"	"C"	"A"	"A"

(b) Dictionary Encoded Column

0	0	0	1	1	2	0	0

(c) Dictionary

Value	**Code**
"A"	0
"B"	1
"C"	2

value, which can be much smaller than the original column values. Bit-packing is usually used to exploit these smaller deltas and store them using as little bits as required. Table 2.4 shows an example of this encoding. Decompressing Delta-encoded columns requires adding the delta of the current value with the preceding value. A disadvantage of this technique is that random access to a value inside the column is impossible unless the prior value is known. Hence, finding out a value might require summing up multiple deltas. Large columns can be split into numerous smaller blocks to mitigate this problem, as the number of required summations is limited to the block size [10]. [11] proposes various prefix sum implementations using SIMD instructions that can speed up Delta decoding.

Table 2.4: An example of Delta encoding on an integer column. Deltas can be bit-packed to 3 bits.

Original Column	2	5	6	6	13
Deltas	2	3	1	0	7

2.1.5 Patching / Exception Handling

The encodings discussed so far are applied to all column values without considering their distribution, which can often be skewed in practice. For instance, a string column might have distinct values in 10% of the rows, while the remaining 90% have low uniqueness that is highly dictionary-encodable. Plain dictionary encoding would require assigning dictionary codes to all unique values, including that distinct 10%, which might make the dictionary codes require more bits than needed. *Patching* can be used to leave out the outlier values and store them as uncompressed exceptions to mitigate the problem [3]. For our example, it would mean that the 10% distinct values are stored uncompressed, while the dictionary only contains values from the remaining 90%. Keeping the number of exceptions low is essential since they are stored as uncompressed values.

2.1.6 Fast Static Symbol Table

Fast Static Symbol Table (FSST) is an LWC scheme designed for strings [12]. Users prefer to store their data as strings because of its suitability for storing any data, even if more specialized data types exist. This trend holds for our dataset explained in Chapter 3 and is confirmed by the Insight 2 of [13]. Therefore, there is a real need to have specialized fast string encoding algorithms like FSST.

FSST works similarly to Dictionary encoding; however, rather than assigning codes to complete strings, it detects commonly occurring substrings called *symbol*s and replaces them with an 8-bit character code. Each *symbol* is limited to 8 bytes for performance reasons. The dictionary (called *symbol table*) is also limited in size to at most 256 entries to fit the CPU cache and ensure code bit-widths do not exceed 8 bits or character size. One of the beneficial properties of FSST is that the encoded values remain as a string containing only the codes. It has an exception-handling mechanism for cases where some characters are not frequent enough to be worth occupying a *symbol table* entry. In those cases, it uses an escape byte before the original byte. Another great property of FSST is that it allows random access to compressed strings compared to general-purpose block-based compression algorithms. Figure 2.1 shows an example application of FSST.

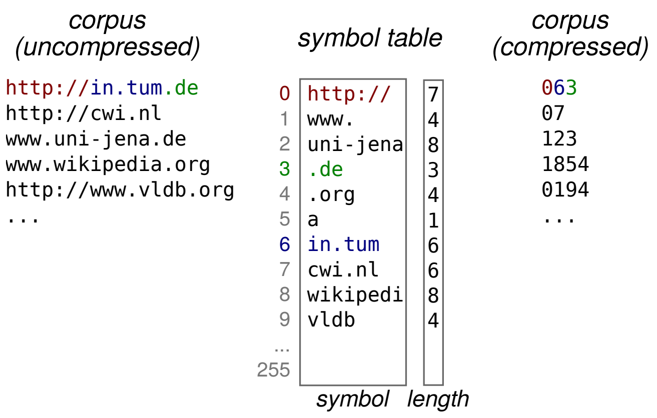


Figure 2.1: Example of applying FSST to a string column containing URLs [12]. Frequently occurring symbols of at most 8 characters are put in the symbol table and replaced with an 8-bit code as a character in the resulting string. The lengths of the strings are greatly reduced after this replacement.

FSST provides a compression ratio of 2 on average. It has an alternate implementation called FSST12[1] that uses 12-bit codes and 4096 *symbol table* size. It does not have the exception handling mechanism described above because the first 256 entries of the *symbol table* are always single-byte symbols. Although the original FSST usually provides better compression ratios, FSST12 can be better on more chaotic string columns with non-English text.

[1]https://github.com/cwida/fsst/blob/master/fsst12.h

2.1.7 Cascading Encoding

LWC schemes often specialize in exploiting a certain kind of data pattern. Hence, combining multiple schemes and applying them one after the other often makes sense. For instance, a string column can first be encoded with a Dictionary encoding, which replaces all strings with an integer code, and then the codes can be compressed with RLE. In this case, Dictionary encoding takes care of the duplicate values, while RLE specializes in consecutive repeating values. Such application of multiple schemes is called Cascading encoding [9]. Combining more than two schemes - Multi-Level Cascading encoding is also possible and can compress the data better by exploiting more patterns of a column [14].

2.1.8 Null Value Handling

It is prevalent for data to contain null values to indicate a missing value. The null values need to be specially handled during encoding because they usually cannot be represented as a value from the column's domain without the possibility of colliding with another non-null value. For instance, if the column type is an integer, it can contain all values from the integer domain with the addition of a null value. Hence, null values are usually encoded separately from the data and are considered equal to another value from the column domain for data encoding purposes.

One standard way of handling null values is storing a boolean value for each value to represent whether it is null. These boolean values are known as *Nullmap*, *Presence*, or *Validity*, and the meaning of true/false values can vary depending on the system. These values can be considered as any other column of boolean type and, hence, can be encoded similarly.

After the nullmap is stored, all null values must be eliminated from the data for other LWC schemes. The choice of this value does not matter from the correctness perspective; however, it might affect the effectiveness of the LWC scheme. For instance, for an integer column, it is possible to replace all null values with 0; however, doing so might affect the data locality and negatively impact the RLE scheme by introducing new runs of length 1 with the value 0 and dividing other possible runs at null values. Hence, it is usually better to consider all null values equal to the column's previous non-null value.

2.2 Nested Type Representation Methods

Due to their complex nature, handling nested data types requires more attention than handling simple types. For instance, lists traditionally require encoding the inner data and the offsets separately. Null values are also more complicated since they can appear at different levels. A list can be equal to null, showing the missing value for a row, or the list might contain a mixture of null and non-null values. The possibility of arbitrarily nesting lists, structs, and maps makes the nullability and list cardinality problem more challenging. Multiple representations have been developed to overcome these challenges.

First, we'll define the nested types that are the focus of this thesis. Then, we'll consider possible representations of these types. Figure 2.2 describes the representation of data types for schema figures in this section.

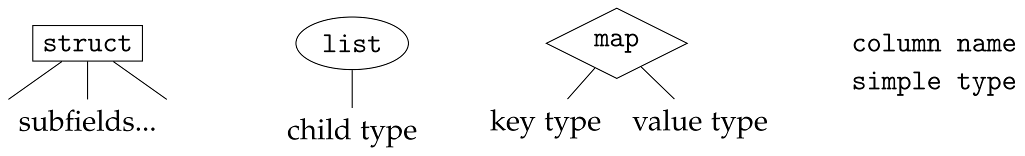


Figure 2.2: Schema representation.

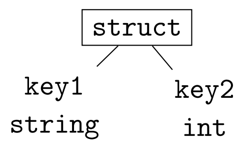
2.2.1 Nested Data Types

Struct Type

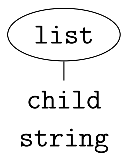
Users can group multiple columns under a single struct-typed column in the schema. Each subfield of the struct must be defined in the schema by its name and type. Subfields can be of different types. The value of a struct can be equal to null, which is considered different than all subfields being null. Figures 2.3a and 2.3b show an example struct value in JavaScript Object Notation (JSON) representation with its schema.

List Type

The list type lets users store multiple values in a single row. All child items in the list must be of an equal type, which is defined in the schema. Lists can be null, empty, or contain a mixture of null and non-null values. An example list value with the corresponding schema is given in Figures 2.3c and 2.3d.



(a) Struct schema.



(c) List schema.

```
{
    "key1": "val1",
    "key2": 42
}
```

(b) Struct Value.

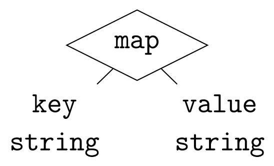
```
[
    "val1",
    "val2"
]
```

(d) List value.

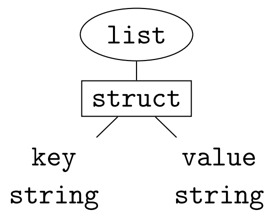
Figure 2.3: Example schema and JSON value for struct and list types.

Map Type

Most systems typically support the map type as a logical type but physically handle it as a combination of list and struct types. This is because the map type has properties similar to list and struct types. A map value can contain arbitrarily many key-value pairs, where all the keys and all the values are the same type, respectively. Hence, a map can be represented as a list of structs where each struct has key and value subfields. The top-level list type provides the ability to contain arbitrarily many values, while the struct type allows mapping each key-value pair of the map to a struct value inside the list. Figure 2.4 shows an example of this mapping from a map type to a list of structs. Note that the order of structs in the list is arbitrary.



(a) Map schema.



(c) Equivalent schema with list and struct types.

```
{
    "key1": "val1",
    "key2": "val2"
}
```

(b) Map value.

```
[
    {"key": "key1", "value": "val1"},
    {"key": "key2", "value": "val2"}
]
```

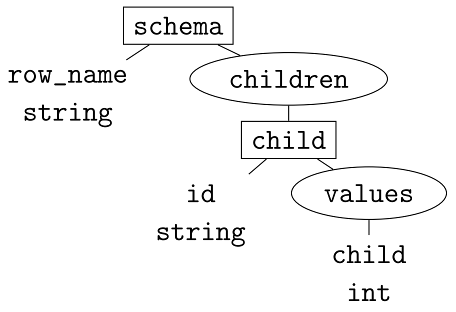
(d) Representation of the map value as a list of structs.

Figure 2.4: Example mapping from a map value to list and struct combination. Each key-value pair in the map is converted to a struct with key and value subfields. These struct values are then put in a top-level list.

2.2.2 Length & Presence Representation

The Length & Presence Representation [15] is a trivial way of storing nested data. The data is flattened to a columnar format where each nested and simple column is stored with a nullmap, as discussed in Subsection 2.1.8. Struct types do not require additional handling as only the nullmap and the child columns are enough to reconstruct them. However, list types also require storing each list's length or offset as an integer since child values are concatenated in the columnar format. This integer column can be encoded with any other integer LWC scheme.

Consider the example nested table in Figure 2.5. Table 2.5 shows the Length & Presence representation of that table.



(a) Schema.

```
{
  "row_name": null,
  "children": null
}
```

(b) Row 1.

```
{
  "row_name": "Row 2",
  "children": []
}
```

(c) Row 2.

```
{
  "row_name": "Row 3",
  "children": [
    null,
    {
      "id": null,
      "values": null
    },
    {
      "id": "3.2",
      "values": []
    },
    {
      "id": "3.3",
      "values": [null, 3]
    }
  ]
}
```

(d) Row 3.

Figure 2.5: Example 3-row nested table with schema list->struct->list containing null and non-null values at all possible schema levels.

Table 2.5: Length & Presence representation of the example table from Figure 2.5. Nulls are replaced with empty string or 0, depending on the column type. The presence value `false` indicates the value is null.

(a) `row_name`

Presence	Value
false	""
true	"Row 2"
true	"Row 3"

(b) `children`

Presence	Length
false	0
true	0
true	4

(c) `children.child`

Presence
false
true
true
true

(d) `children.child.id`

Presence	Value
false	""
false	""
true	"3.2"
true	"3.3"

(e) `children.child.values`

Presence	Length
false	0
false	0
true	0
true	2

(f) `children.child.values.child`

Presence	Value
false	0
true	3

2.2.3 Google's Dremel Representation

Dremel is an extensive dataset analysis system developed by Google [16]. In this work, we are interested in its representation of nested data, a novel idea at the time. However, Dremel made other significant contributions, such as a new in-place query execution and applying data structures designed for web search engines to database systems.

In the Dremel representation, only the leaf nodes in the schema are stored, while the nested types higher up in the schema are ignored from the storage perspective. Each leaf node value (simple-typed) is assigned a level metadata to make lossless original data reconstruction feasible. The *Definition Level* of a value is the number of non-null fields in the data path of the value that could have been null. This allows Dremel to find the exact schema level of each null value. Table 2.6 shows the Dremel representation of the example in Figure 2.5. Considering the `row_name` column, the first row has a Definition Level of 0, meaning no field in the column's path has a non-null value. This is true because the `row_name` value of the first row is null and is not nested inside any other column. Meanwhile, the second row has a non-null value. Hence, the Definition Level equals 1; one column, namely `row_name` itself, has a non-null value in the column's path. As a more complicated example, let's consider the column `children.values` with empty list value in Line 11 of the third row. Dremel has a null value in the `children.values` column with the Definition Level of 4, even though that list contains no nulls to represent this empty list.

Additionally, Dremel uses integer *Repetition Level* to specify the repeatable schema level (for our nested types, list level) of each value. In combination with the Definition Level, the Repetition Level of each value clarifies the list it belongs to in case it is nested inside multiple levels of lists. The first value of the list always has its Repetition Level equal to the number of lists it is contained in, not counting the list itself. The following values of the list have a Repetition Level of one more than the first element. For instance, the value in Line 4 of the third row is the first element in the list, and the column `children` is not contained in any other lists; hence, the Repetition Level of that list is 0. However, the following values, like the one from Line 6 of the third row, have a Repetition Level of 1 since they are repeated inside the `children` list. Similarly, the first value in Line 15 of the third row has a Repetition Level of 1; since it is the first value of the `children.values` list, and higher up in the schema, there is one list (`children`). The Repetition Level of the second value of the same list is two since it is repeated in the hierarchy's second list.

Dremel's Definition and Repetition Levels represent the nested data with two additional integer columns. As can also be seen from the example, the levels are usually very small and highly repeating, making them a perfect fit for Bit-Packing and RLE. The columnar data can be compressed using any other suitable LWC scheme. Dremel representation has the advantage that the complex nested structure of the data gets flattened into a columnar representation where all the nested types in the schema are removed, and only the simple-typed leaf nodes with two additional highly compressible integer columns are stored. Storing only simple-typed columns decreases the number

Table 2.6: Dremel representation of the example table from Figure 2.5. The meaning of the level values is shown in parentheses.

Column	Value	Definition Level (Meaning)	Repetition Level (Repeating Column)
`row_name`	`null`	0 (`row_name = null`)	0 (no list value)
	`"Row 2"`	1 (`row_name != null`)	0 (no list value)
	`"Row 3"`	1 (`row_name != null`)	0 (no list value)
`children.id`	`null`	0 (`children = null`)	0 (no list value)
	`null`	1 (`children = []`)	0 (no list value)
	`null`	2 (`children.child = null`)	0 (first value - no repetition yet)
	`null`	3 (`children.id = null`)	1 (`children`)
	`"3.2"`	4 (`children.id != null`)	1 (`children`)
	`"3.3"`	4 (`children.id != null`)	1 (`children`)
`children.values`	`null`	0 (`children = null`)	0 (no list value)
	`null`	1 (`children = []`)	0 (no list value)
	`null`	2 (`children.child = null`)	0 (first value - no repetition yet)
	`null`	3 (`children.values = null`)	1 (`children`)
	`null`	4 (`children.values = []`)	1 (`children`)
	`null`	5 (`children.values.child = null`)	1 (first in `children.values`)
	`3`	6 (`children.values.child != null`)	2 (`children.values`)

of columns that need to be read during query processing. However, some information about nested types might be stored multiple times in the child columns. For instance, information about a null struct value is stored in all struct subfields, leading to redundant data storage and increased file size. Unlike Length & Presence representation, reconstructing the original data structure is also more involved, which might slow down the query execution depending on the query.

2.3 Open Big Data File Formats

Big data file formats are designed to store large amounts of data efficiently by employing compression. This section covers such widespread open-source file formats, analyzes their compression infrastructure, and describes their representation for nested data types.

2.3.1 Apache Parquet

Apache Parquet [1] is the most popular file format for storing data efficiently. It is an open-source columnar file format that was first released in 2013. Over the years, the format has been upgraded various times, the latest version being 2.10.0, although v1 is still widely used in practice for maximum compatibility.

The Parquet format stores the data in the Partition Attribute Across (PAX) layout [17]. The tabular data is first horizontally distributed into *Row Group*s - a logical collection of rows. The row group size, either in bytes or in the number of rows, is usually a configurable parameter in the writer implementation and is not defined by the format. Each row group is then split into *Column Chunk*s, which contain the data of a particular column in the row group. Each column chunk, in turn, is split into one or more *Page*s, which is the smallest compressible data unit in Parquet.

Parquet uses Dremel representation to store the nested types. It skips storing the Definition Levels if a column is mandatory (it is not inside a list, and it is not nullable, including parent nested columns) since, in that case, all Definition Levels would be equal to the depth of the column in the schema. Similarly, Repetition Levels are omitted if the column is not nested inside another one. The level data is always encoded in the latest Parquet version with a hybrid RLE/Bit-Packing scheme.

The hybrid RLE/Bit-Packing scheme works similarly to plain RLE, except that the chaotic, non-repeating parts of the data are encoded with Bit-Packing. Parquet also supports Dictionary encoding for all column types, where the integer codes are then encoded with hybrid RLE/Bit-Packing. Additionally, Parquet v2 supports many other LWC schemes for compressing the data pages. Delta encoding on integer columns is supported. For storing strings, in addition to Delta encoding the string lengths, Parquet also supports Incremental Encoding, where each string is encoded as the length of the longest common prefix with the previous string and the differing suffix. Interestingly, Parquet does not support any specialized encodings for floating-point numbers. Instead, it has a mechanism called Byte Stream Split, which reorders the bytes to make it more suitable for general-purpose compression schemes.

As the supported LWC schemes usually do not provide satisfactory compression ratios, the Parquet file format optionally allows the users to apply general-purpose compression to the data block inside pages. However, as discussed previously, doing so negatively impacts the decompression and scan performance. The custom LWC schemes might also hurt the decompression speed. For instance, the hybrid RLE/Bit-Packing scheme requires branches in the decompression path to decide if the next run is encoded as an RLE or Bit-Packed run, which is SIMD and GPU-unfriendly.

2.3.2 Apache ORC

Apache ORC [18] is another open-source columnar file format released in 2013. Similar to Parquet, it employs the PAX layout, although the terms differ. An ORC file is partitioned into *Stripe*s (similar to Row Groups). Unlike Parquet, ORC fixes the byte size of stripes to ensure a low memory footprint, even for wide tables, at the risk of not having enough rows to utilize vectorized execution [19]. Inside stripes, data is stored as *Stream*s where a single column is represented as multiple streams (data, presence, length).

ORC employs Length & Presence representation for nested data types. The presence stream is a boolean stream that is omitted if all the values in a stripe are non-null.

Like Parquet, ORC supports several custom LWC schemes. Boolean streams are converted to bytes where each bit represents a value from the stream, then encoded via a modified RLE where runs shorter than three values are stored as a literal list of values, and the first byte determines how the following bytes should be interpreted. A similar scheme with four sub-encodings determined by the first byte is used for integer columns. Unlike Parquet, only string streams can be Dictionary-encoded. There is no support for floating-point type encoding, as in Parquet.

ORC also supports general-purpose data compression; however, it is done in chunks of configurable size (256KB by default), usually smaller than a stream. This allows readers to omit reading more data if a chunk can be skipped, but the misalignment between the chunks and the streams makes it more difficult for query processors. Regarding the decompression performance on modern hardware, ORC faces drawbacks similar to those of Parquet.

2.3.3 DuckDB Database Format

DuckDB [20] is an analytical in-process database designed to be fast, reliable, and easy to use, offering many clients for different environments. It has a custom database format that stores the data as compressed columns. The format is similar to Parquet as it splits the data into fixed-size *Row Group*s containing 120K rows each, which are stored as multiple *Column Segment*s.

DuckDB handles nested types using Length & Presence representation. As one of the design goals of DuckDB is query execution speed, it only uses LWC schemes to compress the column segments, facilitating fast data access and query execution. It supports a rich set of LWC schemes, such as Bit-Packing, RLE, Dictionary, and FSST, among others[2]. DuckDB first analyzes the column segment during compression to pick the best suitable LWC scheme, then compresses the column in fixed-size blocks. Blocks are crucial as DuckDB supports ACID transactions that might update or delete data from compressed blocks.

2.3.4 FastLanes

FastLanes is a next-generation file format currently being developed at CWI. Its main goal is to address the shortcomings of the other file formats, mainly in terms of compression ratio and decompression speed. To achieve this goal, FastLanes uses data-parallel compression layouts for a higher decompression speed and implements Cascading encoding for a higher compression ratio. Note that the underlying compression layouts described below are already completed and can be found at the FastLanes GitHub repository[3], whereas its file layout is still being worked on.

Interleaved Bit-Packing

Optimizing bit-packing and unpacking is crucial as it is an essential part of many other LWC schemes. In a standard implementation, bits of adjacent values are tightly packed

[2]`https://duckdb.org/2022/10/28/lightweight-compression.html`
[3]`https://github.com/cwida/FastLanes`

next to each other. However, this prevents efficient utilization of vectorization because the adjacent values fall into the same SIMD lane. Researchers have proposed *interleaved* or *vertical* bit-packing layouts to prevent this problem and make bit-unpacking SIMD-friendly [10, 21]. However, these layouts have been designed with a specific SIMD lane width in mind, meaning they cannot take advantage of the larger SIMD registers on modern CPUs, as the layouts for thinner registers do not have enough data parallelism to run efficiently on wider SIMD registers. It has been shown that the layouts designed for larger registers are generally more efficient [22].

Since layouts designed for larger registers can efficiently map to smaller registers by using multiple thinner SIMD instructions, the interleaved layout of FastLanes [6] is designed for 1024-bit SIMD width, which currently does not exist in any CPU. This design choice lets FastLanes be future-proof while efficiently supporting all existing CPUs. The implementation of this layout is fully scalar C++ code without any explicit SIMD intrinsic calls. FastLanes relies on the compiler auto-vectorization to pick the widest available SIMD registers. Figure 2.6 shows this interleaved layout in action.

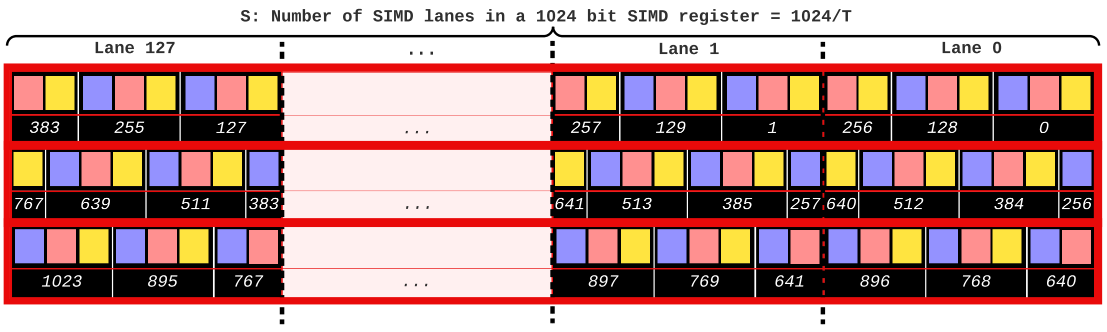


Figure 2.6: Interleaved bit-packing layout of FastLanes [6]. This example shows 3-bit 1024 values (indexed 0 to 1023 in black boxes) bit-packed round-robin over 128 8-bit SIMD lanes.

Unified Transposed Layout

In addition to the optimized bit-packing layout, FastLanes proposes the *Unified Transposed Layout* [6] as a row ordering technique that facilitates SIMDized decoding for many LWC schemes with data dependencies, like Delta and Dictionary encodings. This layout removes sequential data dependencies by ensuring each SIMD instruction can operate on independent values. This layout is a generic solution independent of the SIMD lane width. An example of this layout is shown for Delta decoding in Figure 2.7.

RLE decompression is challenging to vectorize via SIMD as it requires two nested loops to loop over runs and then the length of the run. FastLanes proposes a modified *FastLanes-RLE* scheme that maps RLE to a combination of Dictionary and Delta encoding, both of which support the Unified Transposed Layout. Rather than storing run value-length pairs, all run values are stored in a dictionary where run values, with possible

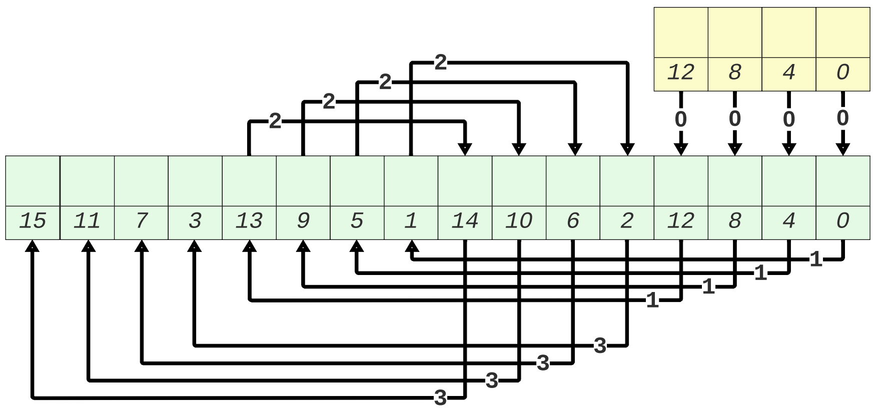


Figure 2.7: Unified Transposed Layout of FastLanes for data-parallel Delta decoding [6]. We start with four base values (yellow boxes) instead of a single one to allow 4-way data-parallel additions. 16 values, whose indices are shown on smaller boxes at the bottom, are decoded in 4 SIMD operations. This layout is also data-parallel for thinner data types.

duplicates, are mapped to the index of the run. The dictionary-encoded column only consists of monotonically increasing integers, which can be Delta-encoded with 1-bit deltas.

Compression Infrastructure

FastLanes file format supports cascading encoding of integer, string, and floating-point number columns to achieve high compression ratios while only using fast LWC schemes. Like other formats described above, the data is split into *Row Group*s of 64K rows and compressed in columnar chunks using vectorized compression algorithms. A vector in FastLanes consists of 1024 column values.

Within the scope of this thesis, support for nested data types is added by employing the Length & Presence representation. More details on implementation are provided in Section 5.1.

3 RealNest Dataset

Before diving deep into compressing nested data types, it is crucial to create a benchmark where the compression schemes can be comparatively evaluated. Although there are many existing datasets for benchmarking simple type compression, like the synthetic TPC-H and TPC-DS [23] and the user-generated Public-BI benchmark [24], to the best of my knowledge, there is no compression corpus with a focus on nested data types. The user-generated RealNest dataset [25] has been created to address this shortage and provide a common nested data compression evaluation ground for this and future work.

3.1 Data Collection

The RealNest dataset is a collection of nested data from publicly available practical datasets. The data in the RealNest dataset originates from various sources and domains described in this section.

3.1.1 Parquet Files

Open Data on AWS

The first source for the RealNest dataset is Open Data on AWS[1]. It is a collection of publicly available datasets via AWS resources. The Registry[2] allows users to search for keywords in these datasets. Since Apache Parquet is a common file format for sharing large datasets that also supports nested types, searching for `"parquet"` brings matching datasets and S3 buckets that likely contain nested data in large quantities.

After the initial filtering, it is necessary to check the S3 buckets to identify the Parquet files with nested data and potentially group multiple files if they have the same schema to extract all possible tables without duplication. A Python script was developed to achieve this. This script listed all the files in a given S3 bucket, and for each file, DuckDB was used to try to get the Parquet schema and filter out nested columns. Note that although Parquet files commonly have `.parquet` extension, some buckets have Parquet files with no extension. Hence, to account for this, the script considered all the files and relied on DuckDB errors to exclude non-Parquet files. Thanks to Parquet having a consistent structure that allows reading a file partially to get its metadata, S3 supporting partial file downloads via HTTP `Range` header[3], and DuckDB having efficient

[1]Open Data on AWS: `https://aws.amazon.com/opendata`
[2]Registry: `https://registry.opendata.aws/`
[3]HTTP `Range` requests: https://datatracker.ietf.org/doc/html/rfc9110#name-range-requests

implementation to take advantage of this, getting the schema of a Parquet file did not require downloading the whole file, substantially speeding up the entire process. The script provided a list of different schemas and a list of S3 object keys representing the Parquet files for each schema.

The results of the previous script made it clear that grouping Parquet files with their schema was not a perfect approach to grouping up and, hence, eliminating duplicate tables. Manual inspection showed that sometimes, although two files should belong to the same table, their schemas were slightly different. For instance, if all column values are null, this column would sometimes be completely missing from the schema. Hence, the previous script assigned it to be from another table. The results are manually cleaned to fix these issues and ensure the tables are unique, and a final list of tables and object keys are extracted. Since the object keys followed a consistent structure where the table name could be inferred from the key, keys are reduced to a Regex string to simplify the results. This metadata about the Parquet files can be found in the `parquet_metadata.json` file[4] of the RealNest repository.

The above process produced 28 tables in the RealNest dataset from 4 S3 buckets. The tables are listed in Table 3.1.

CERN Open Data

CERN Open Data is a portal for hosting Particle Physics data. The Institute for Research and Innovation in Software for High Energy Physics (IRIS-HEP) has a public benchmark for Analysis Description Languages (ADLs), which is a combination of nested data from a CERN Open Data entry [26] about an experiment in 2012 and nested queries for that data. The original data is in domain-specific ROOT file format. Still, the RumbleDB team has converted the data to Parquet format [27, 28]. This data is also included in the RealNest dataset as one of the tables after processing it, similar to the procedure described above.

3.1.2 JSON Files

Another common human-readable file format used to store nested data is JSON[5]. Similar formats like JSON Lines (JSONL)[6] and Newline Delimited JSON (NDJSON)[7] are often used for large data storage since they can store multiple rows per file where a newline character separates each row. Frequently, people compress the large JSON files, for instance, using Gzip[8], to get more manageable file sizes.

While Parquet files have a consistent structure, allowing easy access to metadata information such as the schema, JSON files do not have a way of getting the schema. In

[4]`https://github.com/cwida/RealNest/blob/main/scripts/parquet_metadata.json`
[5]https://json.org
[6]https://jsonlines.org
[7]https://github.com/ndjson/ndjson-spec
[8]https://gnu.org/software/gzip

Table 3.1: List of RealNest Parquet tables from Open Data on AWS.

Origin (S3 Bucket & Prefix)	Table Name
aws-roda-hcls-datalake/clinvar_summary_variants	gene_specific_summary hgvs4variation submission_summary
aws-roda-hcls-datalake/gnomad	sites
aws-roda-hcls-datalake/gtex_8	rnaseqcv1_1_9_gene_tpm rsemv1_3_0_transcript_expected_count rsemv1_3_0_transcript_tpm
aws-roda-hcls-datalake/opentargets_latest	aotfelasticsearch cooccurrences diseasetophenotype epmccooccurrences failedcooccurrences failedmatches interaction interactionevidence knowndrugsaggregated matches
aws-roda-hcls-datalake/thousandgenomes_dragen	var_partby_samples
aws-public-blockchain	btc-transactions eth-logs
daylight-openstreetmap	osm_elements osm_features
overturemaps-us-west-2	admins base buildings divisions places transportation

JSON, values can be of the following types: string, number, object, array, `true/false`, `null`. There are some problems with this representation. In particular, both map and struct nested data types are represented as objects in JSON, making them indistinguishable from each other without referring to the documentation of the original data source. Similarly, date, time, and timestamps are stored as strings. Additionally, it is impossible to determine the entire schema of a JSON file without reading it completely. Thankfully, DuckDB has some heuristics that help acquire the JSON file schema.

DuckDB JSON Reader

At the time of writing, the latest release of DuckDB is version 1.0.0. Although this version can infer JSON schemas with many heuristics, one essential functionality, namely detecting map types, is missing. In other words, JSON objects are either assumed to be of type struct or stored similarly to a string in a DuckDB `JSON` type if many object keys are inconsistent between multiple rows. This is problematic for the RealNest dataset for two reasons. The main reason is that if the original dataset intends to have a map type and is represented as a struct or JSON string in RealNest, the original dataset is modified. It can no longer be considered to represent the way people store their data. The second reason is that people use map type when the subfield names are unknown beforehand. This allows storing any key inside the map. However, if this map is interpreted as a struct, each possible subfield name must be defined in the schema, sometimes resulting in a schema size blow-up. Since RealNest relies on DuckDB to infer the JSON schema, I implemented some heuristics in DuckDB JSON reader to tackle this map-type inference problem.

As mentioned above, DuckDB had a heuristic to resort to the `JSON` type when objects had inconsistent keys. Since inconsistent keys can be considered to indicate the map type, my implementation infers map type in this case. In JSON, all object keys are of string type. Hence, the key type of the inferred map is a string. However, inferring value type is more tricky since JSON allows arbitrary nesting of values. My implementation recursively merges all the different value types seen in the JSON file to determine the correct value type of the map.

Since the check for inconsistent keys is imperfect, it was possible to get struct types with many keys in the schema. A large number of keys is also typical for map types. Hence, in this case, we can also infer the map type instead of the struct type. I added a new configurable option to the JSON reader function that specifies the lower limit for the number of keys to infer a map instead of a struct, with the default value of 25. However, having a large number of keys in an object is not enough to say with certainty that it is a map: since a map requires all values to be of the same type, the value types in the object need to be related to each other. Therefore, map type is only inferred if an object has many different keys and all the value types are at least 80% similar. To check for this similarity, we can merge all value types and get the average similarity of each value type to the merged type.

At the time of writing, my Pull Request for this feature[9] has been merged, and it is expected to be released in the next minor DuckDB version 1.1.0. The rest of this work uses a custom build of DuckDB 1.0.0 with this map inference feature.

RealNest Table from Amazon Berkeley Objects Listings

Amazon Berkeley Objects [29] is a dataset on Open Data on AWS containing information about Amazon products. It contains metadata about each product in nested JSON format. The nested columns of these metadata are included as a table in RealNest.

RealNest Tables from GitHub Archive

GitHub Archive (GHArchive)[10] is a project that records all the public events on GitHub in hourly JSON files. These JSON files contain all the events during that hour. The files have a `payload` field, which has a different schema and meaning depending on the event type. Each GitHub event type is converted to its own RealNest table to avoid mixing rows with different schemas. Only the `payload` field is kept, as it is the only deeply nested field. After this preprocessing, the GitHub Archive provides 11 tables with nested columns to the RealNest dataset.

RealNest Table from Twitter Stream Archive

The Twitter Stream Grab[11] is a public collection of tweets in JSON format from the Twitter API. The RealNest dataset contains this table with the data from the latest uploaded collection with tweets from January 2023.

RealNest Table from CORD-19

The CORD-19 (COVID-19 Open Research Dataset) [30] corpus is a collection of academic papers about COVID-19 and related coronavirus research. It contains the full text of each paper in JSON format, with details like abstract, body paragraphs, bibliography entries, and metadata. Since this dataset contains huge strings deeply nested in the data, it is also included in the RealNest dataset.

All RealNest Tables from JSON Files

Table 3.2 shows all the RealNest tables originating from JSON files.

[9] `https://github.com/duckdb/duckdb/pull/11285`
[10] `https://www.gharchive.org/`
[11] `https://archive.org/details/twitterstream`

Table 3.2: List of RealNest JSON tables from various sources.

Origin	Table Name
amazon-berkeley-objects	listings
cord-19	document_parses
gharchive	CommitCommentEvent ForkEvent GollumEvent IssueCommentEvent IssuesEvent MemberEvent PullRequestEvent PullRequestReviewCommentEvent PullRequestReviewEvent PushEvent ReleaseEvent
twitter-stream	2023-01

3.1.3 Downloading the Data

Since some data sources, like GitHub Archive and `aws-public-blockchain`, are updated frequently with the most recent data available, the RealNest dataset is provided as a Python script that downloads and generates the data. Since the dataset grows daily, the download script allows users to specify the target number of rows they want to download for each table. If a table does not have enough rows in the source, all the available data is downloaded. However, only the latest data that fit the specified row limit is collected for large tables.

A static version of the dataset has also been made publicly available to facilitate standardized comparisons. This version was downloaded in mid-May 2024 and comes in two sizes: $64 * 1024 = 65,536$ and $10 * 64 * 1024 = 655,360$ rows.

The RealNest dataset contains two files per table:

1. `data.jsonl[.gz]` - The actual data in JSONL format (optionally Gzip compressed depending on download script configuration).

2. `schema.json` - The schema of the data in JSON format. The schema is a JSON object with a single key `‘columns’`, containing a list of columns. Each column is a JSON object with 2 or 3 keys:
   - `‘name’` - The name of the column as a string.
   - `‘type’` - The type of the column as a string.
   - `‘children’` - Optional, only exists for nested types. Describes the child types of the nested type as a list of column objects. The list type always has a single

child column with the name ‘child’. The map type always has two child columns with the names ‘key’ and ‘value’.

3.2 Dataset Characteristics

Table 3.3 shows an overview of all the tables in the RealNest dataset (the largest static version). It contains 43 tables with over 23 million rows and 1700 nested columns, and the uncompressed JSONL data is approximately 110 Gigabytes. As discussed in the previous section, the data originates from various real-world use cases such as Genetics, Blockchain, Social Media, Scientific Papers, and Geographic Information Systems.

3.2.1 Schema Analysis

The distribution of various column types is shown in Figure 3.1. Struct columns are used slightly more than the list columns regarding nested types, while map type is comparably rare. The varchar/string type is dominant in simple types since people tend to overuse strings due to their simplicity and capability to contain any other type. The next dominant simple types are bigint and boolean, followed by float, timestamp, double, and integer.

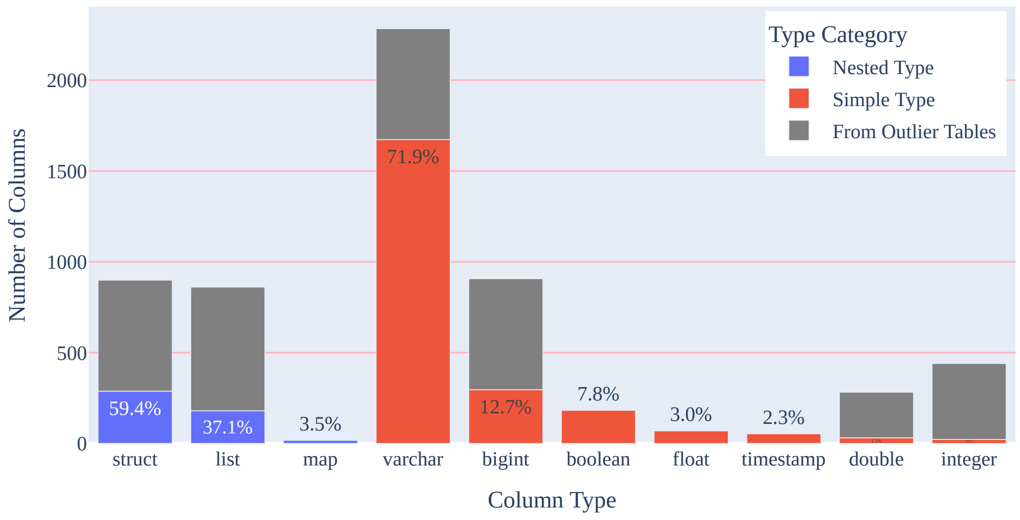


Figure 3.1: Column type distributions in the RealNest dataset. Percentages are calculated per type category. Outlier tables are `gnomad-sites` and `gtex_8-rsemv1_3_0_transcript_expected_count` for the extreme number of nested columns with similar types compared to other tables.

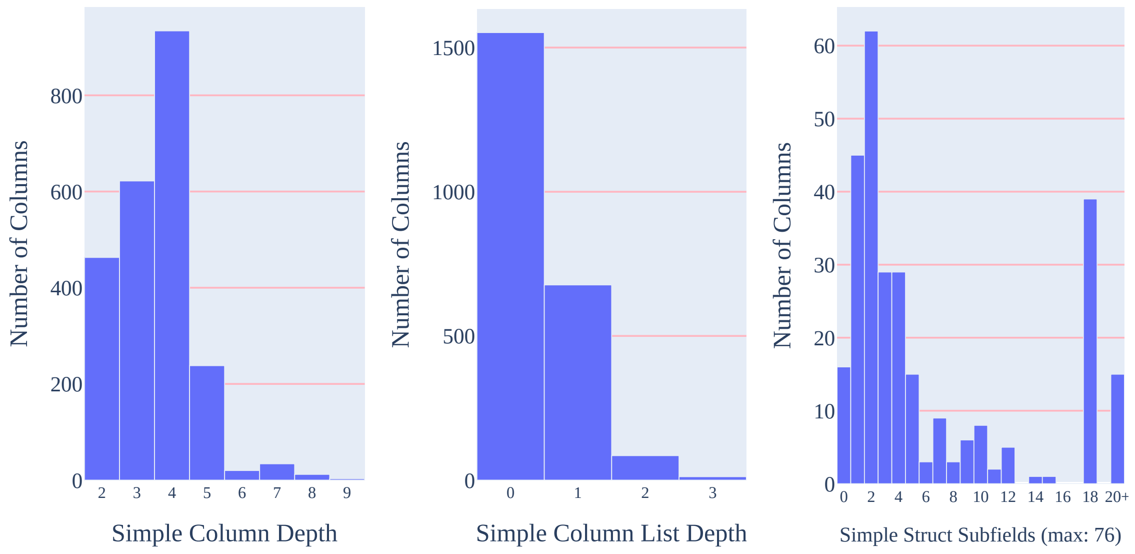
Since this work focuses on nested types, nested type-specific schema properties should also be analyzed. The depth of a column is defined as the number of nested typed columns it is contained in, plus 1 for the column itself. As shown by Figure 3.2a, some simple columns in the dataset are deeply nested by up to 9 levels, although most

Table 3.3: Overview of all RealNest tables with their row, column, and size information.

Origin & Table Name	Rows	Columns		JSONL Size	
		Nested	Simple	Plain	Gzipped
`amazon-berkeley-objects-listings`	147,702	49	52	792.49 MB	76.20 MB
`aws-public-blockchain-btc-transactions`	655,360	5	18	1.05 GB	292.62 MB
`aws-public-blockchain-eth-logs`	655,360	1	1	132.08 MB	21.95 MB
`clinvar_summary_variants-gene_specific_summary`	92,227	5	10	24.44 MB	315.18 KB
`clinvar_summary_variants-hgvs4variation`	655,360	1	2	24.86 MB	79.99 KB
`clinvar_summary_variants-submission_summary`	655,360	1	2	25.00 MB	1.50 MB
`cord-19-document_parses`	401,214	25	44	23.08 GB	5.98 GB
`daylight-openstreetmap-osm_elements`	655,360	7	8	177.28 MB	31.73 MB
`daylight-openstreetmap-osm_features`	655,360	1	2	43.34 MB	7.57 MB
`gharchive-CommitCommentEvent`	177,699	3	40	720.84 MB	80.41 MB
`gharchive-ForkEvent`	655,360	3	95	3.31 GB	242.32 MB
`gharchive-GollumEvent`	299,042	2	6	89.05 MB	16.72 MB
`gharchive-IssueCommentEvent`	655,360	16	165	5.45 GB	967.50 MB
`gharchive-IssuesEvent`	655,360	9	95	2.49 GB	414.16 MB
`gharchive-MemberEvent`	655,360	1	18	590.81 MB	46.33 MB
`gharchive-PullRequestEvent`	655,360	32	377	11.55 GB	1003.17 MB
`gharchive-PullRequestReviewCommentEvent`	655,360	39	420	13.64 GB	1.23 GB
`gharchive-PullRequestReviewEvent`	655,360	37	394	12.40 GB	1.09 GB
`gharchive-PushEvent`	655,360	3	6	258.22 MB	56.82 MB
`gharchive-ReleaseEvent`	655,360	7	73	2.31 GB	219.15 MB
`gnomad-sites`	655,360	681	681	13.55 GB	386.51 MB
`gtex_8-rnaseqcv1_1_9_gene_tpm`	56,200	2	4	6.00 MB	264.73 KB
`gtex_8-rsemv1_3_0_transcript_expected_count`	199,324	611	1,222	6.09 GB	635.38 MB
`gtex_8-rsemv1_3_0_transcript_tpm`	199,324	13	26	132.64 MB	3.54 MB
`hep-adl-ethz-Run2012B_SingleMu`	655,360	13	78	2.07 GB	402.85 MB
`opentargets_latest-aotfelasticsearch`	655,360	9	9	587.05 MB	3.88 MB
`opentargets_latest-cooccurrences`	655,360	1	1	89.57 MB	9.06 MB
`opentargets_latest-diseasetophenotype`	140,823	5	14	58.68 MB	5.97 MB
`opentargets_latest-epmccooccurrences`	655,360	4	5	3.45 GB	400.20 MB
`opentargets_latest-failedcooccurrences`	655,360	3	5	456.46 MB	46.53 MB
`opentargets_latest-failedmatches`	655,360	3	10	295.28 MB	34.50 MB
`opentargets_latest-interaction`	655,360	2	6	103.23 MB	1.04 MB
`opentargets_latest-interactionevidence`	655,360	9	15	210.87 MB	1009.26 KB
`opentargets_latest-knowndrugsaggregated`	272,568	6	6	164.97 MB	23.42 MB
`opentargets_latest-matches`	655,360	1	1	94.29 MB	5.56 MB
`overturemaps-us-west-2-admins`	655,360	8	16	138.19 MB	22.87 MB
`overturemaps-us-west-2-base`	655,360	9	18	171.40 MB	23.57 MB
`overturemaps-us-west-2-buildings`	655,360	8	16	141.08 MB	18.93 MB
`overturemaps-us-west-2-divisions`	655,360	16	24	286.26 MB	42.70 MB
`overturemaps-us-west-2-places`	655,360	22	36	396.33 MB	63.75 MB
`overturemaps-us-west-2-transportation`	655,360	9	17	214.24 MB	29.81 MB
`thousandgenomes_dragen-var_partby_samples`	655,360	14	14	149.33 MB	18.93 MB
`twitter-stream-2023-01`	655,360	81	177	3.90 GB	540.59 MB
Total: 43 Tables	**23,613,003**	**1,777**	**4,229**	**110.77 GB**	**14.35 GB**

simple columns lie on levels 2 to 5. Another important property is the list-only depth of columns since each list level can have more cardinality than the previous level. It is defined similarly to the depth of a column, but only the list columns are counted in the depth calculation. Since maps are also stored as lists, they are considered as a list for this analysis. Figure 3.2b shows the number of simple columns at each list level. Although some simple columns are inside two or even three list levels, most are inside only a single list or none.

Focusing on struct type, we can consider the number of simple-typed columns in a struct. According to Figure 3.2c, most structs only have a few subfields, but they can go up to 70s in extreme cases.



(a) Column distribution by the depth in the schema.

(b) Column distribution by the level of lists it is contained in.

(c) Struct column distribution by the number of simple-typed subfields.

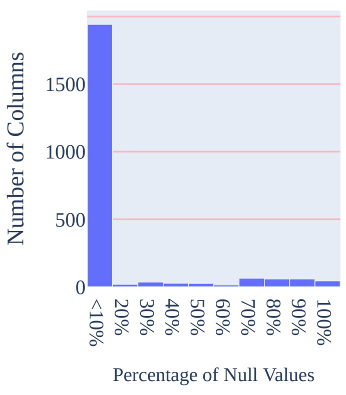
Figure 3.2: Schema analysis focusing on the nested properties, excluding the outlier tables (`gnomad-sites` & `gtex_8-rsemv1_3_0_transcript_expected_count`).

3.2.2 Data Analysis

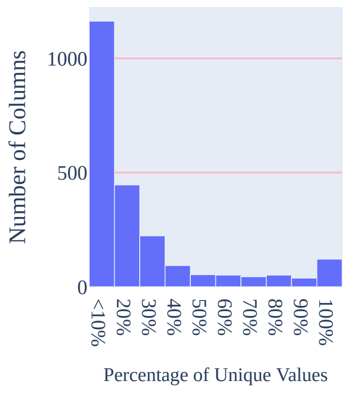
Let's consider the percentage of nulls and unique values for each column. The percentage of null values is defined as $\frac{\text{number of nulls}}{\text{number of all values}} * 100\%$. If a column is inside a list or multiple lists (or, similarly, a map), all the values are combined from all the lists in all rows to count the values. The uniqueness percentage is defined similarly. The number of unique values is defined as the size of the column after removing duplicates. A struct value is a duplicate of another one if, recursively, all the nested fields have equal values in both structs. For list values, two lists are equal if their sizes are equal, and each value in the list is equal to the values from the other list with the same order of occurrence.

As shown in Figures 3.3a and 3.3b, the ratio of null values for most columns is close to 0. However, if we exclude these columns, the trend differs for simple and nested types. While the percentage of null values is evenly distributed for simple types, most are closer to 100% for nested types, meaning that almost all values are null. In terms of uniqueness, both simple and nested-typed columns follow a similar trend, as shown in Figures 3.3c and 3.3d. Many columns have low uniqueness percentages, showing the RealNest dataset's compression potential for both simple and nested types.

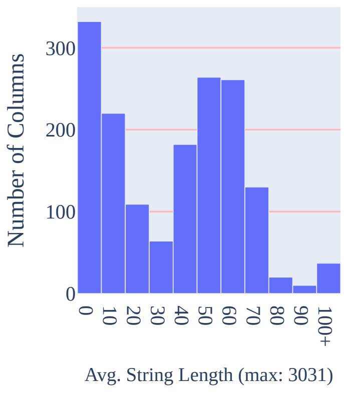
Another interesting dataset attribute is the data length in string and list fields. Maps are considered as a list for this analysis. According to Figure 3.3e, many string columns in the RealNest dataset have less than 100 characters on average, although the average length of some columns exceeds 3,000 characters. Lists are generally much shorter than strings, with most having only a few elements, as shown in Figure 3.3f.



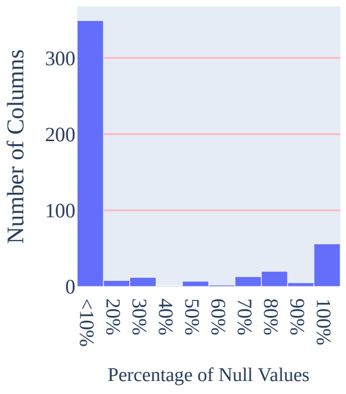
(a) Distribution of simple columns by the percentage of null values.



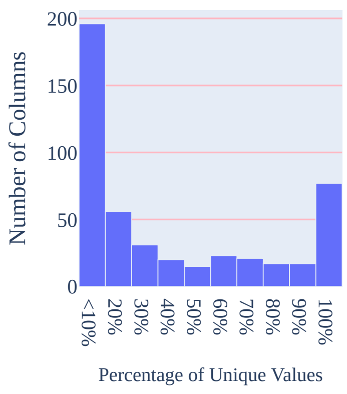
(c) Distribution of simple columns by the percentage of unique values.



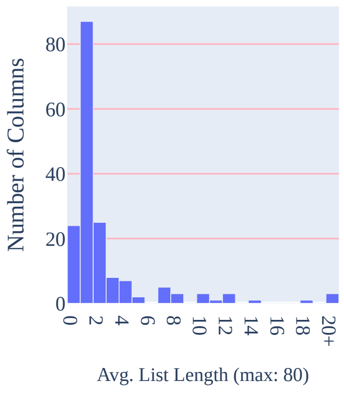
(e) Distribution of string columns by average length.



(b) Distribution of nested columns by the percentage of null values.



(d) Distribution of nested columns by the percentage of unique values.



(f) Distribution of list/map columns by average length.

Figure 3.3: Various data properties, excluding the outlier tables (`gnomad-sites` and `gtex_8-rsemv1_3_0_transcript_expected_count`).

The RealNest dataset is not without its imperfections. Although it contains 43 tables, some tables have data patterns allowing extreme compression ratios (in the range of hundreds), while others are more challenging to compress. The size of a single row also varies greatly amongst tables, as RealNest contains a mix of wide and narrow tables with varying levels of schema depth. Taking special care when reporting compression results is essential to prevent highly compressible tables from dominating the findings. Another factor to remember when using RealNest is the extreme average length of some string columns. Data from some tables, like the `cord-19-document_parses` table with a string column containing the text of each paragraph of academic papers, might be unsuitable for a specific use case. Depending on the solution domain, these columns can be excluded from the dataset to align with the developed solution for a fair evaluation.

4 Nested Data Type Encoding Schemes

Like simple types, it is vital for file formats to store nested data types as efficiently as possible. This chapter will discuss new LWC schemes for nested data types. The focus will be on the list type and the columns inside of a list, as the map type is mapped to a list, and the struct type at the top level is usually used to group multiple related columns together logically in the schema. This is not to claim that the structs are not highly compressible. For instance, one could analyze the correlations inside of a struct and exploit these for compression. However, that is left for future work.

4.1 ListRLE - RLE for List Values

ListRLE is an RLE scheme for lists. It works best when the lists have row locality and repeat consecutively. Table 4.1 shows an example of data from the RealNest dataset as a motivating example for the ListRLE scheme.

In the example data, simple RLE for strings would not be effective as the strings do not repeat, resulting in many runs of length 1. However, considering list-level RLE, we only get four runs for these 11 rows, as displayed in Table 4.2. This application of ListRLE already results in a compression ratio of around 2.5 for this example.

It is possible to achieve more compression by using cascading encoding. ListRLE produces two columns: integer-typed lengths and list-typed values. Further compression can be applied to these columns, like Bit-packing and Dictionary encoding, as shown in Table 4.3. This variation of cascading ListRLE provides a 4.6 compression ratio on the example rows while enabling fast decoding. Note that any other valid combination of LWC schemes could have also been applied.

4.2 ListDict - Dictionary Encoding of List Values

The following scheme, named ListDict, applies dictionary encoding to lists rather than simple types, which is effective if the list column contains many duplicate lists without row locality, unlike ListRLE. All unique lists are replaced with a distinct integer code, resulting in a new integer column. Table 4.4 shows the ListDict encoding of the column from Table 4.1, where dictionary codes are bit-packed, resulting in a compression ratio of 2.6.

Like ListRLE, we can use cascading encoding with ListDict to achieve more significant compression. For instance, if we apply Dictionary encoding to the underlying strings and then apply ListDict, we get a total size of 140 bytes (130 bytes string dictionary and

Table 4.1: An excerpt of consecutive rows from the ListRLE-compressible `facet_therapeuticAreas` column of the `opentargets_latest-aotfelasticsearch` table. The total uncompressed size is 631 bytes.

(a) Rows in JSON-like representation.

Row №	Value
1	`["endocrine system disease", "cancer or benign tumor"]`
2	`["endocrine system disease", "cancer or benign tumor"]`
3	`["endocrine system disease", "cancer or benign tumor"]`
4	`["endocrine system disease", "cancer or benign tumor"]`
5	`["cancer or benign tumor", "gastrointestinal disease"]`
6	`["cancer or benign tumor", "gastrointestinal disease"]`
7	`["cancer or benign tumor", "reproductive system or breast disease"]`
8	`["hematologic disease", "cancer or benign tumor"]`
9	`["hematologic disease", "cancer or benign tumor"]`
10	`["hematologic disease", "cancer or benign tumor"]`
11	`["hematologic disease", "cancer or benign tumor"]`

(b) Columnar representation. Lengths are assumed to be 4-byte integers.

№	Lengths
1	2
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	2
10	2
11	2
Total: 44 bytes	

№	Values
1	`"endocrine system disease"`
2	`"cancer or benign tumor"`
3	`"endocrine system disease"`
4	`"cancer or benign tumor"`
5	`"endocrine system disease"`
6	`"cancer or benign tumor"`
7	`"endocrine system disease"`
8	`"cancer or benign tumor"`
9	`"cancer or benign tumor"`
10	`"gastrointestinal disease"`
11	`"cancer or benign tumor"`
12	`"gastrointestinal disease"`
13	`"cancer or benign tumor"`
14	`"reproductive system or breast disease"`
15	`"hematologic disease"`
16	`"cancer or benign tumor"`
17	`"hematologic disease"`
18	`"cancer or benign tumor"`
19	`"hematologic disease"`
20	`"cancer or benign tumor"`
21	`"hematologic disease"`
22	`"cancer or benign tumor"`
587 bytes (88 bytes lengths, 499 bytes data)	

Table 4.2: ListRLE compression of the example data. Each run is stored as the repeating list and the number of times it repeats. The total compressed size is 256 bytes.

Length	Value
4	["endocrine system disease", "cancer or benign tumor"]
2	["cancer or benign tumor", "gastrointestinal disease"]
1	["cancer or benign tumor", "reproductive system or breast disease"]
4	["hematologic disease", "cancer or benign tumor"]
16 bytes	**240 bytes ($4 * 4$ and $8 * 4$ bytes list resp. string lengths, 192 bytes data)**

Table 4.3: Cascading compression (Dictionary, ListRLE, Bit-packing) of the example data. The list of each run is stored as a list of dictionary codes. All lengths (string, list, and RLE lengths) and dictionary codes are bit-packed. The total compressed size is 136 bytes.

Dictionary

№	Value
0	"endocrine system disease"
1	"cancer or benign tumor"
2	"gastrointestinal disease"
3	"hematologic disease"
4	"reproductive system or breast disease"
$5 * 6$ **bits** + 126 **bytes** ≈ 130 **bytes**	

ListRLE

Length	Value
4	[0, 1]
2	[1, 2]
1	[1, 3]
4	[4, 1]
$4 * 3$ **bits** ≈ 2 **bytes**	$4 * 2$ **bits** + $8 * 3$ **bits** = 4 **bytes**

Table 4.4: ListDict compression of the example data. Each unique list is assigned and replaced with a bit-packed dictionary code. The total compressed size is 246 bytes.

ListDict Dictionary

№	Value
0	["endocrine system disease", "cancer or benign tumor"]
1	["cancer or benign tumor", "gastrointestinal disease"]
2	["cancer or benign tumor", "reproductive system or breast disease"]
3	["hematologic disease", "cancer or benign tumor"]
240 bytes ($4 * 4$ and $8 * 4$ bytes list resp. string lengths, 192 bytes data)	

№	Code
1	0
2	0
3	0
4	0
5	1
6	1
7	2
8	3
9	3
10	3
11	3
$11 * 4$ **bits** ≈ 6 **bytes**	

4 bytes ListDict dictionary similar to Dictionary and ListRLE values from Table 4.3 with +6 bytes for ListDict-encoded integer column) and 4.5 compression ratio.

4.3 ListAsString - Applying String Compression to Lists

Strings can be considered as a special case of lists where the child type is 1-byte integers. Since specialized string compression algorithms like FSST exist, we can try applying them to lists to get their compression benefits for the list type. The challenge is uniquely encoding each list as a string.

Dictionary encoding is a helpful starting point that allows us to convert any column to an integer column. So, as a first step, we can dictionary encode the child column of the list. Then, if the dictionary contains less than 256 entries, the list column can be trivially converted to a string column and encoded with string LWC schemes. Otherwise, dictionary codes no longer fit a 1-byte character in a string. It is possible to use more characters to represent a single child item. For instance, if the dictionary has less than 65536 entries, each code can be converted to 2-byte characters in a string that is double the length of the original list. This conversion applied to the example list column is given in Table 4.5. After this conversion, any string encoding scheme can be used on the resulting column.

Table 4.5: String representation of the example list column, assuming a byte/character can only hold codes 0-3. The dictionary from Table 4.3 is used here. The largest code is 4, so we need 2 bytes/characters per code.

№	Value
1	"0001"
2	"0001"
3	"0001"
4	"0001"
5	"0102"
6	"0102"
7	"0103"
8	"1001"
9	"1001"
10	"1001"
11	"1001"

Another method of dealing with larger dictionaries is the following exception-handling technique. Any dictionary entry mapping to an integer >255 can be considered an exception and replaced with 255. Exceptions are stored separately as uncompressed offset-code pairs. The dictionary must be sorted by the descending occurrence frequency, and the dictionary codes must be assigned in that order to keep the size of exceptions to a minimum. Table 4.6 shows the conversion of the example list to a string column with this exception-handling technique.

Table 4.6: Representing the example list column as a string column. Codes 0-3 are shown as the matching character, and larger codes are considered an exception (marked with 'X') for demonstration purposes.

Dictionary

Code	Value	Frequency ↓
0	`"cancer or benign tumor"`	11
1	`"endocrine system disease"`	4
2	`"hematologic disease"`	4
3	`"gastrointestinal disease"`	2
4	`"reproductive system or breast disease"`	1

Exceptions

Offset	Code
13	4

№	Value
1	`"10"`
2	`"10"`
3	`"10"`
4	`"10"`
5	`"03"`
6	`"03"`
7	`"0X"`
8	`"20"`
9	`"20"`
10	`"20"`
11	`"20"`

4.4 Handling List of Structs

So far, we have introduced list-specific compression schemes on lists of simple types. However, lists of structs are widespread in practice. One prominent example of such a case is the map type, represented as a list of struct-typed columns. This section will discuss two ways of encoding such lists using the abovementioned compression schemes. We will consider Table 4.7 as an example of a list with struct contents from the RealNest dataset.

The first way of dealing with lists of structs is trivially considering each struct value as a single complex list item. For instance, ListDict encoding requires storing unique (non-equal) lists in a dictionary. To check for the equality of structs, we can check for the equality of each struct subfield individually, and struct values are considered equal if all the subfields are equal. If the struct also contains a list as a subfield, lists are considered equal if their lengths and contents are equal. Although this is easier to implement, it is only effective if all struct subfields have the same data pattern for compression. If the patterns do not match, the list column cannot be effectively compressed using list compression schemes. This problem holds for the example column, where all the structs are unique because of the uniqueness of the `record_id` subfield, even though the `dataset` subfield has a repeating pattern.

Another method of handling structs inside a list is pushing down the list structure to individual subfields of a struct. In other words, a list is split into multiple lists where

Table 4.7: An excerpt of consecutive rows from the `sources` column of the `overturemaps-us-west-2-divisions` table.

Row №	Value
1	`[{"dataset": "OpenStreetMap", "record_id": "R2277758"}]`
2	`[{"dataset": "OpenStreetMap", "record_id": "R2277664"}]`
3	`[{"dataset": "OpenStreetMap", "record_id": "R2277717"}, {"dataset": "OpenStreetMap", "record_id": "N26037814"}]`
4	`[{"dataset": "OpenStreetMap", "record_id": "R2277771"}]`
5	`[{"dataset": "OpenStreetMap", "record_id": "R2277743"}, {"dataset": "OpenStreetMap", "record_id": "N26036692"}]`
6	`[{"dataset": "OpenStreetMap", "record_id": "R2277818"}]`
7	`[{"dataset": "OpenStreetMap", "record_id": "R2277776"}]`
8	`[{"dataset": "OpenStreetMap", "record_id": "R2277725"}, {"dataset": "OpenStreetMap", "record_id": "N26035929"}]`

each new list holds values of a single struct subfield. Table 4.8 shows the result after splitting the example column. Since the struct contained two subfields, we get two new list columns with string content where the lists have the same lengths. After this split, the list with `dataset` values can be effectively compressed with Dictionary and ListDict combination, while the `record_id` list can be kept uncompressed. With this method, it is essential to share the lengths of new columns where possible. For instance, if multiple new lists are decided to be kept uncompressed, the lengths of those lists will be equal and can be stored once to prevent redundant data storage. This adds to the complexity of the implementation of this method.

Table 4.8: Splitting a list of structs into multiple lists of simple types for individual compression.

Row №	`dataset`	`record_id`
1	`["OpenStreetMap"]`	`["R2277758"]`
2	`["OpenStreetMap"]`	`["R2277664"]`
3	`["OpenStreetMap", "OpenStreetMap"]`	`["R2277717", "N26037814"]`
4	`["OpenStreetMap"]`	`["R2277771"]`
5	`["OpenStreetMap", "OpenStreetMap"]`	`["R2277743", "N26036692"]`
6	`["OpenStreetMap"]`	`["R2277818"]`
7	`["OpenStreetMap"]`	`["R2277776"]`
8	`["OpenStreetMap", "OpenStreetMap"]`	`["R2277725", "N26035929"]`

5 Evaluation

This chapter will evaluate the list compression schemes ListRLE, ListDict, and ListAsString in terms of compression ratio on the RealNest dataset. The schemes have been implemented using FastLanes with multi-level cascading encoding support. The benchmarks in this chapter are performed on the 100 MiB per table version[1] of the RealNest dataset. For evaluating list compression schemes, the columns not inside any list in the schema are ignored as these are not affected by the list schemes.

5.1 Implementation Details

Support for nested data types in FastLanes has been added as a part of this thesis. This includes importing nested data with schema from JSON files and converting it to an in-memory columnar format. The columnar representation of each column is associated with a nullmap to store null values. Additionally, structs contain a column per subfield, while lists contain one child column for the list content and an integer column for list offsets. Maps are converted to a list of struct columns as described in Subsection 2.2.1.

A compression scheme has been implemented for nested data types to evaluate the effectiveness of FastLanes compression schemes for simple typed columns on unnested data. For structs, this scheme compresses the subfields individually and adds up the byte size of each subfield. For lists, the offsets column is first compressed using the best FastLanes integer scheme, and then the list contents are compressed. Upon reaching a simple column, the best FastLanes compression for that column is used to compress the data. All available schemes are used to compress the data to determine the best scheme, and the one with the least total size is selected. Since FastLanes compression schemes only support compressing data in vectors of size 1024 at this time, the last valid value of the column is duplicated until the number of tuples reaches a multiple 1024 to ensure correctness.

The implementation of the ListRLE scheme looks for the repeated values in a given list column and produces two output columns: run lengths as an integer column and run values as a column of the list child type. Run lengths are further compressed using FastLanes integer compression schemes, while run values are bit-packed. ListRLE scheme has a dictionary variant, where the list values are first dictionary-encoded, producing a column of type list of integers. Then, the ListRLE is run on this produced list column. The dictionary is stored as bit-packed dictionary keys in the order of their assigned codes.

[1] `https://github.com/cwida/RealNest/tree/main/sample-data/100mib`

The ListDict scheme creates a dictionary of all distinct lists in a list column. The dictionary keys are bit-packed and stored according to the order of their assigned codes. ListDict scheme produces a new integer column that is further compressed using FastLanes integer schemes. Like ListRLE, ListDict has a dictionary variant for running ListDict on a list column with dictionary-encoded values.

The ListAsString implementation first dictionary-encodes the items in the list, taking into account the number of occurrences of items. The dictionary is stored as bit-packed keys. Then, all lists are converted to a string where each list item is replaced with the dictionary code. For large dictionaries, both multi-byte characters and exception-handling variants are implemented. The exceptions are stored bit-packed as offset/original code pairs. The resulting string column is compressed using FastLanes string compression schemes.

List compression schemes have been implemented where structs in lists are kept as a single value and not split as per the explanation in Section 4.4 to keep the implementation simple. However, splitting without offset sharing is also evaluated in this chapter.

5.2 Evaluating ListRLE

Figure 5.1 shows the effect of adding each list compression scheme to FastLanes in isolation for all RealNest tables with list columns. We see that ListRLE can be very effective for some tables, like `opentargets_latest-aotfelasticsearch`, where the compression ratio has been improved by more than a factor of 10 compared to running FastLanes compression schemes on the unnested lists. This is shown more clearly in Figure 5.2. ListRLE provides, on average, 13% more compression with a maximum of 95% on one of the tables, which consists of repeating large lists with long strings.

5.3 Evaluating ListDict

As can be seen from Figure 5.1, ListDict is usually less effective on tables where ListRLE can achieve high compression ratios, as the lists need to be stored in an additional dictionary and replaced with the dictionary code in the original column. However, ListDict can be applied to a more extensive set of tables with duplicate lists, like the `opentargets_latest-knowndrugsaggregated` table, since it doesn't require lists to be repeated consecutively. As per Figure 5.3, ListDict is generally more beneficial than ListRLE, with a 14% average improvement, and applicable to more tables.

5.4 Evaluating ListAsString

Figure 5.1 shows that although ListAsString can benefit some RealNest tables in FastLanes, it is not enough to match ListRLE and ListDict schemes when considered at the

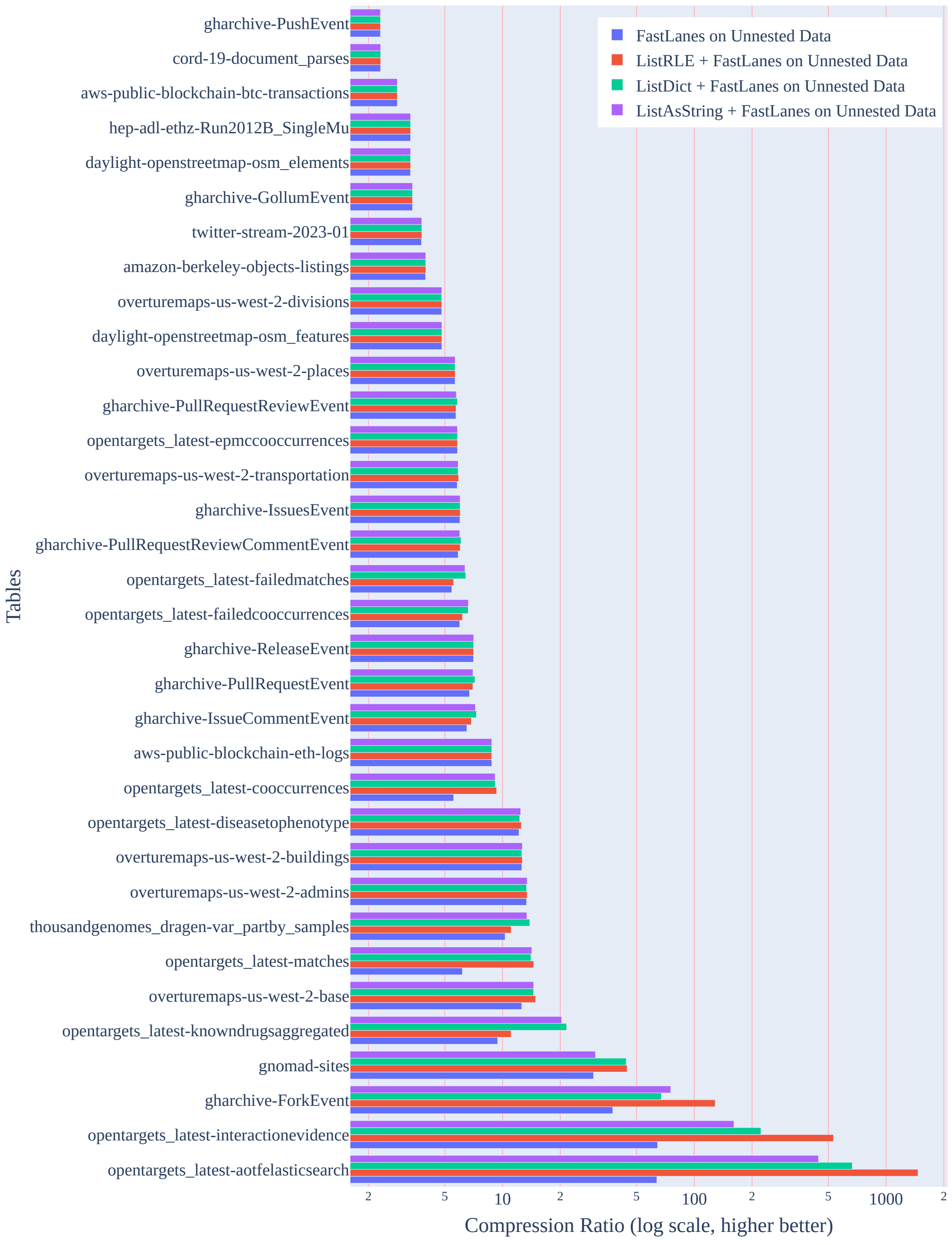


Figure 5.1: The compression ratio for RealNest tables where each list scheme is evaluated in isolation.

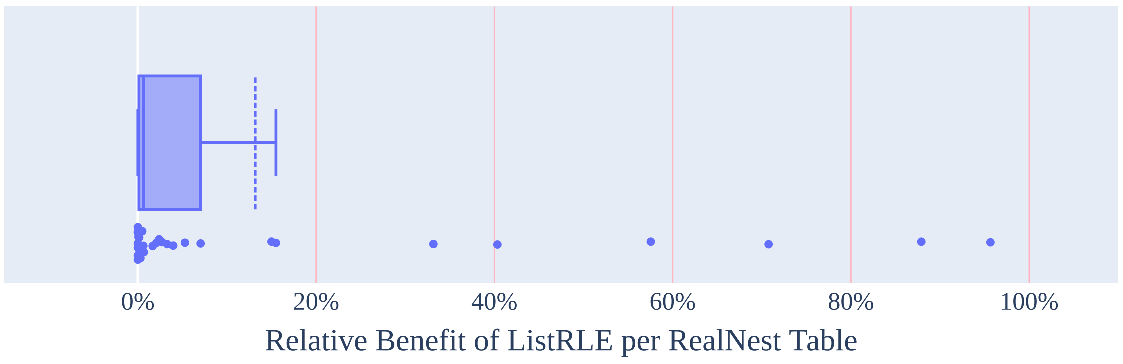


Figure 5.2: The relative benefit of the ListRLE scheme compared to FastLanes on unnested lists of the RealNest dataset.

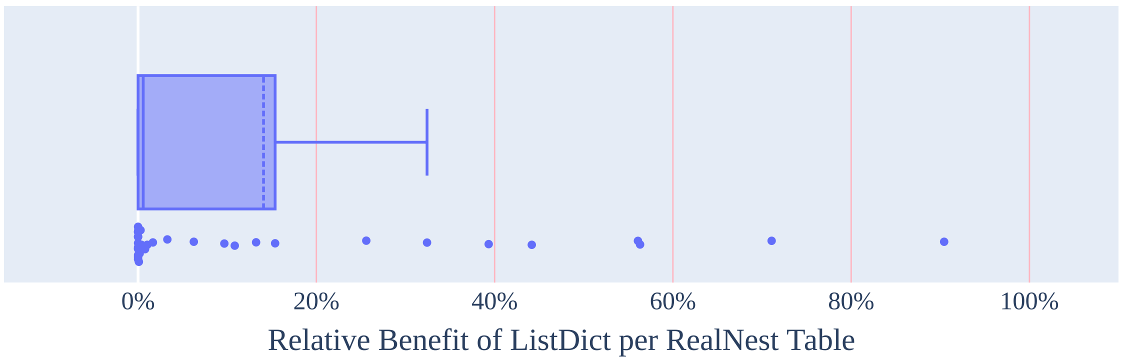


Figure 5.3: The relative benefit of the ListDict scheme compared to FastLanes on unnested lists of the RealNest dataset.

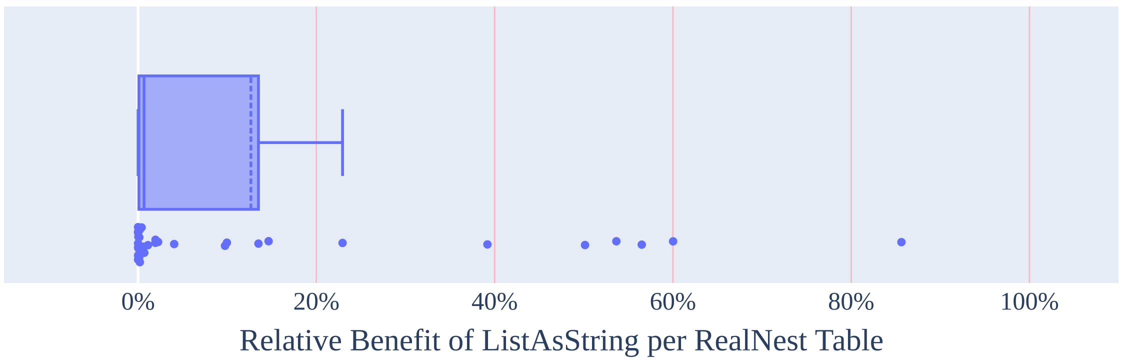


Figure 5.4: The relative benefit of the ListAsString scheme compared to FastLanes on unnested lists of the RealNest dataset.

table level. This is because most lists contain more than 256 unique elements, and depending on the variant, either the exceptions table grows too large or the strings become too long for effective string encoding, making ListAsString unfeasible for those columns. The lists often repeat for list columns with fewer distinct elements, making ListDict more feasible. The ListAsString scheme provides, on average, a 13% improvement in isolation in FastLanes, as shown in Figure 5.4.

5.5 Evaluating Splitting Lists of Structs

Section 4.4 discussed two ways of handling list columns with struct elements for list LWC schemes. In this section, we evaluate those two methods. Note that the offsets after splitting a list of structs are not shared for this evaluation. Figure 5.5 shows the relative difference between splitting structs and considering structs as a complex list item. Splitting structs can be up to 17% better at the cost of potentially having to store the same offsets multiple times. The result shows that splitting structs without offset sharing is generally not worthwhile. However, offset sharing can still be a viable alternative that improves the compression ratio.

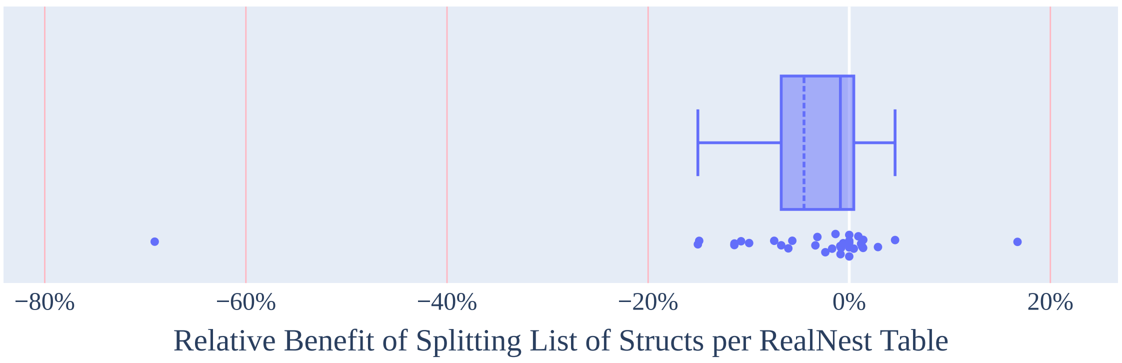


Figure 5.5: The relative difference between applying list compression to a list of structs where structs are considered as a single unit and structs are split into multiple lists.

5.6 Evaluating FastLanes with New List Compression Schemes

The power of all discussed list schemes is shown in Figure 5.6. List-specific LWC schemes improve the compression ratio by 16% on average on lists of the RealNest tables. This proves that the application of traditional LWC schemes to the unnested data does not take advantage of the nested data patterns, and the encodings designed for nested data can significantly enhance the effectiveness of LWC schemes for nested tables.

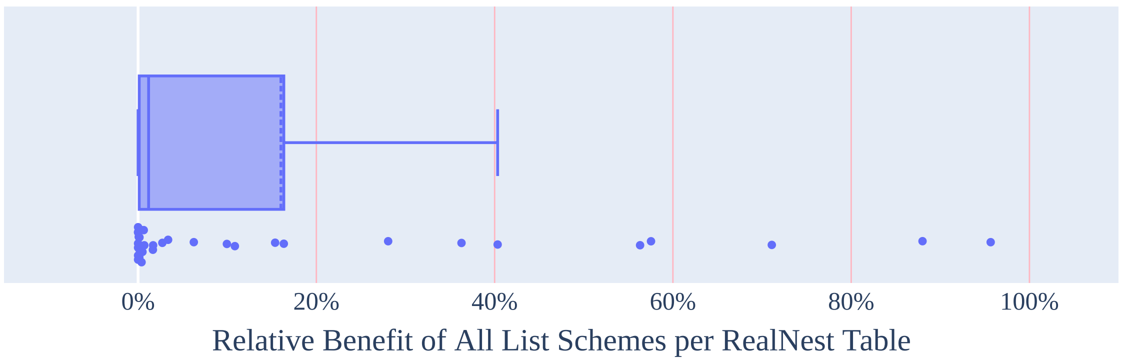


Figure 5.6: The relative benefit of the list schemes compared to FastLanes on unnested lists of the RealNest dataset.

5.7 FastLanes vs. Other Open Big Data File Formats

This section will compare FastLanes with other widespread open file formats - Parquet, ORC, and DuckDB. Parquet and ORC files were created using Apache Arrow[2] v16.1.0 writers. Parquet writer was configured to use dictionary encoding, set the data page format version to v2.0, and set the size of data pages to 4 MB. Parquet v1.0 is evaluated with Snappy as the general-purpose compression scheme since this is the most common configuration in practice. Parquet v2.6 is evaluated twice, without general-purpose compression and with Zstd. The compression strategy of the ORC writer was set to reduce file size. All other parameters were left as defaults. Since DuckDB does not provide a reliable way of getting the table sizes, a custom build of v1.0.0 was used with a mechanism to access the table sizes.

The compression ratios and the exact file sizes are shown in Figure 5.7 and Table 5.1, respectively. Parquet and ORC files perform the worst for almost all tables without applying general-purpose compression. This indicates that their LWC schemes are insufficient, and they rely on general-purpose compression to achieve good compression ratios. Indeed, with Zstd, a slow-to-decompress general-purpose compression algorithm, both formats achieve much better compression ratios. Parquet v1.0 with Snappy achieves less compression than v2.6 with Zstd, which is expected as Snappy is less efficient but faster than Zstd. DuckDB generally compresses data better than the plain Parquet and ORC files; however, it loses significantly to Zstd. This can be because of the lack of cascading compression in DuckDB. FastLanes results show that list compression schemes and multi-level cascading encoding can get close to the compression levels of Zstd-compressed Parquet and ORC files and sometimes even exceed them while only using LWC schemes.

For completeness, the compressed sizes of all RealNest tables, including all columns (not only lists), are shown in Table 5.2. The similar Zstd results of Parquet and ORC files display how Zstd can capture missed compression opportunities of different LWC

[2]`https://arrow.apache.org/`

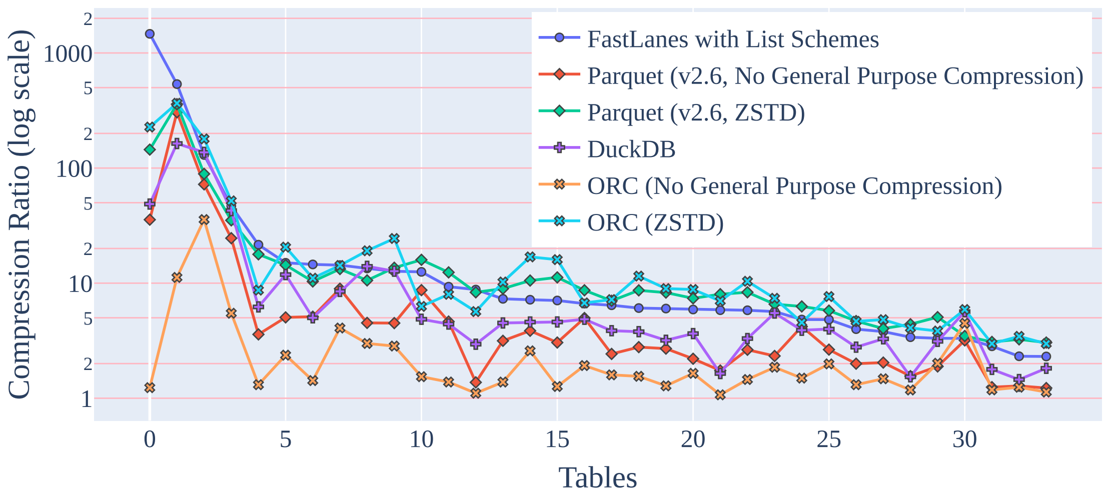


Figure 5.7: Comparing compression ratio of FastLanes (with List compression schemes), Parquet, ORC, and DuckDB for all RealNest tables (only lists are included).

Table 5.1: File sizes for all RealNest tables (only list columns) of different file formats. 'With List' means with new list schemes. GPC stands for General-Purpose Compression.

Table Name (Truncated)	Binary	FastLanes		DuckDB	Parquet			ORC	
		With List	No List		v1.0 Snappy	v2.6 No GPC	v2.6 Zstd	No GPC	Zstd
`...bjects-listings`	59.16 MB	14.85 MB	14.91 MB	21.30 MB	16.96 MB	29.64 MB	12.64 MB	45.12 MB	12.70 MB
`...tc-transactions`	75.72 MB	26.78 MB	26.78 MB	42.50 MB	45.04 MB	60.69 MB	24.55 MB	64.06 MB	25.43 MB
`...kchain-eth-logs`	12.19 MB	1.39 MB	1.39 MB	4.14 MB	3.48 MB	8.87 MB	1.46 MB	11.03 MB	2.15 MB
`gnomad-sites`	37.12 MB	812.06 KB	1.24 MB	890.13 KB	1.23 MB	1.51 MB	1.06 MB	6.78 MB	732.44 KB
`...tfelasticsearch`	48.95 MB	34.18 KB	785.78 KB	1.01 MB	398.36 KB	1.38 MB	346.46 KB	39.61 MB	220.71 KB
`...t-cooccurrences`	10.21 MB	1.10 MB	1.84 MB	2.31 MB	1.13 MB	2.20 MB	843.04 KB	7.38 MB	1.28 MB
`...easetophenotype`	15.93 MB	1.27 MB	1.31 MB	3.28 MB	1.19 MB	1.83 MB	1022.80 KB	10.35 MB	2.54 MB
`...mccooccurrences`	85.35 MB	14.66 MB	14.66 MB	52.06 MB	16.69 MB	49.02 MB	10.65 MB	79.73 MB	12.19 MB
`...edcooccurrences`	24.30 MB	3.66 MB	4.07 MB	5.01 MB	3.72 MB	4.88 MB	2.80 MB	12.64 MB	3.61 MB
`...t-failedmatches`	20.63 MB	3.21 MB	3.79 MB	5.34 MB	4.01 MB	8.50 MB	2.94 MB	12.93 MB	2.87 MB
`...ractionevidence`	1.75 MB	3.35 KB	27.92 KB	11.00 KB	5.34 KB	5.89 KB	5.01 KB	160.32 KB	4.91 KB
`...drugsaggregated`	35.56 MB	1.65 MB	3.77 MB	5.72 MB	3.31 MB	9.92 MB	2.00 MB	27.13 MB	4.09 MB
`..._latest-matches`	9.94 MB	700.02 KB	1.61 MB	1.99 MB	1.28 MB	1.94 MB	984.70 KB	6.99 MB	921.72 KB
`..._partby_samples`	21.91 MB	1.53 MB	2.12 MB	2.58 MB	1.97 MB	2.45 MB	1.65 MB	5.38 MB	1.54 MB
`...document_parses`	63.17 MB	27.28 MB	27.30 MB	43.50 MB	27.43 MB	49.39 MB	19.45 MB	50.69 MB	18.32 MB
`...ap-osm_elements`	12.83 MB	3.87 MB	3.87 MB	4.11 MB	4.18 MB	6.79 MB	2.53 MB	6.37 MB	3.34 MB
`...ap-osm_features`	4.45 MB	943.09 KB	943.09 KB	1.14 MB	903.29 KB	1.06 MB	725.57 KB	2.97 MB	1000.00 KB
`...chive-ForkEvent`	331.46 KB	2.55 KB	8.82 KB	2.42 KB	3.78 KB	4.59 KB	3.73 KB	9.29 KB	1.85 KB
`...ive-GollumEvent`	13.19 MB	3.89 MB	3.89 MB	8.59 MB	4.81 MB	8.48 MB	3.00 MB	11.19 MB	3.23 MB
`...sueCommentEvent`	6.63 MB	928.55 KB	1.02 MB	1.47 MB	951.26 KB	2.10 MB	757.62 KB	4.79 MB	663.36 KB
`...ive-IssuesEvent`	8.91 MB	1.48 MB	1.49 MB	2.80 MB	1.39 MB	3.30 MB	1.08 MB	6.96 MB	1017.12 KB
`...ullRequestEvent`	3.22 MB	459.25 KB	489.88 KB	717.30 KB	385.23 KB	854.41 KB	312.89 KB	1.25 MB	194.95 KB
`...iewCommentEvent`	4.65 MB	783.77 KB	811.08 KB	1.23 MB	709.01 KB	1.67 MB	549.92 KB	3.00 MB	413.41 KB
`...uestReviewEvent`	5.93 MB	1.02 MB	1.04 MB	1.80 MB	941.88 KB	2.25 MB	728.63 KB	4.08 MB	584.57 KB
`...chive-PushEvent`	21.84 MB	9.47 MB	9.47 MB	12.00 MB	11.42 MB	17.81 MB	7.17 MB	19.30 MB	7.34 MB
`...ve-ReleaseEvent`	21.59 MB	3.05 MB	3.06 MB	4.70 MB	2.58 MB	7.09 MB	1.92 MB	17.07 MB	1.35 MB
`...n2012B_SingleMu`	33.07 MB	9.97 MB	9.97 MB	5.85 MB	10.20 MB	10.40 MB	9.52 MB	7.39 MB	5.61 MB
`...s-west-2-admins`	4.09 MB	311.96 KB	314.15 KB	299.42 KB	499.54 KB	928.17 KB	396.45 KB	1.37 MB	219.23 KB
`...-us-west-2-base`	6.91 MB	471.06 KB	563.12 KB	596.55 KB	655.97 KB	1.37 MB	492.70 KB	2.92 MB	344.42 KB
`...est-2-buildings`	4.23 MB	341.51 KB	343.70 KB	340.96 KB	434.07 KB	963.52 KB	318.40 KB	1.49 MB	177.28 KB
`...est-2-divisions`	13.24 MB	2.75 MB	2.75 MB	3.32 MB	2.89 MB	5.01 MB	2.29 MB	6.66 MB	1.73 MB
`...s-west-2-places`	19.16 MB	3.38 MB	3.39 MB	3.48 MB	4.32 MB	8.23 MB	2.92 MB	10.31 MB	2.59 MB
`...-transportation`	10.37 MB	1.76 MB	1.79 MB	2.85 MB	2.15 MB	4.71 MB	1.41 MB	6.30 MB	1.18 MB
`...-stream-2023-01`	55.16 MB	14.53 MB	14.61 MB	16.80 MB	17.69 MB	27.04 MB	13.68 MB	37.37 MB	11.45 MB
Average	22.70 MB	4.65 MB	4.86 MB	7.75 MB	5.73 MB	10.07 MB	3.88 MB	15.61 MB	3.85 MB

schemes and reduce the file size to a nearly equal amount. Moreover, results show that there is still room for improvement in FastLanes to reach the compression ratios of Zstd-compressed data. Upon manual inspection, it can be seen that the string columns with very long text are the main point where LWC schemes lag behind the general-purpose compression schemes. FSST, the best string compression scheme available in FastLanes, cannot capture all the text patterns, given its limited table and symbol sizes.

Table 5.2: File sizes for all RealNest tables (including all columns) of different file formats.

Table Name (Truncated)	**Binary**	**FastLanes**		**DuckDB**	**Parquet**			**ORC**	
		With List	**No List**		**v1.0 Snappy**	**v2.6 No GPC**	**v2.6 Zstd**	**No GPC**	**Zstd**
`...bjects-listings`	61.13 MB	15.16 MB	15.21 MB	22.79 MB	17.20 MB	29.92 MB	12.87 MB	46.08 MB	12.89 MB
`...tc-transactions`	75.72 MB	26.78 MB	26.78 MB	41.22 MB	45.04 MB	60.69 MB	24.55 MB	64.06 MB	25.43 MB
`...kchain-eth-logs`	12.19 MB	1.39 MB	1.39 MB	4.28 MB	3.48 MB	8.87 MB	1.46 MB	11.03 MB	2.15 MB
`...pecific_summary`	3.97 MB	169.27 KB	169.27 KB	228.46 KB	183.23 KB	234.50 KB	153.00 KB	340.77 KB	95.29 KB
`...-hgvs4variation`	768.26 KB	4.12 KB	4.12 KB	392.00 B	1.22 KB	1.32 KB	1.23 KB	1.70 KB	883.00 B
`...mission_summary`	768.26 KB	18.58 KB	18.58 KB	11.16 KB	382.98 KB	636.63 KB	193.51 KB	16.83 KB	2.78 KB
`gnomad-sites`	37.12 MB	812.06 KB	1.24 MB	887.98 KB	1.23 MB	1.51 MB	1.06 MB	6.78 MB	732.44 KB
`...v1_1_9_gene_tpm`	1.60 MB	209.24 KB	209.24 KB	202.61 KB	336.75 KB	469.48 KB	263.81 KB	389.20 KB	195.59 KB
`..._expected_count`	22.46 MB	2.95 MB	2.95 MB	4.33 MB	3.40 MB	4.33 MB	2.67 MB	2.77 MB	1.51 MB
`..._transcript_tpm`	10.41 MB	781.91 KB	781.91 KB	716.89 KB	674.53 KB	729.30 KB	610.87 KB	955.59 KB	421.72 KB
`...tfelasticsearch`	48.95 MB	34.18 KB	785.78 KB	1.42 MB	398.36 KB	1.38 MB	346.46 KB	39.61 MB	220.71 KB
`...t-cooccurrences`	10.21 MB	1.10 MB	1.84 MB	2.31 MB	1.13 MB	2.20 MB	843.04 KB	7.38 MB	1.28 MB
`...easetophenotype`	15.93 MB	1.27 MB	1.31 MB	3.23 MB	1.19 MB	1.83 MB	1022.80 KB	10.35 MB	2.54 MB
`...mccooccurrences`	85.35 MB	14.66 MB	14.66 MB	52.06 MB	16.69 MB	49.02 MB	10.65 MB	79.73 MB	12.19 MB
`...edcooccurrences`	24.30 MB	3.66 MB	4.07 MB	5.01 MB	3.72 MB	4.88 MB	2.80 MB	12.64 MB	3.61 MB
`...t-failedmatches`	20.63 MB	3.21 MB	3.79 MB	5.34 MB	4.01 MB	8.50 MB	2.94 MB	12.93 MB	2.87 MB
`...est-interaction`	4.20 MB	56.01 KB	56.01 KB	199.31 KB	38.70 KB	69.13 KB	28.25 KB	2.24 MB	61.92 KB
`...ractionevidence`	7.39 MB	36.57 KB	61.15 KB	236.87 KB	11.10 KB	11.82 KB	10.75 KB	2.80 MB	10.37 KB
`...drugsaggregated`	35.56 MB	1.65 MB	3.77 MB	5.69 MB	3.31 MB	9.92 MB	2.00 MB	27.13 MB	4.09 MB
`..._latest-matches`	9.94 MB	700.02 KB	1.61 MB	1.99 MB	1.28 MB	1.94 MB	984.70 KB	6.99 MB	921.72 KB
`..._partby_samples`	21.91 MB	1.53 MB	2.12 MB	2.58 MB	1.97 MB	2.45 MB	1.65 MB	5.38 MB	1.54 MB
`...document_parses`	63.27 MB	27.34 MB	27.36 MB	43.28 MB	27.50 MB	49.49 MB	19.50 MB	50.78 MB	18.36 MB
`...ap-osm_elements`	12.83 MB	3.87 MB	3.87 MB	4.11 MB	4.18 MB	6.79 MB	2.53 MB	6.37 MB	3.34 MB
`...ap-osm_features`	4.45 MB	943.09 KB	943.09 KB	1.14 MB	903.29 KB	1.06 MB	725.57 KB	2.97 MB	1000.00 KB
`...mitCommentEvent`	87.26 MB	32.22 MB	32.22 MB	17.92 MB	18.09 MB	69.90 MB	10.05 MB	81.46 MB	11.90 MB
`...chive-ForkEvent`	74.43 MB	23.33 MB	23.34 MB	25.85 MB	21.19 MB	62.89 MB	14.36 MB	65.71 MB	13.31 MB
`...ive-GollumEvent`	13.19 MB	3.89 MB	3.89 MB	8.62 MB	4.81 MB	8.48 MB	3.00 MB	11.19 MB	3.23 MB
`...sueCommentEvent`	83.18 MB	27.00 MB	27.11 MB	28.58 MB	20.66 MB	54.80 MB	13.46 MB	73.32 MB	15.15 MB
`...ive-IssuesEvent`	83.78 MB	38.84 MB	38.85 MB	21.83 MB	33.95 MB	62.96 MB	22.40 MB	75.17 MB	22.53 MB
`...ive-MemberEvent`	44.87 MB	11.46 MB	11.46 MB	11.58 MB	12.37 MB	42.40 MB	8.20 MB	40.76 MB	7.24 MB
`...ullRequestEvent`	69.54 MB	15.84 MB	15.87 MB	24.73 MB	13.65 MB	34.13 MB	9.38 MB	60.62 MB	10.58 MB
`...iewCommentEvent`	69.57 MB	16.08 MB	16.10 MB	26.28 MB	13.58 MB	32.91 MB	9.33 MB	61.39 MB	10.50 MB
`...uestReviewEvent`	78.19 MB	17.57 MB	17.59 MB	29.04 MB	15.56 MB	38.90 MB	10.67 MB	69.39 MB	12.29 MB
`...chive-PushEvent`	21.84 MB	9.47 MB	9.47 MB	12.00 MB	11.42 MB	17.81 MB	7.17 MB	19.30 MB	7.34 MB
`...ve-ReleaseEvent`	77.29 MB	21.02 MB	21.02 MB	28.94 MB	16.59 MB	45.24 MB	11.28 MB	67.79 MB	11.29 MB
`...n2012B_SingleMu`	37.63 MB	12.94 MB	12.94 MB	7.19 MB	12.65 MB	12.85 MB	11.78 MB	9.04 MB	7.07 MB
`...s-west-2-admins`	6.34 MB	1.87 MB	1.87 MB	1.31 MB	1.91 MB	2.33 MB	1.65 MB	2.37 MB	1.00 MB
`...-us-west-2-base`	9.24 MB	2.02 MB	2.11 MB	1.69 MB	2.10 MB	2.86 MB	1.72 MB	4.00 MB	1.09 MB
`...est-2-buildings`	6.48 MB	1.64 MB	1.64 MB	1.15 MB	1.56 MB	2.08 MB	1.31 MB	2.49 MB	781.23 KB
`...est-2-divisions`	16.44 MB	4.83 MB	4.83 MB	5.08 MB	4.93 MB	7.35 MB	4.07 MB	8.41 MB	2.89 MB
`...s-west-2-places`	24.74 MB	5.90 MB	5.90 MB	5.88 MB	7.04 MB	11.59 MB	5.10 MB	13.99 MB	4.30 MB
`...-transportation`	12.73 MB	2.97 MB	3.00 MB	3.48 MB	3.10 MB	5.66 MB	2.22 MB	7.42 MB	1.70 MB
`...-stream-2023-01`	60.03 MB	16.25 MB	16.32 MB	18.65 MB	19.73 MB	29.94 MB	15.25 MB	40.84 MB	12.84 MB
Average	34.14 MB	8.68 MB	8.85 MB	11.23 MB	8.68 MB	18.42 MB	5.86 MB	25.91 MB	5.87 MB

6 Conclusion and Future Work

This thesis deeply analyzed the nested data types and their encodings for FastLanes. The RealNest dataset is created to capture the real-world storage of nested data types in various file formats, and it can serve as a compression corpus for current and future nested data encodings in file formats. Analyzing the RealNest dataset brought up missed compression opportunities in the existing nested data encodings of the widespread open big data file formats. New specialized nested data type schemes were introduced to take advantage of these patterns and reduce table sizes further. These schemes and their cascading variants were evaluated in FastLanes against the traditional LWC schemes and other popular open file formats.

The key findings are given below as answers to the main research questions from Section 1.1 of this thesis.

RQ1. How is nested data structured in real-world use cases? What is a suitable dataset that can be used to benchmark existing file formats and their nested type encoding schemes?

As there was no nested data-focused compression corpus, the RealNest dataset was introduced to fill this gap. It comprises the nested columns of various public user-generated datasets from multiple real-world domains. The RealNest dataset provides insight into the structure of nested data in practice. As discussed in Section 3.2, columns can be deeply nested in the schema up to nine levels. Future work in nested data type encodings can benefit from the RealNest dataset for benchmarking the effectiveness of the schemes.

RQ2. Is there a need for specialized nested data-type encoding schemes in file formats?

In Table 4.1, we saw some values from a list column of a RealNest table, where unnested representation made it challenging to apply LWC schemes as this lost the essential list-level repetition information. This shows that the structure of the data is as crucial as its values in terms of data patterns for compression. Hence, there is a need for special encoding schemes that consider the data structure for nested types to achieve greater compression ratios.

RQ3. What are the possible encoding schemes for nested data types that facilitate fast decompression?

This work introduced three new encoding schemes for the list type inspired by the traditional LWC schemes. These were ListRLE, ListDict, and ListAsString schemes and cascading versions that applied conventional LWC schemes on top to get more compression. These schemes remain lightweight and should be efficiently decodable due to the simplicity of operations required.

Although these three list schemes have their respective merits, not all need to be included in a finalized FastLanes file format to achieve satisfactory compression ratios. For instance, the ListAsString scheme rarely beats the other schemes on the list columns of the RealNest dataset; hence, it can be omitted from the final file format. ListRLE and ListDict are also similar in terms of the kind of columns they can compress. ListRLE compressible columns usually can achieve high compression using a cascading encoding of integer RLE on ListDict-encoded columns. Therefore, cascading ListDict encoding is sufficient to get close to the best achievable compression ratios with the list schemes introduced in this thesis. If a second list scheme is desired, ListRLE can also be included to increase the compression ratios further, as shown in Figure 5.1.

RQ4. How much compression improvement can FastLanes get with specialized nested data type encoding schemes?

As shown in Section 5.6, these simple list encodings can provide great compression benefits for nested data. On the lists of RealNest, 16% more compression was achieved on average, whereas for some tables, the gain reached 90%, sometimes achieving better table sizes than Zstd-compressed Parquet and ORC files. This benefit justifies having specialized nested data type encodings in FastLanes.

6.1 Future Work

Although this thesis made essential contributions to encoding nested data types, more research remains for future work. When evaluating the new list encoding schemes, we focused only on the compression ratio; however, the decompression speed is another critical factor. After efficient vectorized implementation of these schemes in FastLanes, it remains to be seen how fast the data can be decoded and how the speed compares to other file formats. The possibility of accelerating the decoding using GPUs also needs to be analyzed.

All three new schemes were derived for the list type. However, other nested data types might also show compressible data patterns. For instance, subfields of a struct column might have internal data correlations. C3 project[1] can be applied to structs to exploit these correlations and reduce the amount of stored data [31]. Application of

[1]`https://github.com/cwida/C3`

traditional LWC schemes, like RLE and Dictionary encoding, to the struct type can also be considered. It might also be possible to derive encodings for the map type. Although we saw how maps are converted to lists of structs before encoding, it might be better to convert maps to structs in terms of compression ratio. If a map column only has a few unique keys, it can be represented as a struct where each unique key is converted to a subfield of a struct. This would remove the associated list overhead and allow maps to be stored like structs.

Finally, Table 5.2 showed that general-purpose compression algorithms still provide a better compression ratio for some tables. Manual analysis showed that the large string columns were the main reason for this difference. Current string LWC schemes, like FSST, are not sufficient to reach the compression levels of general-purpose compression, like Zstd, when presented with large texts. More research is required in string compression to develop new LWC schemes that push the compression ratio higher. As a significant component of the ListAsString scheme, developments in string compression will also improve its performance.

Abbreviations

SIMD Single Instruction/Multiple Data

LWC LightWeight Compression

RLE Run Length Encoding

FSST Fast Static Symbol Table

PAX Partition Attribute Across

JSON JavaScript Object Notation

JSONL JSON Lines

NDJSON Newline Delimited JSON

List of Figures

List of Tables

Bibliography

[1] "Apache Parquet." (2024), [Online]. Available: https://parquet.apache.org (visited on 07/15/2024).

[2] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden, "Tile-based lightweight integer compression in GPU," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 1390–1403, ISBN: 9781450392495. DOI: 10.1145/3514221.3526132.

[3] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar RAM-CPU cache compression," in *22nd International Conference on Data Engineering (ICDE'06)*, Apr. 2006, pp. 59–59. DOI: 10.1109/ICDE.2006.150.

[4] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06, Chicago, IL, USA: Association for Computing Machinery, 2006, pp. 671–682, ISBN: 1595934340. DOI: 10.1145/1142473.1142548.

[5] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970, ISSN: 0001-0782. DOI: 10.1145/362384.362685.

[6] A. Afroozeh and P. Boncz, "The FastLanes compression layout: Decoding > 100 billion integers per second with scalar code," *Proc. VLDB Endow.*, vol. 16, no. 9, pp. 2132–2144, May 2023, ISSN: 2150-8097. DOI: 10.14778/3598581.3598587.

[7] A. Afroozeh, L. Felius, and P. Boncz, "Accelerating GPU data processing using FastLanes compression," in *Proceedings of the 20th International Workshop on Data Management on New Hardware*, ser. DaMoN '24, Santiago, AA, Chile: Association for Computing Machinery, 2024, ISBN: 9798400706677. DOI: 10.1145/3662010.3663450.

[8] M. Zukowski, N. Nes, and P. Boncz, "DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing," in *Proceedings of the 4th International Workshop on Data Management on New Hardware*, ser. DaMoN '08, Vancouver, Canada: Association for Computing Machinery, 2008, pp. 47–54, ISBN: 9781605581842. DOI: 10.1145/1457150.1457160.

[9] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner, "Lightweight data compression algorithms: An experimental survey (experiments and analyses).," in *EDBT*, 2017, pp. 72–83. DOI: 10.5441/002/edbt.2017.08.

[10] D. Lemire and L. Boytsov, “Decoding billions of integers per second through vectorization,” *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, 2015. doi: https://doi.org/10.1002/spe.2203. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2203.

[11] W. Zhang, Y. Wang, and K. A. Ross, “Parallel prefix sum with SIMD,” in *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2020, Tokyo, Japan, August 31, 2020*, R. Bordawekar and T. Lahiri, Eds., 2020, pp. 1–11.

[12] P. Boncz, T. Neumann, and V. Leis, “FSST: Fast random access string compression,” *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2649–2661, Jul. 2020, issn: 2150-8097. doi: 10.14778/3407790.3407851.

[13] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then, “Get real: How benchmarks fail to represent the real world,” in *Proceedings of the Workshop on Testing Database Systems*, ser. DBTest’18, Houston, TX, USA: Association for Computing Machinery, 2018, isbn: 9781450358262. doi: 10.1145/3209950.3209952.

[14] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis, “BtrBlocks: Efficient columnar compression for data lakes,” *Proc. ACM Manag. Data*, vol. 1, no. 2, Jun. 2023. doi: 10.1145/3589263.

[15] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, H. Ahmadi, D. Delorey, S. Min, M. Pasumansky, and J. Shute, “Dremel: A decade of interactive SQL analysis at web scale,” *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3461–3472, Aug. 2020, issn: 2150-8097. doi: 10.14778/3415478.3415568.

[16] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 330–339, Sep. 2010, issn: 2150-8097. doi: 10.14778/1920841.1920886.

[17] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, “Weaving relations for cache performance,” in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB ’01, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 169–180, isbn: 1558608044.

[18] “Apache ORC.” (2024), [Online]. Available: https://orc.apache.org (visited on 07/15/2024).

[19] X. Zeng, Y. Hui, J. Shen, A. Pavlo, W. McKinney, and H. Zhang, “An empirical evaluation of columnar storage formats,” *Proc. VLDB Endow.*, vol. 17, no. 2, pp. 148–161, Oct. 2023, issn: 2150-8097. doi: 10.14778/3626292.3626298.

[20] “DuckDB.” (2024), [Online]. Available: https://duckdb.org (visited on 07/15/2024).

[21] B. Schlegel, R. Gemulla, and W. Lehner, "Fast integer compression using SIMD instructions," in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, ser. DaMoN '10, Indianapolis, Indiana: Association for Computing Machinery, 2010, pp. 34–40, ISBN: 9781450301893. DOI: `10.1145/1869389.1869394`.

[22] D. Habich, P. Damme, A. Ungethüm, and W. Lehner, "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*, ser. DBTest '18, Houston, TX, USA: Association for Computing Machinery, 2018, ISBN: 9781450358262. DOI: `10.1145/3209950.3209957`.

[23] R. O. Nambiar and M. Poess, "The making of TPC-DS," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06, Seoul, Korea: VLDB Endowment, 2006, pp. 1049–1058.

[24] B. V. Ghita, D. G. Tomé, and P. A. Boncz, "White-box compression: Learning and exploiting compact table representations," in *Conference on Innovative Data Systems Research*, 2020.

[25] "RealNest - nested data from real-world datasets." (2024), [Online]. Available: `https://homepages.cwi.nl/~boncz/RealNest/` (visited on 07/15/2024).

[26] CMS collaboration, *SingleMu primary dataset in AOD format from run of 2012 (/SingleMu/Run2012B-22Jan2013-v1/AOD)*, CERN Open Data Portal, 2017. DOI: `10.7483/OPENDATA.CMS.IYVQ.1J0W`.

[27] D. Graur, I. Müller, M. Proffitt, G. Fourny, G. T. Watts, and G. Alonso, "Evaluating query languages and systems for high-energy physics data," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 154–168, Oct. 2021, ISSN: 2150-8097. DOI: `10.14778/3489496.3489498`.

[28] D. Graur, I. Müller, M. Proffitt, G. Fourny, G. T. Watts, and G. Alonso, *Benchmark Scripts for "Evaluating Query Languages and Systems for High-Energy Physics Data"*, version v0.2, Apr. 2022. DOI: `10.5281/zenodo.6505492`.

[29] J. Collins, S. Goel, K. Deng, A. Luthra, L. Xu, E. Gundogdu, X. Zhang, T. F. Y. Vicente, T. Dideriksen, H. Arora, M. Guillaumin, and J. Malik, "ABO: Dataset and benchmarks for real-world 3D object understanding," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2022, pp. 21 094–21 104. DOI: `10.1109/CVPR52688.2022.02045`.

[30] L. L. Wang, K. Lo, Y. Chandrasekhar, R. Reas, J. Yang, D. Eide, K. Funk, R. M. Kinney, Z. Liu, W. Merrill, P. Mooney, D. A. Murdick, D. Rishi, J. Sheehan, Z. Shen, B. Stilson, A. D. Wade, K. Wang, C. Wilhelm, B. Xie, D. A. Raymond, D. S. Weld, O. Etzioni, and S. Kohlmeier, "CORD-19: The Covid-19 open research dataset," *ArXiv*, 2020.

[31] T. Glas, "Exploiting column correlations for compression," M.S. thesis, School of Computation, Information and Technology, Technische Universität München, Dec. 2023.