An Empirical Evaluation of Columnar Storage Formats (Extended Version)

Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo[†], Wes McKinney[‡], Huanchen Zhang
Tsinghua University, [†]Carnegie Mellon University , [‡]Voltron Data
{zeng-xy21,huiyl22,shen-jh20}@mails.tsinghua.edu.cn
pavlo@cs.cmu.edu,wes@voltrondata.com,huanchen@tsinghua.edu.cn

ABSTRACT

Columnar storage is a core component of a modern data analytics system. Although many database management systems (DBMSs) have proprietary storage formats, most provide extensive support to open-source storage formats such as Parquet and ORC to facilitate cross-platform data sharing. But these formats were developed over a decade ago, in the early 2010s, for the Hadoop ecosystem. Since then, both the hardware and workload landscapes have changed.

In this paper, we revisit the most widely adopted open-source columnar storage formats (Parquet and ORC) with a deep dive into their internals. We designed a benchmark to stress-test the formats' performance and space efficiency under different workload configurations. From our comprehensive evaluation of Parquet and ORC, we identify design decisions advantageous with modern hardware and real-world data distributions. These include using dictionary encoding by default, favoring decoding speed over compression ratio for integer encoding algorithms, making block compression optional, and embedding finer-grained auxiliary data structures. We also point out the inefficiencies in the format designs when handling common machine learning workloads and using GPUs for decoding. Our analysis identified important considerations that may guide future formats to better fit modern technology trends.

1 INTRODUCTION

Columnar storage has been widely adopted for data analytics because of its advantages such as irrelevant attribute skipping, efficient data compression, and vectorized query processing [60, 64, 73]. In the early 2010s, organizations developed data processing engines for the open-source big data ecosystem [12], including Hive [13, 110], Impala [16], Spark [20, 118], and Presto [19, 103], to respond to the petabytes of data generated per day and the growing demand for large-scale data analytics. To facilitate data sharing across the various Hadoop-based query engines, vendors proposed open-source columnar storage formats [11, 17, 18, 81], represented by Parquet and ORC, that have become the *de facto* standard for data storage in today's data warehouses and data lakes [14, 15, 19, 20, 30, 41, 66].

These formats, however, were developed more than a decade ago. The hardware landscape has changed since then: persistent storage performance has improved by orders of magnitude, achieving gigabytes per second [53]. Meanwhile, the rise of data lakes means more column-oriented files reside in cheap cloud storage (e.g., AWS S3 [7], Azure Blob Storage [25], Google Cloud Storage [35]), which exhibits both high bandwidth and high latency. On the software side, a number of new lightweight compression schemes [62, 70, 92, 120], as well as indexing and filtering techniques [82, 91, 106, 119], have been proposed in academia, while existing open columnar formats are based on DBMS methods from the 2000s [61].

Prior studies on storage formats focus on measuring the end-to-end performance of Hadoop-based query engines [77, 85]. They fail to analyze the design decisions and their trade-offs. Moreover, they use synthetic workloads that do not consider skewed data distributions observed in the real world [114]. Such data sets are less suitable for storage format benchmarking.

The goal of this paper is to analyze state-of-the-art columnar file formats and to identify important design considerations to provide insights for developing next-generation column-oriented storage formats. To achieve this, we created a benchmark with predefined workloads whose configurations were extracted from a collection of real-world data sets. We then performed a comprehensive analysis for the major components in Parquet and ORC, including encoding algorithms, block compression, metadata organization, indexing and filtering, and nested data modeling. In particular, we investigated how efficiently the columnar formats support common machine learning workloads and whether their designs are friendly to GPUs. We detail the lessons learned in Section 6 and summarize our main findings below.

First, there is no clear winner between Parquet and ORC in format efficiency. Parquet has a slight file size advantage because of its aggressive dictionary encoding. Parquet also has faster column decoding due to its simpler integer encoding algorithms, while ORC is more effective in selection pruning due to the finer granularity of its zone maps (a type of sparse index).

Second, most columns in real-world data sets have a small number of distinct values (or low "NDV ratios" defined in Section 4.1), which is ideal for dictionary encoding. As a result, the efficiency of integer-encoding algorithms (i.e., to compress dictionary codes) is critical to the format's size and decoding speed. Third, faster and cheaper storage devices means that it is better to use faster decoding schemes to reduce computation costs than to pursue more aggressive compression to save I/O bandwidth. Formats should not apply general-purpose block compression (e.g., Snappy [36], zstd [59]) by default because the bandwidth savings do not justify the decompression overhead.

Fourth, Parquet and ORC provide simplistic support for auxiliary data structures (e.g., zone maps, Bloom Filters). As bottlenecks shift from storage to computation, there are opportunities to embed more sophisticated structures and precomputed results into the format to trade inexpensive space for less computation.

Fifth, existing columnar formats are inefficient in serving common machine learning (ML) workloads. Current designs are suboptimal in handling projections of thousands of features during

ML training and low-selectivity selection during top-k similarity search in the vector embeddings. Finally, the current formats do not provide enough parallel units to fully utilize the computing power of GPUs. Also, unlike the CPUs, more aggressive compression is preferred in the formats with GPU processing because the I/O overhead (including PCIe transfer) dominates the file scan time.

We make the following contributions in this paper. First, we created a feature taxonomy for columnar storage formats like Parquet and ORC. Second, we designed a benchmark to stress-test the formats and identify their performance vs. space trade-offs under different workloads. Lastly, we conducted a comprehensive set of experiments on Parquet and ORC using our benchmark and summarized the lessons learned for the future format design.

2 BACKGROUND AND RELATED WORK

The Big Data ecosystem in the early 2010s gave rise to open-source file formats. Apache Hadoop first introduced two row-oriented formats, **SequenceFile** [54] organized as key-value pairs, and **Avro** [10] based on JSON. At the same time, column-oriented DBMSs, such as C-Store [107], MonetDB [84], and VectorWise [122], developed the fundamental methods for efficient analytical query processing [60]: columnar compression, vectorized processing, and late materialization. The Hadoop community then adopted these ideas from columnar systems and developed more efficient formats.

In 2011, Facebook/Meta released a column-oriented format for Hadoop called **RCFile** [81]. Two years later, Meta refined RCFile and announced the PAX (Partition Attribute Across)-based [64] **ORC** (Optimized Record Columnar File) format [17, 83]. A month after ORC's release, Twitter and Cloudera released the first version of **Parquet** [18]. Their format borrowed insights from earlier columnar storage research, such as the PAX model and the record-shredding and assembly algorithm from Google's Dremel [96].

Since then, both Parquet and ORC have become top-level Apache Foundation projects. They are also supported by most data processing platforms, including Hive [13], Presto/Trino [19, 103], and Spark [20, 118]. Even database products with proprietary storage formats (e.g., Redshift [80], Snowflake [75], ClickHouse [28], and BigQuery [34]) support Parquet and ORC through external tables.

Huawei's **CarbonData** [11] is another open-source columnar format that provides built-in inverted indexing and column groups. Because of its closer relationship with Spark, previous work failed to evaluate the format in isolation [111]. Recent work concludes that CarbonData has a worse performance compared with Parquet and ORC and has a less active community [74].

A number of large companies have developed their own proprietary columnar formats in the last decade. Google's **Capacitor** format is used by many of their systems [3], including BigQuery [97] and Napa [63]. It is based on the techniques from Dremel [96] and Abadi et al. [61] that optimize layout based on workload behavior. YouTube developed the **Artus** format in 2019 for the Procella DBMS that supports adaptive encoding without block compression and $O(1)$ seek time for nested schemas [71]. Meta's **DWRF** is a variant of ORC with better support for reading and encrypting nested data [55]. Meta recently developed **Alpha** to improve the training workloads of machine learning (ML) applications [113].

Arrow is an in-memory columnar format designed for efficient exchange of data with limited or no serialization between different application processes or at library API boundaries [8]. Unlike Parquet or ORC, Arrow supports random access and thus does not require block-based decoding on reads. Because Arrow is not meant for long-term disk storage [5], we do not evaluate it in this paper.

The recent lakehouse [67] trend led to an expansion of formats to support better metadata management (e.g., ACID transactions). Representative projects include Delta Lake [66], Apache Iceberg [15], and Apache Hudi [14]. They add an auxiliary metadata layer and do not directly modify the underlying columnar file formats.

There are also scientific data storage formats for HPC workloads, including HDF5 [37, 78], BP5 [26], NetCDF [44], and Zarr [58]. They target heterogeneous data that has complex file structures, types, and organizations. Their data is typically multi-dimensional arrays and not support column-wise encoding. Although they expose several language APIs (e.g., Python API to interoperate with Pandas and Numpy), few DBMSs support these formats because of their lack of columnar storage features.

Most of the previous investigations on columnar formats target entire query processing systems without analyzing the format internals in isolation [77, 85, 100]. Trivedi et al. compared the read performance of Parquet, ORC, Arrow, and JSON on the NVMe SSDs [111], but they only measured sequential scans with synthetic data sets (i.e., TPC-DS [108]). There are also older industry articles that compare popular columnar formats, but they do not provide an in-depth analysis of the internal design details [1, 2, 4].

Other research proposes ways to optimize these existing columnar formats under specific workloads or hardware configurations [68, 69, 94]. For example, Jiang et al. use ML to select the best encoding algorithms for Parquet according to the query history [86]. BtrBlocks integrates a sampling-based encoding selection algorithm to achieve the optimal decompression speed with network-optimized instances [88]. Li et al. proposed using BMI instructions to improve selection performance on Parquet [90]. None of these techniques, however, have been incorporated in the most popular formats.

3 FEATURE TAXONOMY

In this section, we present a taxonomy of columnar formats features (see Table 1). For each feature category, we first describe the common designs between Parquet and ORC and then highlight their differences as well as the rationale behind the divergence.

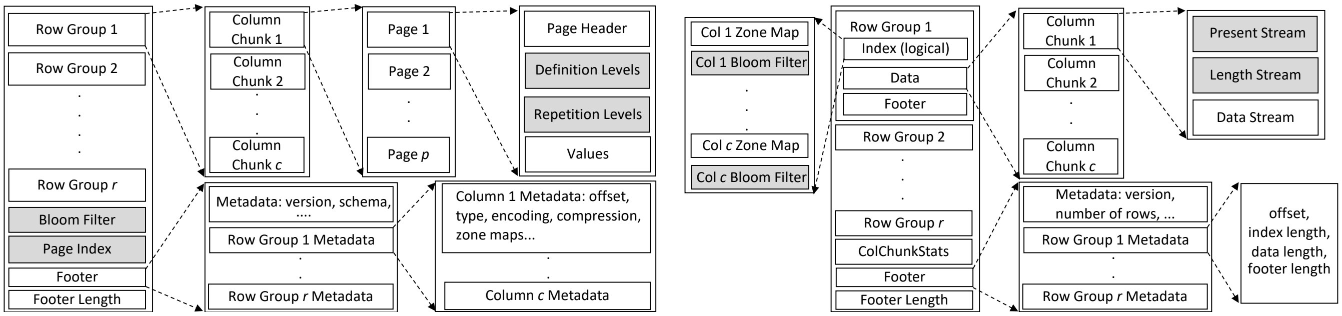
3.1 Format Layout

As shown in Figure 1, both Parquet and ORC employ the PAX format. The DBMS first partitions a table horizontally into row groups. It then stores tuples column-by-column within each row group, with each attribute forming a column chunk. The hybrid columnar layout enables the DBMS to use vectorized query processing and mitigates the tuple reconstruction overhead in a row group. Many systems and libraries, such as DuckDB [31] and Arrow [22], leverage the PAX layout to perform parallel reads: each column chunk can be decoded by a separate thread.

Both formats first apply lightweight encoding schemes (Section 3.2) to the values for each column chunk. The formats then use general-purpose block compression algorithms (e.g., Snappy [36],

		Parquet	ORC
Features	Internal Layout (§3.1)	PAX	PAX
	Encoding Variants (§3.2)	plain, RLE_DICT, RLE, Delta, Bitpacking	plain, RLE_DICT, RLE, Delta, Bitpacking, FOR
	Compression (§3.3)	Snappy, gzip, LZO, zstd, LZ4, Brotli	Snappy, zlib, LZO, zstd, LZ4
	Type System (§3.4)	Separate logical and physical type system	One unified type system
	Zone Map / Index (§3.5)	Min-max per smallest zone map/row group/file	Min-max per smallest zone map/row group/file
	Bloom Filter (§3.5)	Supported per column chunk	Supported per smallest zone map
	Nested Data Encoding (§3.6)	Dremel Model	Length and presence

Table 1: Feature Taxonomy – An overview of the features of columnar storage formats.



(a) Parquet layout.

(b) ORC layout.

Figure 1: Format Layout – Blocks in gray are optional depending on configurations/data.

This Paper	Parquet	ORC
Row Group	Row Group	Stripe
Smallest Zone Map	Page Index (a Page)	Row Index (10k rows)
Compression Unit	Page	Compression Chunk

Table 2: Concepts Mapping – Terms used in this paper and the corresponding ones in the formats.

zstd [59]) to reduce the column chunk's size. The entry point of a Parquet/ORC file is called a footer. Besides file-level metadata such as table schema and tuple count, the footer keeps the metadata for each row group, including its offset in the file and zone maps for each column chunk. For clarity in our exposition, in Table 2 we also summarize the mapping between the terminologies used in this paper and those used in Parquet/ORC.

Although the layouts of Parquet and ORC are similar, they differ in how they map logical blocks to physical storage. For example, (non-Java) Parquet uses a row-group size based on the number of rows (e.g., 1M rows) whereas ORC uses fixed physical storage size (e.g., 64 MB). Parquet seeks to guarantee that there are enough entries within a row group to leverage vectorized query processing, but it may suffer from large memory footprints, especially with wide tables. On the other hand, ORC limits the physical size of a row group to better control memory usage, but it may lead to insufficient entries with large attributes. Different systems configure the row group sizes differently to trade-off between compression ratio, metadata overhead, and scan parallelism. For example, DuckDB sets a relatively small row-group size for Parquet to facilitate parallel reads even with moderate file sizes.

Another difference is Parquet maps its the compression unit to the smallest zone map. ORC provides flexibility in tuning the performance-space trade-off of a block compression algorithm. However, misalignment between the smallest zone map and compression units imposes extra complexity during query processing (e.g., a value may be split across unit boundaries).

3.2 Encoding

Applying lightweight compression schemes to the columns can reduce both storage and network costs [61]. Parquet and ORC support standard OLAP compression techniques, such as Dictionary Encoding, Run-Length Encoding (RLE), and Bitpacking.

Parquet applies Dictionary Encoding aggressively to every column regardless of the data type by default, while ORC only uses it for strings. They both apply another layer of integer encoding on the dictionary codes. The advantage of applying Dictionary Encoding to an integer column, as in Parquet, is that it might achieve additional compression for large-value integers. However, the dictionary codes are assigned based on the values' first appearances in the column chunk and thus might destroy local serial patterns that could be compressed well by Delta Encoding or Frame-of-Reference (FOR) [79, 89, 121]. Therefore, Parquet only uses Bitpacking and RLE to further compress the dictionary codes.

Parquet imposes a limit (1 MB by default) to the dictionary size for each column chunk. When the dictionary is full, later values fall back to "plain" (i.e., no encoding) because a full dictionary indicates that the number of distinct values (NDVs) is too large On the other hand, ORC computes the *NDV-ratio* (i.e., NDV / row count) of the column to determine whether to apply Dictionary Encoding to it. If a column's NDV-ratio is greater than a predefined threshold (e.g., 0.8), then ORC disables encoding. Compared to Parquet's dictionary size physical limit, ORC's approach is more intuitive, and the tuning of the NDV-ratio threshold is independent of the row-group size.

For integer columns, Parquet first dictionary encodes and then applies a hybrid of RLE and Bitpacking to the dictionary codes. If the same value repeats ≥ 8 times consecutively, it uses RLE; otherwise, it uses bitpacking. Interestingly, we found that the RLE-threshold 8 is a non-configurable parameter hard-coded in every implementation of Parquet. Although it saves Parquet a tuning

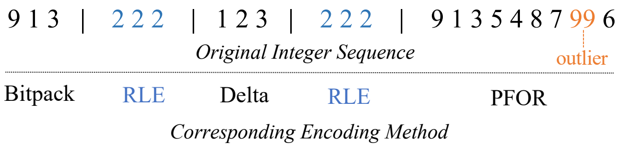


Figure 2: ORC's Hybrid Integer Encoding – Each encoding subsequence has a header for decoder to decide which algorithm to use at run time.

knob, such inflexibility could lead to suboptimal compression ratios for specific data sets (e.g., when the common repetition length is 7).

Unlike Parquet's RLE + Bitpacking scheme, ORC includes four schemes to encode both dictionary codes (for string columns) and integer columns. ORC's integer encoder uses a rule-based greedy algorithm to select the best scheme for each subsequence of values. Starting from the beginning of the sequence, the algorithm keeps a look-ahead buffer (with a maximum size of 512 values) and tries to detect particular patterns. First, if there are subsequences of identical values with lengths between 3 and 10, ORC uses RLE to encode them. If the length of the identical values is greater than 10, or the values of a subsequence are monotonically increasing or decreasing, ORC applies Delta Encoding to the values. Lastly, for the remaining subsequences, the algorithm encodes them using either Bitpacking or a variant of PFOR [121], depending on whether there exist "outliers" in a subsequence. Figure 2 is an example of ORC's integer encoding schemes.

The sophistication (compared to Parquet) of ORC's integer encoding algorithm allows ORC to seize more opportunities for compression. However, switching between four encoding schemes slows down the decoding process and creates more fragmented subsequences that require more metadata to keep track. All the open-source DBMSs and libraries that we surveyed follow Parquet and ORC's default encoding schemes without implementing their own tools for selecting encoding algorithms in the files.

3.3 Compression

Both Parquet and ORC enable block compression by default. The algorithms supported by each format are summarized in Table 1. Because a general-purpose block compression algorithm is type-agnostic (i.e., it treats any data as a byte stream), it is mostly orthogonal to the underlying format layout. Most block compression algorithms contain parameters to configure the "compression level" to make trade-offs between the compression ratio and the compression/decompression speed. Parquet exposes these tuning knobs directly to the users, while ORC provides a wrapper with two pre-configured options, "optimize for speed" and "optimize for compression", for each algorithm.

One of our key observations is that applying block compression to columnar storage formats is unhelpful (or even detrimental) to the end-to-end query speed on modern hardware. Section 5 further discusses this issue with experimental evidence.

3.4 Type System

Parquet provides a minimal set of primitive types (e.g., INT32, FLOAT, BYTE_ARRAY). All the other supported types (e.g., INT8, date, timestamp) in Parquet are implemented using those primitives. For example, INT8 in Parquet is encoded as INT32 internally. Because small integers may be dictionary compressed well, such a "type expansion" has minimal impact on storage efficiency. On the other hand, every type in ORC has a separate implementation with a dedicated reader and writer. Although this could bring more type-specific optimizations, it makes the implementation bloated.

As for complex types, Parquet and ORC both support Struct, List and Map, but Parquet does not provide the Union type like ORC. Union allows data values to have different types for the same column name. Recent work shows that a Union type can help optimize Parquet's Dremel model with schema changes [65].

3.5 Index and Filter

Parquet and ORC include zone maps and optional Bloom Filters to enable selection pruning. A zone map contains the min value, the max value, and the row count within a predefined range in the file. If the range of the values of the zone does not satisfy a predicate, the entire zone can be skipped during the table scan. Both Parquet and ORC contain zone maps at the file level and the row group level. The smallest zone map granularity in Parquet is a physical page (i.e., the compression unit), while that in ORC is a configurable value representing the number of rows (10000 rows by default). Whether to build the smallest zone maps is optional in Parquet.

In earlier versions of Parquet, the smallest zone maps are stored in the page headers. Because the page headers are co-located with each page and are thus discontinuous in storage, (only) checking the zone maps requires a number of expensive random I/Os. In Parquet's newest version (2.9.0), this is fixed by having an optional component called the PageIndex, stored before the file footer to keep all the smallest zone maps. Similarly, ORC stores its smallest zone maps at the beginning of each row group, as shown in Figure 1.

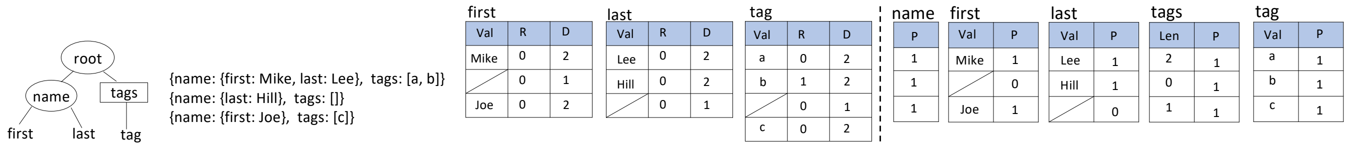
Bloom Filters are optional in Parquet and ORC. The Bloom Filters in ORC have the same granularity as the smallest zone maps, and they are co-located with each other. Bloom Filters in Parquet, however, are created only at the column chunk level partly because the PageIndex (i.e., the smallest zone maps) in Parquet is optional. In terms of the Bloom Filter implementation, Parquet adopts the Split Block Bloom Filter (SBBF) [101], which is designed to have better cache performance and SIMD support [47].

According to our survey, Arrow and DuckDB only adopt zone maps at the row group level for Parquet, while InfluxDB and Spark enable PageIndex and Bloom Filters to trade space for better selection performance [51]. When writing ORC files, Arrow, Spark, and Presto enable row indexes but disable Bloom Filters by default.

Zone maps are only effective when the values are clustered (e.g., mostly sorted). As data processing bottlenecks shift from storage to computation, whether adding more types of auxiliary data structures [82, 91, 106, 119] to the format will be beneficial to the overall query performance remains an interesting open question.

3.6 Nested Data Model

As semi-structured data sets such as those in JSON [42] and Google's Protocol Buffers [49] have become prevalent, an open format must support nested data. Figure 3a shows an example. The nested data model in Parquet is based on Google's Dremel format [96]. As shown in Figure 3b, Parquet stores the values of each atomic field (the leaf nodes in the hierarchical schema) as a separate column.



(a) Example schema and three sample records. **(b)** Parquet's. R/D=Repetition/Definition Level. **(c)** ORC's. Len=length, P=presence.

Figure 3: Nested Data Example – Assume all nodes except the root can be null.

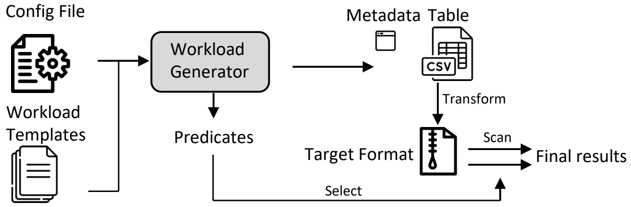


Figure 4: Benchmark Procedure Overview

Each column is associated with two integer sequences of the same length, called the repetition level and the definition level, to encode the nested structure. The repetition levels link the values to their corresponding "repeated fields", while the definition levels keep track of the NULLs in the "non-required fields".

On the other hand, ORC adopts a more intuitive model based on length and presence to encode nested data [97]. As shown in Figure 3c, ORC associates a boolean column to each optional field to indicate value presence. For each repeated field, ORC includes an additional integer column to record the repeated lengths.

For comparison, ORC creates separate presence and length columns for non-atomic fields (e.g., "name" and "tags" in Figure 3c), while Parquet embeds this structural information in the atomic fields via the repetition and definition levels. The advantage of Parquet's approach is that reads fewer columns (i.e., atomic fields only) during query processing. However, Parquet often produces a larger file size because the information about the non-atomic fields could be duplicated in multiple atomic fields (e.g., "first" and "last" both contains the information about the presence of "name" in Figure 3b).

4 COLUMNAR STORAGE BENCHMARK

The next step is to stress-test the performance and space efficiency of the storage formats using data sets using varying value distributions. Standard OLAP benchmarks such as SSB [98], TPC-H [109] and TPC-DS [108] generate data sets with uniform distributions. Second, although some benchmarks, such as YCSB [72], DSB [76], and BigDataBench [116] allow users to set data skewness, the configuration space is often too small to generate distributions that are close to real-world data sets. Lastly, using real-world data is ideal, but the number of high-quality resources available is insufficient to cover a comprehensive analysis.

Given this, we designed a benchmark framework based on real-world data to evaluate multiple aspects of columnar formats. We first define several salient properties of the value distribution of a column (e.g., sortedness, skew pattern). We then extract these properties from real-world data sets to form predefined workloads representing applications ranging from BI to ML. To use our benchmark, as shown in Figure 4, a user first provides a configuration file (or an existing workload template) that specifies the parameter values of the properties. The workload generator then produces the data using this configuration and then generates point and range predicates to evaluate the format's (filtered) scan performance.

4.1 Column Properties

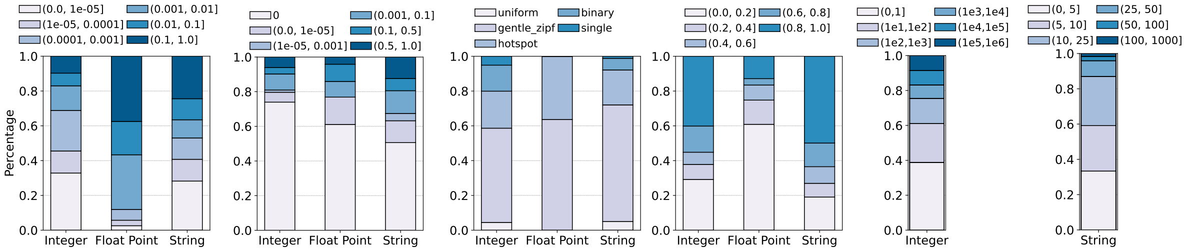
We first introduce the core properties that define the value distribution of a column. We use $[a_1, a_2, ..., a_N]$ to represent the values in a particular column, where N denotes the number of records.

NDV Ratio: Defined as the number of distinct values (NDV) divided by the total number of records in a column: $f_{cr} = \frac{NDV}{N}$. A numeric column typically has a higher NDV ratio than a categorical column. A column with a lower NDV ratio is usually more compressible via Dictionary Encoding and RLE, for example.

Null Ratio: Defined as the number of NULLs divided by the total number of records in a column: $f_{nr} = \frac{|\{i|a_i \text{ is null}\}|}{N}$. It is important for a columnar storage format to handle NULL values efficiently both in terms of space and query processing.

Value Range: This property defines the range of the absolute values in a column. Users pass two parameters: the average value (e.g., 1000 for an integer column) and the variance of the value distribution. The value range directly impacts the compressed file size because most columnar formats apply Bitpacking to the values. For string, this is defined as byte length.

Sortedness: The degree of sortedness of a column affects not only the efficiency of encoding algorithms such as RLE and Delta Encoding, but also the effectiveness of zone maps. Prior work has proposed ways to measure the sortedness of a sequence [95], including *inversions* that counts the number of value pairs with an inversed order, *runs* that counts the number of ascending subsequences, and *exchanges* that counts the least number of swaps needed to bring the sequence in order. Since these metrics do not correlate strongly with encoding efficiency, we developed a simple metric that puts more emphasis on local sortedness. We divide the column into fixed-sized blocks (512 entries by default). Within each block, we compute a sortedness score to reflect its ascending or descending tendency:
$asc = |\{i|1 \le i < n \text{ and } a_i < a_{i+1}\}|$; $desc = |\{i|1 \le i < n \text{ and } a_i > a_{i+1}\}|$
$eq = |\{i|1 \le i < n \text{ and } a_i = a_{i+1}\}|$; $f_{sortness} = \frac{\max(asc, desc) + eq - \lfloor \frac{N}{2} \rfloor}{\lceil \frac{N}{2} \rceil - 1}$
We then take the average of the per-block scores to represent the column's overall sortedness. A score of 1 means that the column is fully sorted, while a score close to 0 indicates a high probability that the column's values are randomly distributed. Although this metric is susceptible to adversarial patterns (e.g., 1, 2, 3, 4, 3, 2, 1), it is sufficient for our generator to produce columns with different sortedness levels. Given a score (e.g., 0.8), we first sort the values



(a) NDV Ratio **(b)** Null Ratio **(c)** Skew Pattern **(d)** Sortedness **(e)** Int Value Range **(f)** String Length

Figure 5: Parameter Distribution – Percentage of total columns from diverse data sets of different parameter values.

in a block in ascending or descending order and then swap value pairs randomly until its sortedness degrades to the target score.

Skew Pattern: We use the following pseudo-zipfian distribution to model the value skewness in a column: $p(k) = \frac{1}{k^s} / (\sum_{n=1}^{C} \frac{1}{n^s})$. C denotes the total number of distinct values, and k refers to the frequency rank (e.g., $p(1)$ represents the portion occupied by the most frequent value). The Zipf-parameter s determines the column skewness: a larger s leads to a more skewed distribution. Based on the range of s, we classified the skew patterns into four categories:

- *Uniform:* When $s \leq 0.01$. Each value appears in the column with a similar probability.
- *Gentle Zipf*: When $0.01 < s \leq 2$. The data is skewed to some extent. The long tail of the values still occupies a significant portion of the column.
- *Hotspot:* When $s > 2$. The data is highly skewed. A few hot values cover almost the entire column.
- *Single/Binary:* This represents extreme cases observed from real-world data where a column contains one/two distinct values.

The skew pattern is a key factor that determines the performance of both lightweight encodings and block compression algorithms.

4.2 Parameter Distribution in Real-World Data

We study the following real-world data sets to depict a parameter distribution of each of the core properties introduced in Section 4.1.

- Public BI Benchmark [50, 114]: real-world data and queries from Tableau with 206 tables (uncompressed 386GB).
- ClickHouse [29]: sample data sets from the ClickHouse tutorials, which represent typical OLAP workloads.
- UCI-ML [6]: a collection of 622 data sets for ML training. We select nine data sets that are larger than 100 MB. All are numerical data excluding unstructured images and embeddings.
- Yelp [57]: Yelp's businesses, reviews, and user information.
- LOG [32]: log information on internet search traffic for EDGAR filings through SEC.gov.
- Geonames [33]: geographical information covering all countries.
- IMDb [40]: data sets that describe the basic information, ratings, and reviews of a collection of movies.

We extracted the core properties from each of the above data sets and plotted their parameter distributions in Figure 5. As shown in Figure 5a, over 80% of the integer columns and 60% of the string columns have an NDV ratio smaller than 0.01. Surprisingly, even for floating-point columns, 60% of them have significant value repetitions with an NDV ratio smaller than 0.1. This implies that Dictionary Encoding would be beneficial to most of the real-world columns. Figure 5b shows that the NULL ratio is low, and string columns tend to have more NULLs than the other data types.

Most columns in the real world exhibit a skewed value distribution, as shown in Figure 5c. Less than 5% of the columns can be classified as *Uniform*. Regardless of the data type, a majority (60 − 70%) of the columns fall into the category of *Gentle Zipf*. The remaining ≈ 30% of the columns contain "heavy hitters". The distribution of the skew patterns indicates that an open columnar format must handle both the "heavy hitters" and the "long tails" (from *Gentle Zipf*) efficiently at the same time.

Figure 5d shows that the distribution of the sortedness scores is polarized: most columns are either well-sorted or unsorted at all. This implies that encoding algorithms that excel only at sorted columns (e.g., Delta Encoding and FOR) could still play an important role. Lastly, as shown in Figure 5e, most integer columns have small values that are ideal for Bitpacking compression. Long string values are also rare in our data set collection (see Figure 5f). We also analyzed real-world Parquet files sampled from public available object store buckets and find that they mostly corroborate Figure 5.

4.3 Predefined Workloads

We extracted the column properties from the real-world data sets introduced in Section 4.1 and categorized them into five predefined workloads: `bi` (based on the Public BI Benchmark), `classic` (based on IMDb, Yelp, and a subset of the Clickhouse sample data sets), `geo` (based on Geonames and the Cell Towers and Air Traffic data sets from Clickhouse), `log` (based on LOG and the machine-generated log data sets from Clickhouse), and `ml` (based on UCL-ML). Table 3 presents the proportion of each data type for each workload, while Table 4 summarizes the parameter settings of the column properties. Each value in Table 4 represents a weighted average across the data types (e.g., if there are 6 integer columns with an NDV ratio of 0.1, 3 string columns with an NDV ratio of 0.2, and 1 float columns with an NDV ratio of 0.4, the value reported in Table 4 would be 0.16). The `classic` workload has a higher Zipf parameter and a higher NDV ratio at the same time, indicating a long-tail distribution. On the other hand, the NDV ratio in `log` is relatively low, but the columns are better sorted. In terms of data types, `classic` and `geo` are string-heavy, while `log` and `ml` are float-heavy.

We then created the `core` workload which is a mix of the five predefined workloads. It contains 50% of `bi` columns, 21% of `classic`,

Type	core	bi	classic	geo	log	ml
Integer	0.37	0.46	0.33	0.31	0.22	0.24
Float	0.21	0.20	0.06	0.08	0.46	0.39
String	0.41	0.34	0.61	0.61	0.32	0.37
Bool	0.003	0.002	0.00	0.00	0.00	0.01

Table 3: Data type distribution of different workloads – Number in the table indicating the proportion of columns.

Properties	core	bi	classic	geo	log	ml
NDV Ratio	0.12	0.08	0.25	0.18	0.08	0.12
Null Ratio	0.09	0.11	0.09	0.13	0.02	0.00
Value Range	medium	small	large	small	small	large
Sortedness	0.54	0.57	0.49	0.45	0.75	0.30
Zipf *s*	1.12	1.10	1.42	0.89	1.26	1.00
Pred. Selectivity	mid	high	high	low	low	mid

Table 4: Summarized Workload Properties – We categorize each property into three levels. The darker the color the higher the number.

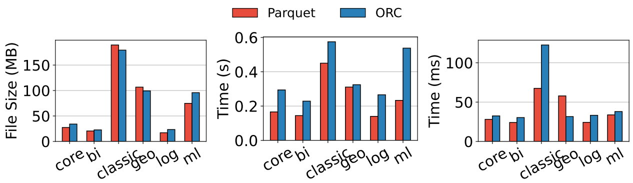
7% of geo, 7% of log, and 15% of ml. We will use core as the default workload in Section 5. For each workload, we also specify a selectivity for our benchmark to generate predicates to evaluate the filtered scan performance of a columnar storage format. As shown in Table 4, bi and classic have high selectivities because these scenarios typically involve large scans. On the contrary, we use a low selectivity in geo and log because their queries request data from small geographic areas or specific time windows.

5 EXPERIMENTAL EVALUATION

In this section, we analyze Parquet and ORC's features presented in Section 3. The purpose is to provide experiment-backed lessons to guide the design of the next-generation columnar storage formats. Section 5.1 describes the experimental setup. Section 5.2 presents the performance and space results of Parquet and ORC under default configurations using the predefined workloads in our benchmark. We then examine the formats' key components with controlled experiments in Sections 5.3 to 5.7. Lastly, we test the formats' ability to support ML workloads (Section 5.8) and GPUs (Section 5.9).

5.1 Experiment Setup

We run the experiments on an AWS i3.2xlarge instance with 8 vCPUs of Intel Xeon CPU E5-2686 v4, 61GB memory, and 1.7TB NVMe SSD. The operating system is Ubuntu 20.04 LTS. We use Arrow v9.0.0 to generate the Parquet and ORC files. For all experiments, we use the following configurations of the formats (unless specified otherwise). Parquet has a row group size of 1m rows, and it sets the dictionary page size limit to 1 MB. The row group size in ORC is 64 MB, and its NDV-ratio threshold for dictionary encoding is v0.8 (Hive's default). Snappy compression is enabled (by default) for both formats. We use the C++ implementation of Parquet (integrated with Arrow C++) [21] and ORC (v1.8) [46] compiled with g++ v9.4. To evaluate page-level zone maps, we use the Rust implementation (v32) of Parquet in Section 5.6. We generate the workloads for the experiments using the benchmark introduced in Section 4. We measure the file sizes and the scan performance (with filters) in these experiments. Each reported measurement is the average of three runs per experiment.



(a) File Size **(b)** Scan Time **(c)** Select Time

Figure 6: Benchmark results with predefined workloads

One approach to measuring the (filtered) scan performance of Parquet and ORC is to decode both formats into Arrow tables. But this approach is unfair because Parquet is tightly coupled with Arrow with native support for format transformation (e.g., Arrow can decode Parquet's dictionary page directly into its dictionary array), while we must convert ORC into an intermediate in-memory representation (ColumnVectorBatch) before transforming it into Arrow tables. Given this, we focus on the raw scan performance of each storage format. We preallocate a fixed-sized memory buffer. After decoding the fixed-size unit of data, the system writes the result to the same buffer, assuming that the previous one has already been consumed by upstream operators.

5.2 Benchmark Result Overview

We first present the results of benchmarking Parquet and ORC with their default configurations using the predefined workloads (Section 4.3). We generate a 20-column table with 1m rows for each workload and store the data in a single Parquet/ORC file. We then perform a sequential scan of file and report the execution time. Lastly, we clear the buffer cache and perform 30 select queries. The selectivities of the range predicates are defined in Table 4, and we report the average latency of the select queries for each workload.

As shown in Figure 6a, there is no clear winner between Parquet and ORC in terms of file sizes. Parquet's file size is smaller than ORC's in log and ml because Parquet applies dictionary encoding on float columns where their NDV ratios are low in real-world data sets (Figure 5a). However, ORC generates smaller files for classic and geo because they mostly contain string data. We provide further analysis of the encoding schemes in Section 5.3.

The results in Figure 6b indicate that Parquet is faster than ORC for scans. The main reason is that Parquet's integer / dictionary code encoding scheme is lightweight: it mostly uses Bitpacking and only applies RLE when value repetition is ≥ 8 (Section 3.2). Because RLE decoding is hard to accelerate using SIMD, it has an inferior performance compared to Bitpacking when the repetition count is small. In contrast, ORC applies RLE more aggressively (when value repetition is ≥ 3, and its integer encoding scheme switches between four algorithms, thus slowing down the decoding process.

Figure 6c shows the average latencies of the select queries. The results generally follow those in sequential scans. The only exception is geo where ORC outperforms Parquet. The reason is that ORC's smallest zone map has a finer granularity than Parquet's (Section 3.5). Compared to other workloads, geo has a relatively high NDV ratio but a low predicate selectivity, which makes fine-grained zone maps more effective.

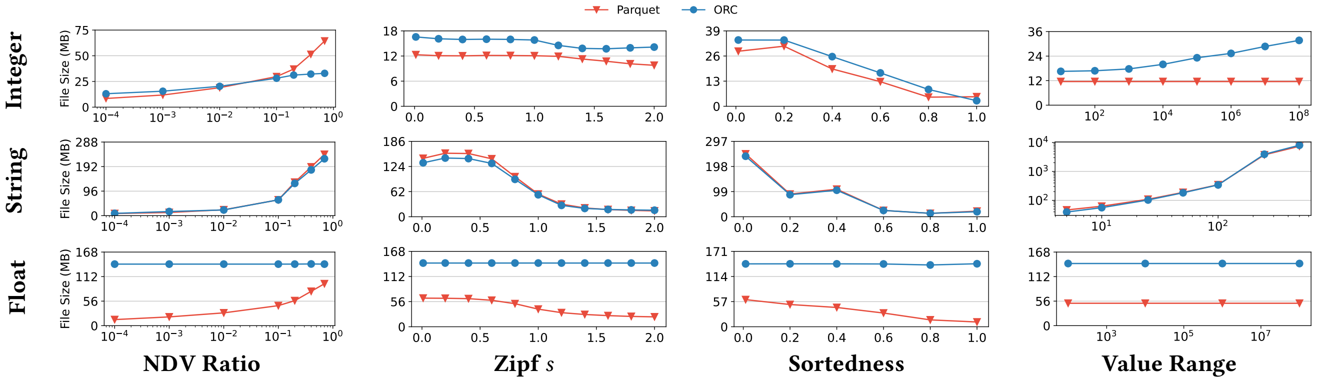


Figure 7: Encoding size differences – Varying parameters on core workload w/o block compression.

5.3 Encoding Analysis

We next examine the performance and space efficiency of the encoding schemes in Parquet and ORC in this section.

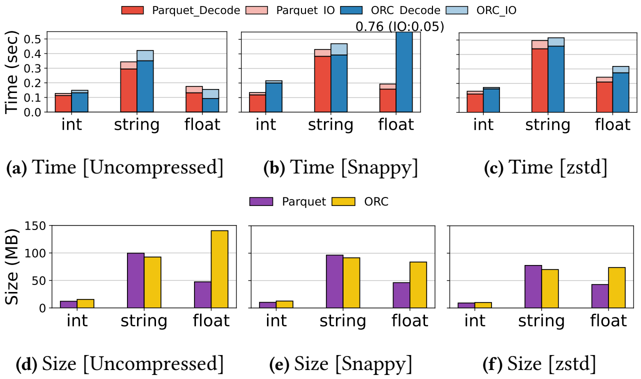
5.3.1 Compression Ratio. To investigate how Parquet and ORC's compression ratios change based on column properties, we generate a series of tables, each having 1m rows with 20 columns of the same data type. For each table, we use the core workload's parameter settings but modify one of the four column properties: NDV ratio, Value Range, Sortedness, and the Zipf parameter. Figure 7 shows how the file size changes when we sweep the parameter of different column properties. We disabled block compression in both Parquet and ORC temporarily in these experiments.

As shown in the first row of Figure 7, Parquet achieves a better compression ratio than ORC for integer columns with a low to medium NDV ratio (which is common in real-world data sets) because Parquet applies Dictionary Encoding on integers before using Bitpacking + RLE. When the NDV ratio grows larger (e.g., > 0.1), this additional layer of Dictionary Encoding becomes less effective than ORC's more sophisticated integer encoding algorithms.

As the Zipf parameter s becomes larger, the compression ratios on integer columns improve for both Parquet and ORC (row 1, column 2 in Figure 7). The file size reduction happens earlier for ORC ($s = 1$) than Parquet ($s = 1.4$). This is because RLE kicks in to replace Bitpacking earlier in ORC (with the run length ≥ 3) than Parquet (with the run length ≥ 8). We also observe that when the integer column is highly sorted, ORC compresses those integers better than Parquet (row 1, column 3 in Figure 7) because of the adoption of Delta Encoding and FOR in its integer encoding.

Parquet's file size is stable as the value range of the integers varies (row 1, column 4 in Figure 7). Parquet applies Dictionary Encoding on the integers and uses Bitpacking + RLE on the dictionary codes only. Because these codes do not change as we vary the value range, the file size of Parquet stays the same in these experiments. On the other hand, the file size of ORC increases as the value range gets larger because ORC encodes the original integers directly.

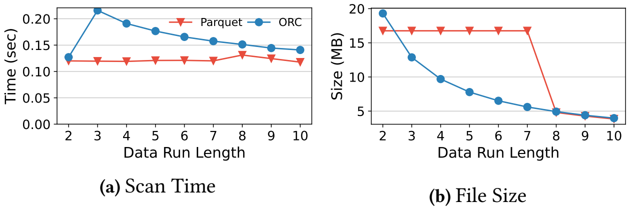
For string columns, as shown in the second row of Figure 7, Parquet and ORC have almost identical file sizes because they both use Dictionary Encoding on strings. ORC has a slight size advantage over Parquet, especially when the dictionary is large because ORC applies encoding on the string lengths of the dictionary entries.



(a) Time [Uncompressed] **(b)** Time [Snappy] **(c)** Time [zstd]

(d) Size [Uncompressed] **(e)** Size [Snappy] **(f)** Size [zstd]

Figure 8: Varying compression on core workload.



(a) Scan Time **(b)** File Size

Figure 9: Varying run length on string, w/o compression.

The third row of Figure 7 shows the results for float columns. Parquet dominates ORC in file sizes because Dictionary Encoding is surprisingly effective on float-point numbers in the real world.

Discussion: Because of the low NDV ratio of real-world columns (as shown in Figure 5), Parquet's strategy of applying Dictionary Encoding on every column seems to be a reasonable default for future formats. However, an encoding selection algorithm such as the one described in [86] is needed to handle the situation when Dictionary Encoding fails. Also, the format should expose certain encoding parameters, such as the minimum run length for RLE [1] for tuning so that users can make the trade-off more smoothly.

5.3.2 Decoding Speed. We next benchmark the decoding speed of Parquet and ORC. We use the data sets in Section 5.3.1 that follow the default core workload. Block compression is still disabled in the experiments in this section. We perform a full table scan on each file and measure the I/O time and the decoding time separately.

[1] The minimum run length for RLE is 8 in Parquet, and it is hardcoded.

	Workload		
Format	**int**	**string**	**float**
ORC	2.9M	3.1M	0.3M
Parquet	0.9M	1.9M	0.6M

Table 6: Branch Mispredictions of Figure 8a.

Encoding			
RLE	**Bitpack**	**Delta**	**PFOR**
.7M(16%)	.7M(32%)	.2M(49%)	.01M(3%)
.2M(46%)	.2M(54%)	0	0

Table 7: Subsequences count and data percentage for integer in Table 6.

Figure 8a shows that Parquet has faster decoding than ORC for integer and string columns. As explained in Section 5.2, there are two main reasons behind this: (1) Parquet relies more on the fast Bitpacking and applies RLE less aggressively than ORC, and (2) Parquet has a simpler integer encoding scheme that involves fewer algorithm options. As shown in Table 6, switching between the four integer encoding algorithms in ORC generates 3× more branch mispredictions than Parquet during the decoding process (done on a similar physical machine to collect the performance counters). According to the breakdown in Table 7, ORC has 4× more subsequences to decode than Parquet, and the encoding algorithm distribution among the subsequences is unfriendly to branch prediction. Parquet's decoding-speed advantage over ORC shrinks for integers compared to strings, indicating a (slight) decoding overhead due to its additional dictionary layer for integer columns. Parquet also optimizes the bit-unpacking procedure using SIMD instructions and code generation to avoid unnecessary branches.

To further illustrate the performance and space trade-off between Bitpacking and RLE, we construct a string column with a pre-configured parameter r where each string value repeats r times consecutively in the column. Recall that ORC applies RLE when $r \geq 3$, while the RLE threshold for Parquet is $r \geq 8$. Figure 9 shows the decoding speed and file sizes of Parquet and ORC with different r's. We observe that RLE takes longer to decode compared to Bitpacking for short repetitions. As r increases, this performance gap shrinks quickly. The file sizes show the opposite trend (Figure 9b) as RLE achieves a much better compression ratio than Bitpacking.

For float columns, ORC achieves a better decoding performance than Parquet because ORC does not apply any encoding algorithms on floating-point values. Although the float columns in ORC occupy much larger space than the dictionary-encoded ones in Parquet (as shown in Figure 7), the saving in computation outweighs the I/O overhead with modern NVMe SSDs.

Discussion: Although more advanced encoding algorithms, such as FSST [70], HOPE [120], Chimp [92] and LeCo [93], have been proposed recently, it is important to keep the encoding scheme in an open format simple to guarantee a fast decoding speed. Selecting from multiple encoding algorithms at run time imposes noticeable performance overhead on decoding. Future format designs should be cautious about including encoding algorithms that only excel at specific situations in the decoding critical path.

In addition, as the storage device gets faster, the local I/O time could be negligible during query processing. According to the float results in Figure 8a, even a scheme as lightweight as Dictionary Encoding adds significant computational overhead for a sequential scan, and this overhead cannot get covered by the I/O time savings. This indicates that most encoding algorithms still make trade-offs between storage efficiency and decoding speed with modern hardware (instead of a Pareto improvement as in the past). Future formats may not want to make any lightweight encoding algorithms “mandatory” (e.g., leave raw data as an option). Also, the ability to operate on compressed data is important with today's hardware.

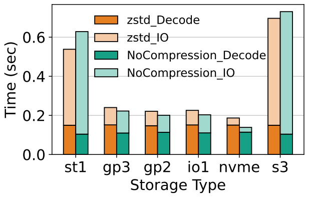


Figure 10: Block Compression

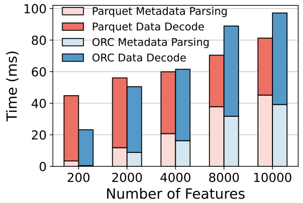
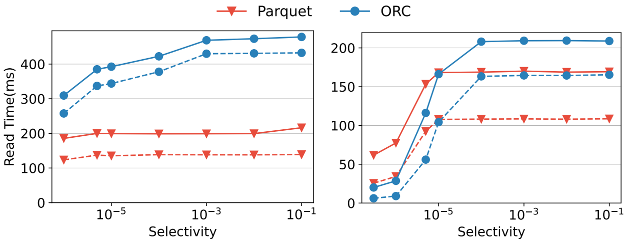


Figure 11: Wide-Table Projection



(a) Low NDV ratio Column

(b) High NDV ratio Column

Figure 12: Select performance with range predicate on float data.

5.4 Block Compression

We study the performance-space trade-offs of block compression on the formats in this section. We first repeat the decoding-speed experiments in Section 5.3.2 with different algorithms (i.e., Snappy [36], Zstd [59]). As shown in Figures 8d to 8f, Zstd achieves a better compression ratio than Snappy for all data types. The results also show that block compression is effective on float columns in ORC because they contain raw values. For the rest of the columns in both Parquet and ORC, however, the space savings of such compression is limited because they are already compressed via lightweight encoding algorithms. Figures 8a to 8c also shows that block compression imposes up 4.2× performance overhead to scanning.

We further investigate the I/O benefit and the computational overhead of block compression on Parquet across different storage-device tiers available in AWS. The x-axis labels in Figure 10 show the storage tiers, where `st1`, `gp3`, `gp2`, and `io1` are from Amazon EBS, while `nvme` is from an AWS i3 instance. These storage tiers are ordered by an increasing read bandwidth. We generate a table with 1m rows and 20 columns according to the `core` workload and store the data in Parquet. The file sizes are 34 MB and 25 MB with Zstd disabled and enabled, respectively. We then perform scans on the Parquet files stored in each storage tier using a single thread.

As shown in Figure 10, applying Zstd to Parquet only speeds up scans on slow storage tiers (e.g., `st1`) where I/O dominates the execution time. For faster storage devices, especially NVMe SSDs, the I/O time is negligible compared to the computation time. In this case, the decompression overhead of Zstd hinders scan performance. The situation is different with S3 because of its high access latency [66]. Reading a Parquet file requires several round trips, including fetching the footer length, the footer, and lastly the column chunks. Therefore, even with multi-threaded optimization to fully utilize S3's bandwidth, the I/O cost of reading a medium-sized (e.g., 10s-100s MB) Parquet file is still noticeable.

Discussion: As storage gets faster and cheaper, the computational overhead of block compression dominates the I/O and storage savings for a storage format. Unless the application is constrained by storage space, such compression should not be used in future formats. Moreover, as more data is located on cloud-resident object stores (e.g., S3), it is necessary to design a columnar format specifically for this operating environment (e.g., high bandwidth and high latency). Potential optimizations include storing all the metadata continuously in the format to avoid multiple round trips, appropriately sizing the row groups (or files) to hide the access latency, and coalescing small-range requests to better utilize the cloud storage bandwidth [9, 68].

5.5 Wide-Table Projection

According to our discussion with Meta's Alpha [113] team, it is common to store a large number of features (thousands of key-value pairs) for ML **training** in ORC files using the "flat map" data type where the keys and values are stored in separate columns. Because each ML training process often fetches a subset of the features, the columnar format must support wide-table projection efficiently. In this experiment, we generate a table of 10K rows with a varying number of float attributes. We store the table in Parquet and ORC and randomly select 10 attributes to project. Figure 11 shows the breakdown of the average latency of the projection queries.

As the number of attributes (i.e., features) in the table grows, the metadata parsing overhead increases almost linearly even though the number of projection columns stays fixed. This is because the footer structures in Parquet and ORC do not support efficient random access. The schema information is serialized in Thrift (Parquet) or Protocol Buffer (ORC), which only supports sequential decoding. We also notice that ORC's performance declines as the table gets wider because there are fewer entries in each row group whose size has a physical limit (64 MB).

Discussion: Wide tables are common, especially when storing features for ML training. Future formats must organize the metadata to support efficient random access to the per-column schema.

5.6 Scan with Predicates

This section evaluates how the index and filter can boost scans with predicates. A scan with predicates contains the following two steps: 1. *select*: Scan the predicate column(s) and output a bitvector containing the matched value positions. 2. *bitvector project*: Use the bitvector to fetch the matched values of projection column(s). Because Parquet C++ does not implement Page Index and Bloom Filter, we use Parquet's Rust implementation in this section.

5.6.1 Select. We first evaluate the performance of select. We generate a table with 10 million rows and 20 columns using the `core` workload and then pick two float columns with NDV ratios of 0.03 and 0.75, respectively. Then we generate range predicates with different target selectivities on the two columns.

The results are in Figure 12. As the selectivity goes down, ORC's reading time decreases faster than Parquet's for both columns. In Figure 12b, ORC outperforms Parquet when selectivity is lower than 10^{-5} even though ORC's full scan performance is worse than Parquet's on float columns. The reason is that ORC's zone map granularity is smaller than Parquet's. Recall that Parquet's smallest zone map granularity is 1 MB page while ORC is 10k values. This means for 8-byte numerical data, Parquet's zone map contains 128k values without encoding and compression. Therefore ORC has more opportunities to skip values using zone maps.

Discussion: Zone maps are only useful on columns where the values are well-clustered. Deciding when to have a zone map (in what granularity) is important for future formats to improve the selection performance with minimal space overhead.

5.6.2 Case Study: Parquet select + bitvector project. Next, we conduct a case study on the performance of a full scan with predicates in Parquet. We did not include ORC because it does not implement bitvector project. We assume the projection columns are all the columns and only one predicate column exists. We use the same table, predicate column, and predicates in Figure 12a.

The results in Figure 13 indicate that when conducting a complete end-to-end selection scan, the Page Index design of Parquet demonstrates efficient data-skipping capabilities through its employment of late materialization (i.e., only the pages containing records indicated by the bitvector produced by select are decoded). Nevertheless, if the selectivity of the query is high (i.e., 0.1 in Figure 13), the computation required for determining the pages to skip surpasses the benefits of skipping. Furthermore, in case of extremely low selectivity, the achievable improvement is bounded by the time to decode a single compression unit.

Discussion: The implementation of the formats should decide wisely on whether to do late materialization (i.e., bitvector project) or a full table scan, depending on predicate selectivity and data distribution. Also, to enable faster low selectivity query, future formats can consider breaking the access granularity limit of a compression unit to enable finer-grained access [43].

5.6.3 Point Query with Bloom Filter. We further evaluate Parquet and ORC's Bloom Filter performance and overhead. False Positive Probability (FPP) of the Bloom Filter is adjustable in both formats. The smaller the FPP, the more space the Bloom Filter takes. We aim to study how the FPP affects each format's space and query time. Because we want the target scenario to be where Bloom Filter can take effect, we use a uniformly distributed 8-byte integer column as the predicate column, with value range [1, 2000000]. We build Bloom Filter on the column with varying FPP in the two formats. Then we run a point query to each file where only 22 rows out of 60 million rows are selected and we record the extra file size compared with the file without building Bloom Filter.

The results are in Figure 14. Figure 14a shows that the Bloom Filter can boost low selectivity point query speed. Interestingly, because ORC's Bloom Filter granularity (10k rows, smallest zone map level) is much finer than Parquet's (1Mi rows, row group level), decreasing FPP to 0.01 helps ORC skip more data. At the same time, there is no benefit to Parquet. Instead, Parquet's performance decreases as FPP is lower, which is also the same for ORC as FPP is below 0.01, because of the more space and decoding overhead.

Figure 14b further shows the extra storage cost introduced by the Bloom Filter. It shows that ORC's Bloom Filter design is more space-efficient than Parquet's. Therefore, although Parquet and

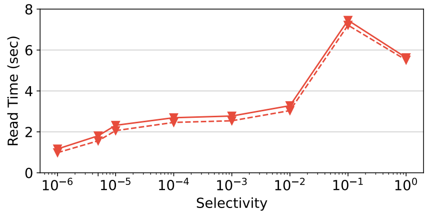
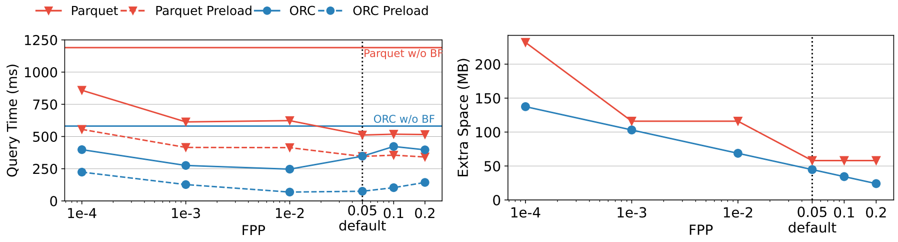


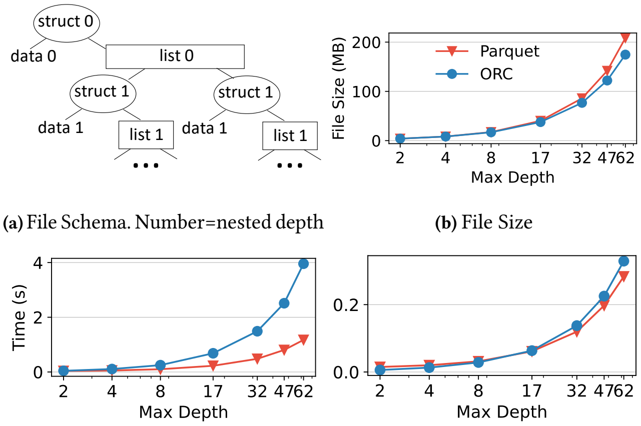
Figure 13: Parquet-selection scan



(a) Query Time

(b) Extra Space

Figure 14: Select performance on varying FPP of Bloom Filter.



(a) File Schema. Number=nested depth

(b) File Size

(c) Time of Scanning to Arrow

(d) Nested Info Decode Overhead

Figure 15: Nested Data Model – Varying max depth in the data.

ORC's performance and space overhead are close under the default FPP level (0.05), ORC offers better options to choose from.

Discussion: The design of Bloom Filter accelerates point query on uniform data, and finer granularity of Bloom Filter brings more benefits to low selectivity queries.

Overall, zone maps and Bloom Filters can boost the performance of low-selectivity queries. However, zone maps are effective only for a smaller number of well-clustered columns, while Bloom Filters are only useful for point queries. When designing future columnar formats, we should consider more advanced indexing and filtering structures, such as column indexes [82, 91, 106] and range filters [112, 119], and study their performance-space trade-offs.

5.7 Nested Data Model

In this section, we quantitatively evaluate the trade-off on the nested data model between Parquet and ORC. To only test the nested model and isolate other noise, we use float data so we can disable encoding and compression on both formats. We test against a synthetic nested schema tree which we design as follows (as shown in Figure 15a): The root node is a struct containing a float field and a list field. The list recursively contains 0-2 structs with the same schema as the root. 97% of the lists contain one struct, and 1% contains no struct. We generate a series of Arrow tables with 256k rows on different max depths of the schema tree and write them into Parquet and ORC. During table generation, the tree of a record stops growing when the depth of the tree reaches the desired max depth. Then we record the file size, the time to read the file into an Arrow table, and the time to decode the nested structure during the table scan.

As shown in Figure 15b, as the depth of the schema tree gets larger, the Parquet file size grows faster than ORC. On the other hand, ORC spends much more time transforming to Arrow (Figure 15c). The reason is that ORC needs to be read into its in-memory data structure and then transformed to Arrow. And the transformation is not optimized. Therefore, we further profile the time decoding the nested information during the scan. The result in Figure 15d shows that ORC's overhead to decode the nested structure information is getting larger than Parquet's as the schema gets deeper. The reason is that ORC needs to decode structure information of struct and list while Parquet only needs to decode leaf fields along with their levels. This result is consistent with Dremel's retrospective work [97].

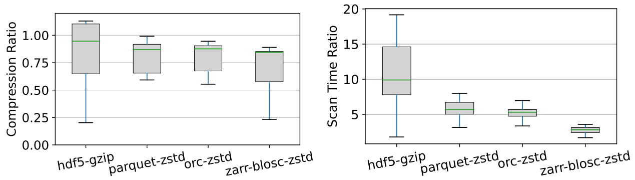
Discussion: The trade-offs between the two nested data models only manifest when the depth is large. Future formats should pay more attention to avoiding extra overhead during the translation between the on-disk and in-memory nested models.

5.8 Machine Learning Workloads

We next investigate how well the columnar formats support common ML workloads. Besides raw data (e.g., image URLs, text) and the associated metadata (e.g., image dimensions, tags), an ML data set often contains the vector embeddings of the raw data, which is a vector of floating-point numbers to enable similarity search in applications such as text-image matching and ad recommendation. It is common to store the entire ML data set in Parquet files [38], where the vector embeddings are stored as lists in Parquet's nested model. Additionally, ML applications often build separate vector indexes directly from Parquet to speed up similarity search [24].

5.8.1 Compression Ratio and Deserialization Performance with Vector Embeddings. In this experiment, we collect 30 data sets with vector embeddings from the top downloaded and top trending lists on Hugging Face and store the embeddings in four different formats: Parquet, ORC, HDF5, and Zarr. We then scan those files into in-memory Numpy arrays and record the scan time for each file. We report the median, 25/75%, and min/max of the compression ratio (format_size / Numpy_size) and the scan slowdown (format_scan_time / disk_Numpy_scan_time) in Figure 16.

Figure 16a shows that none of the four formats achieves good compression with vector embeddings, although Zarr is optimized



(a) Compression Ratio

(b) Scan Time w.r.t Numpy

Figure 16: Efficiency of storing and scanning embeddings

for storing numerical arrays. Zarr, however, incurs a smaller scanning overhead compared to Parquet and ORC, as shown in Figure 16b. This is because Zarr divides a list of (fixed-length) vector embeddings into grid chunks to facilitate parallel scanning/decoding of the vectors. On the other hand, Parquet and ORC only support sequential decoding within a row group.

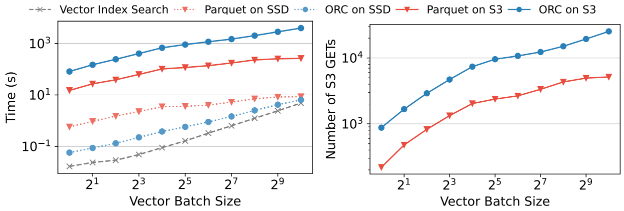
Discussion: Existing columnar formats are less optimized to store and deserialize vector embeddings, which are prevailing in ML data sets. Future format designs should include specialized data types/structures to allow better floating point compression [88, 92, 99] and better parallelism.

5.8.2 Integration with Vector Search Pipeline. Despite the emerging vector databases [27, 48, 115], it is still common to perform the vector search directly in the data lake to avoid the expensive ETL process. Databricks recently announced their vision of Vector Data Lakes to support querying vector embeddings stored in Parquet inside Delta Lake [56]. In this experiment, we evaluate the performance of Parquet and ORC in top-k similarity search queries.

We use the image-text LAION-5B data set [102] with the corresponding vector embeddings. We store the first 100M entries in Parquet/ORC and then use the embeddings from the rest of the data set to perform top-k similarity search queries (k = 10). We maintain an in-memory vector index auto-tuned using the FAISS library [23, 87]. Each query first searches the vector index to get the row IDs of the top 10 most similar entries. The query then uses those row IDs to fetch the corresponding image URLs and text from the underlying columnar storage. We batch the queries to amortize the read amplification to the files.

Figure 17a shows the average time (over 20 trials) of performing the top-k queries with a varying batch size on the x-axis. We repeated the queries using local NVMe SSDs and AWS S3 for storage. We observe that the selection operations in ORC are faster than those in Parquet on local SSDs because ORC includes fine-grained zone maps to reduce the read amplification. As the query batch size gets larger, the performance gap between ORC and Parquet shrinks because the query batch fetches a significant portion of the file. The result is different when the files are stored in S3. Fetching records is much slower in ORC because it issues ≈ 4× S3 GET than Parquet during the process, as shown in Figure 17b. The reason is that the zone maps in ORC are scattered in the row-group footers while those in Parquet are centralized in the file footer.

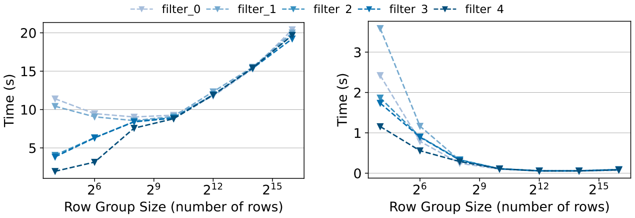
Discussion: ML workloads often involve low-selectivity vector search queries. Although aggressive query batching could amortize the read amplification, fine-grained indexes (e.g., zone maps) are necessary to guarantee the search latency. Also, as more and more large-scale ML data sets reside in data lakes, it is critical for future formats to reduce the number of small reads (e.g., zone map fetches in ORC) to the high-latency cloud object stores.



(a) Time of vector index search vs. selection on files using resulting row IDs

(b) S3 GET requests issued

Figure 17: Top-k Search Workflow Breakdown (k = 10)



(a) With images in projection

(b) Without images in projection

Figure 18: Filterscan on Image Data in Parquet – Filters 0-4 correspond to low to high selectivities. Filters are applied on tabular data.

5.8.3 Storage of Unstructured Data. Besides tabular data, deep learning data sets often include unstructured data such as images, audio, and videos. One approach for storing them in the columnar format is to use their external URLs, as done in the LAION-5B data set above. This approach, however, could suffer from massive http-get requests and invalid URLs over time. Therefore, it is beneficial to store the unstructured data within the same file [39].

We evaluate this on Parquet using the LAION-5B data set with the image URLs replaced by the original binaries. The result Parquet file is 13 GB with 219K rows and is stored on NVMe SSD. We perform scans with five different filters (filter_0 - filter_4) whose selectivities are 1, 0.1, 0.01, 0.001, and 0.0001, respectively. We enable parallel read and pre-buffer of column chunks via Arrow's API. Figure 18a shows the query times when the image column is projected, while Figure 18b presents the query times with only the tabular columns projected. We vary the row-group size of the Parquet file on the x-axis. We observe that a smaller row-group size works better when fetching the images because more row groups allow better parallel read of the large binaries with asynchronous I/Os. A smaller row group, however, compromises the compression of the structured data, and the increased I/O time dominates the latency of queries that only project structured data.

Discussion: It is inefficient to store large binaries with structured data in the same PAX format with a default row-group size. Future designs should separate them in the physical layout of the format while providing a unified query interface logically.

5.9 GPU Decoding

Besides machine learning, GPUs are used to speed up data analytics [104, 117] and decompression [105]. In this section, we investigate the decoding efficiency of Parquet and ORC with GPUs. We use state-of-the-art GPU readers for Parquet and ORC in RAPIDS cuDF 23.10a [52]. The machine for the experiments is equipped with NVIDIA GeForce RTX 3090, AMD EPYC 7H12 with 128 cores, 512GB DRAM, and Intel P5530 NVMe SSD. We generate the data set using the `core` workload with a table of 32 columns and a varying number of rows.

In the first experiment, we scan and decode the files using Arrow (with multithread and I/O prefetching enabled) and cuDF, respectively. As shown in Figure 19a, ORC-cuDF exhibits higher decoding throughput than Parquet-cuDF because ORC has more independent blocks to better utilize the massive parallelism provided by the GPU: the smallest zone map in ORC maps to fewer rows than Parquet's, and each GPU thread block is assigned to each smallest zone map region in cuDF. As the number of rows increases in the files, the decoding throughput of Parquet-Arrow scales because there are more row groups to leverage for multi-core parallel decoding with asynchronous I/O. On the contrary, the Arrow implementation for ORC does not support parallel read.

We further profile the GPU's peak throughput in the above experiment over its theoretical maximum throughput using Nsight Compute [45]. As shown in Figure 19b, the overall compute utilization is low (although the GPU occupancy is full when row count reaches 8M). This is because the integer encoding algorithms used in Parquet and ORC (e.g., hybrid RLE + Bitpacking) are not designed for parallel processing: all threads must wait for the first thread to scan the entire data block to obtain their offsets in the input and output buffers. Moreover, because cuDF assigns a warp to each encoded run, a short run (e.g., a length-3 RLE run in ORC) would cause the threads within a warp underutilized.

We next performed a controlled experiment under the same setting as above to evaluate how block compression affects GPU decoding. The results in Figure 19c show that applying zstd improves the scan throughput for both Parquet and ORC when there are enough rows in the files (i.e., enough data to leverage GPU parallelism). Figure 19d shows the scan time breakdown. We observe that the I/O time (including the PCIe transfer between GPU and CPU) dominates the scan performance, making aggressive block compression pay off.

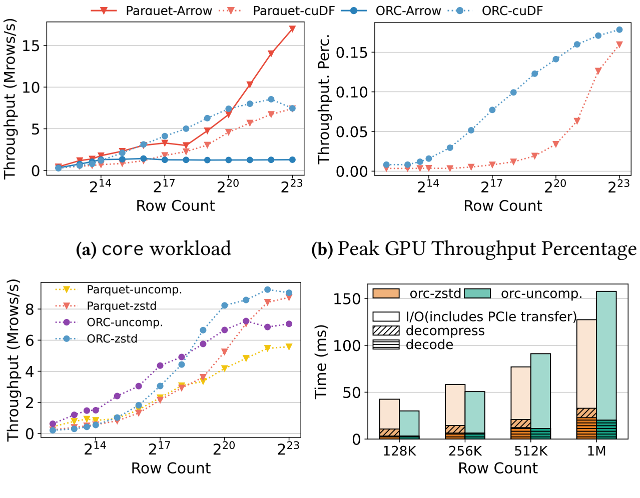
Discussion: Existing columnar formats are not designed to be GPU-friendly. The integer encoding algorithms operate on variable-length subsequences, making decoding hard to parallelize efficiently. Future formats should favor encodings with better parallel processing potentials. Besides, aggressive block compression is beneficial to alleviate the dominating I/O overheads (unlike with CPUs).

6 LESSONS AND FUTURE DIRECTIONS

We summarize the lessons learned from our evaluation of Parquet and ORC to guide future innovations in columnar storage formats.

Lesson 1. Dictionary Encoding is effective across data types (even for floating-point values) because most real-world data have low NDV ratios. Future formats should continue to apply the technique aggressively, as in Parquet.

Lesson 2. It is important to keep the encoding scheme simple in a columnar format to guarantee a competitive decoding performance. Future format designers should pay attention to the performance cost of selecting from many codec algorithms during decoding.



(a) `core` workload

(b) Peak GPU Throughput Percentage

(c) cuDF varying compression

(d) Time breakdown of ORC in (c)

Figure 19: GPU Decoding

Lesson 3. The bottleneck of query processing is shifting from storage to (CPU) computation on modern hardware. Future formats should limit the use of block compression and other heavyweight encodings unless the benefits are justified in specific cases.

Lesson 4. The metadata layout in future formats should be centralized and friendly to random access to better support wide (feature) tables common in ML training. The size of the basic I/O block should be optimized for the high-latency cloud storage.

Lesson 5. As storage is getting cheaper, future formats could afford to store more sophisticated indexing and filtering structures to speed up query processing.

Lesson 6. Future formats should design their nested data models with an affinity to modern in-memory formats to reduce the translation overhead.

Lesson 7. The characteristics of common machine learning workloads require future formats to support both wide-table projections and low-selectivity selections efficiently. This calls for better metadata organization and more effective indexing. Besides, future formats should allocate separate regions for large binary objects and incorporate compression techniques specifically designed for floats in vector embeddings.

Lesson 8. Future formats should consider the decoding efficiency with GPUs. This requires not only sufficient parallel data blocks at the file level but also encoding algorithms that are parallelizable to fully utilize the computation within a GPU thread block.

7 CONCLUSION

In this paper, we comprehensively evaluate the common columnar formats, including Parquet and ORC. We build a taxonomy of the two widely-used formats to summarize the design of their format internals. To better test the formats' trade-offs, we analyze real-world data sets and design a benchmark that can sweep data distribution to demonstrate the differences in encoding algorithms. Using the benchmark, we conduct experiments on various metrics of the formats. Our results highlight essential design considerations that are advantageous for modern hardware and emerging machine learning workloads.

REFERENCES

[1] 2016. File Format Benchmark - Avro, JSON, ORC & Parquet. https://www.slideshare.net/HadoopSummit/file-format-benchmark-avro-json-orc-parquet.

[2] 2016. Format Wars: From VHS and Beta to Avro and Parquet. http://www.svds.com/dataformats/.

[3] 2016. Inside Capacitor, BigQuery's next-generation columnar storage format. https://cloud.google.com/blog/products/bigquery/inside-capacitor-bigquerys-next-generation-columnar-storage-format.

[4] 2017. Apache Arrow vs. Parquet and ORC: Do we really need a third Apache project for columnar data representation? http://dbmsmusings.blogspot.com/2017/10/apache-arrow-vs-parquet-and-orc-do-we.html.

[5] 2017. Some comments to Daniel Abadi's blog about Apache Arrow. https://wesmckinney.com/blog/arrow-columnar-abadi/.

[6] 2022. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets.php. Accessed: 2022-09-22.

[7] 2023. Amazon S3. https://aws.amazon.com/s3/.

[8] 2023. Apache Arrow. https://arrow.apache.org/.

[9] 2023. Apache Arrow Dataset API. https://arrow.apache.org/docs/python/generated/pyarrow.parquet.ParquetDataset.html.

[10] 2023. Apache Avro. https://avro.apache.org/.

[11] 2023. Apache Carbondata. https://carbondata.apache.org/.

[12] 2023. Apache Hadoop. https://hadoop.apache.org/.

[13] 2023. Apache Hive. https://hive.apache.org/.

[14] 2023. Apache Hudi. https://hudi.apache.org/.

[15] 2023. Apache Iceberg. https://iceberg.apache.org/.

[16] 2023. Apache Impala. https://impala.apache.org/.

[17] 2023. Apache ORC. https://orc.apache.org/.

[18] 2023. Apache Parquet. https://parquet.apache.org/.

[19] 2023. Apache Presto. https://prestodb.io/.

[20] 2023. Apache Spark. https://spark.apache.org/.

[21] 2023. Arrow C++ and Parquet C++. https://github.com/apache/arrow/tree/main/cpp.

[22] 2023. Arrow Parquet Multithread Reading. https://arrow.apache.org/docs/python/parquet.html#multithreaded-reads.

[23] 2023. AutoFaiss. https://github.com/criteo/autofaiss.

[24] 2023. AutoFAISS build index API. https://criteo.github.io/autofaiss/API/_autosummary/autofaiss.external.quantize.build_index.html. Accessed: 2023-07-17.

[25] 2023. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[26] 2023. BP5. https://adios2.readthedocs.io/en/latest/engines/engines.html#bp5.

[27] 2023. Chroma. https://github.com/chroma-core/chroma/.

[28] 2023. ClickHouse. https://clickhouse.com/.

[29] 2023. ClickHouse Example Datasets. https://clickhouse.com/docs/en/getting-started/example-datasets.

[30] 2023. Dremio. https://www.dremio.com//.

[31] 2023. DuckDB. https://duckdb.org/.

[32] 2023. EDGAR Log File Data Sets. https://www.sec.gov/about/data/edgar-log-file-data-sets.html.

[33] 2023. GeoNames Dataset. http://www.geonames.org/.

[34] 2023. Google BigQuery. https://cloud.google.com/bigquery.

[35] 2023. Google Cloud Storage. https://cloud.google.com/storage.

[36] 2023. Google snappy. http://google.github.io/snappy/.

[37] 2023. HDF5. https://www.hdfgroup.org/solutions/hdf5/.

[38] 2023. Hugging Face Datasets Server. https://huggingface.co/docs/datasets-server/quick_start#access-parquet-files. Accessed: 2023-07-09.

[39] 2023. image-parquet. https://discuss.huggingface.co/t/image-dataset-best-practices/13974.

[40] 2023. IMDb Datasets. https://www.imdb.com/interfaces/.

[41] 2023. InfluxData. https://www.influxdata.com/.

[42] 2023. JSON. https://www.json.org/.

[43] 2023. Lance. https://github.com/eto-ai/lance.

[44] 2023. NetCDF. https://www.unidata.ucar.edu/software/netcdf/.

[45] 2023. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.

[46] 2023. ORC C++. https://github.com/apache/orc/tree/main/c%2B%2B.

[47] 2023. Parquet Bloom Filter Jira Discussion. https://issues.apache.org/jira/browse/PARQUET-41.

[48] 2023. Pinecone. https://www.pinecone.io/.

[49] 2023. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[50] 2023. Public BI benchmark. https://github.com/cwida/public_bi_benchmark.

[51] 2023. Querying Parquet with Millisecond Latency. https://www.influxdata.com/blog/querying-parquet-millisecond-latency/.

[52] 2023. RAPIDS. https://rapids.ai/.

[53] 2023. Samsung 980 PRO 4.0 NVMe SSD. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/. Accessed: 2023-02-21.

[54] 2023. SequenceFile. https://cwiki.apache.org/confluence/display/HADOOP2/SequenceFile.

[55] 2023. The DWRF Format. https://github.com/facebookarchive/hive-dwrf.

[56] 2023. Vector Data Lakes. https://www.databricks.com/dataaisummit/session/vector-data-lakes/. Accessed: 2023-07-28.

[57] 2023. Yelp Open Dataset. https://www.yelp.com/dataset/.

[58] 2023. Zarr. https://zarr.dev/.

[59] 2023. Zstandard. https://github.com/facebook/zstd.

[60] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.

[61] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data.* 671–682.

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

[63] Ankur Agiwal and Kevin Lai et al. 2021. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *Proceedings of the VLDB Endowment (PVLDB)* 14 (12) (2021), 2986–2998.

[64] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.

[65] Wail Y. Alkowaileet and Michael J. Carey. 2022. Columnar Formats for Schemaless LSM-Based Document Stores. *Proc. VLDB Endow.* 15, 10 (sep 2022), 2085–2097. https://doi.org/10.14778/3547305.3547314

[66] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[67] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*. 8.

[68] Haoqiong Bian and Anastasia Ailamaki. 2022. Pixels: An Efficient Column Store for Cloud Data Lakes. In *2022 IEEE 38th International Conference on Data Engineering (ICDE).* IEEE, 3078–3090.

[69] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. 2017. Wide table layout optimization based on column ordering and duplication. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 299–314.

[70] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[71] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, et al. 2019. Procella: Unifying serving and analytical data at YouTube. (2019).

[72] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.

[73] George P Copeland and Setrag N Khoshafian. 1985. A decomposition storage model. *Acm Sigmod Record* 14, 4 (1985), 268–279.

[74] Dario Curreri, Olivier Curé, and Marinella Sciortino. [n.d.]. *RDF DATA AND COLUMNAR FORMATS.* Master's thesis.

[75] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*.

[76] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[77] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. 2014. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1295–1306.

[78] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases.* 36–47.

[79] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering.* IEEE, 370–379.

[80] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD*.

[81] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 2011. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering.* IEEE, 1199–1208.

[82] Brian Hentschel, Michael S Kester, and Stratos Idreos. 2018. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of*

the 2018 International Conference on Management of Data. 857–872.

[83] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1235–1246.

[84] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* (2012).

[85] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32, 5 (2020), e5523.

[86] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A Chien, Jihong Ma, and Aaron J Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *Proceedings of the 2021 International Conference on Management of Data*. 843–856.

[87] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

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

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

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

[91] Yinan Li and Jignesh M Patel. 2013. Bitweaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 289–300.

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

[93] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2023. LeCo: Lightweight Compression via Learning Serial Correlations. *arXiv preprint arXiv:2306.15374* (2023).

[94] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David Cohen, Timothy Mattson, and Nesime Tatbul. 2022. Self-Organizing Data Containers. In *The Conference on Innovative Data Systems Research, CIDR*.

[95] Heikki Mannila. 1985. Measures of presortedness and optimal sorting algorithms. *IEEE transactions on computers* 100, 4 (1985), 318–325.

[96] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[97] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, et al. 2020. Dremel: A decade of interactive SQL analysis at web scale. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3461–3472.

[98] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). *Pat* 200, 0 (2007), 50.

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

[100] Pouria Pirzadeh, Michael Carey, and Till Westmann. 2017. A performance study of big data analytics platforms. In *2017 IEEE international conference on big data (big data)*. IEEE, 2911–2920.

[101] Felix Putze, Peter Sanders, and Johannes Singler. 2010. Cache-, Hash-, and Space-Efficient Bloom Filters. *ACM J. Exp. Algorithmics* 14, Article 4 (Jan 2010), 18 pages.

[102] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, Patrick Schramowski, Srivatsa Kundurthy, Katherine Crowson, Ludwig Schmidt, Robert Kaczmarczyk, and Jenia Jitsev. 2022. LAION-5B: An open large-scale dataset for training next generation image-text models. In *NeurIPS*.

[103] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[104] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

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

[106] Lefteris Sidirourgos and Martin Kersten. 2013. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 893–904.

[107] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 553–564.

[108] The Transaction Processing Council. 2021. TPC-DS Benchmark (Revision 3.2.0).

[109] The Transaction Processing Council. 2022. TPC-H Benchmark (Revision 3.0.1).

[110] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[111] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. 2018. Albis:{High-Performance} File Format for Big Data Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 615–630.

[112] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1632–1644.

[113] Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared Foundations: Modernizing Meta's Data Lakehouse. In *The Conference on Innovative Data Systems Research, CIDR*.

[114] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages.

[115] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[116] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499. https://doi.org/10.1109/HPCA.2014.6835958

[117] Bobbi W Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2491–2503.

[118] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.

[119] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.

[120] Huanchen Zhang, Xiaoxuan Liu, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-preserving key compression for in-memory search trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1601–1615.

[121] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 59–59.

[122] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. 2012. Vectorwise: A vectorized analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1349–1350.