A Deep Dive into Common Open Formats for Analytical DBMSs

Chunwei Liu
MIT CSAIL
chunwei@csail.mit.edu

Anna Pavlenko
Microsoft
annapa@microsoft.com

Matteo Interlandi
Microsoft
mainterl@microsoft.com

Brandon Haynes
Microsoft
brhaynes@microsoft.com

ABSTRACT

This paper evaluates the suitability of Apache Arrow, Parquet, and ORC as formats for subsumption in an analytical DBMS. We systematically identify and explore the high-level features that are important to support efficient querying in modern OLAP DBMSs and evaluate the ability of each format to support these features. We find that each format has trade-offs that make it more or less suitable for use as a format in a DBMS and identify opportunities to more holistically co-design a unified in-memory and on-disk data representation. Our hope is that this study can be used as a guide for system developers designing and using these formats, as well as provide the community with directions to pursue for improving these common open formats.

PVLDB Reference Format:
Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. A Deep Dive into Common Open Formats for Analytical DBMSs . PVLDB, 16(11): 3044 - 3056, 2023.
doi:10.14778/3611479.3611507

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Tranway1/ColumnarFormatsEval.

1 INTRODUCTION

Over the last decade, a number of new common and open formats have been proposed that purport to improve performance and ease interoperation across OLAP DBMSs and storage layers such as data lakes [49]. Today, storage formats such as Parquet [12] and ORC [11] are the cornerstone reference architectures for cloud-scale data warehousing systems [14]. At the same time, the in-memory format Apache Arrow [8] is widely considered to be the default means of interoperation across different data systems [50], and several systems are even exploring how to leverage it end-to-end [18]. Each of these in-memory and storage formats attempt to minimize disk, memory, and IO costs, and each applies a wide variety of optimizations to maximize analytic read performance.

Though the benefits for common open formats are now well established [29], there has been less exploration of the relative benefits of these formats for *direct subsumption* in an analytical DBMS as a native format. This is despite the robust discussion in database community about the relative merits of these formats for this purpose (e.g., [3, 43, 58]). One reason for this is that each format makes design choices that optimize for its use as a common and open format, and these choices often conflict with longstanding analytical DBMS techniques. For example, we increasingly see a push to directly leverage Arrow as an end-to-end, in-memory format within a DBMS [37]. At the same time, DBMS in-memory columnar formats typically encode data [4, 20] to minimize space and reduce memory requirements. However, Apache Arrow by default provides no encoding support, leaving it at odds with typical DBMS design. On the other hand, on-disk formats such as Parquet arrange data in a form that is much closer to that found in modern columnar DBMSs (e.g., by employing run-length encoding mixed with dictionary encoding and bit packing). However, Parquet is widely leveraged only as storage format and exposes no dedicated in-memory representation. Instead, developers bring Parquet data into memory and convert it to the Arrow format, which, as stated above, is suboptimal for a columnar DBMS. Finally, a format such as ORC at first glance appears to offer the best of both worlds, since it provides both an efficiently encoded data format and a related in-memory representation. Nevertheless, Arrow and Parquet are considered the standard nowadays because of their popularity in terms of activity in open-source projects and support from big data frameworks and large-scale query providers [30].

Given this environment, the goal of this paper is to evaluate these three formats, explore their trade-offs, and evaluate their performance as candidates for direct subsumption in an analytical DBMS. Three main challenges exist in subsuming an in-memory format such as Arrow or traditionally on-disk formats such as Parquet or ORC. First, a DBMS needs to be able to efficiently (de)serialize and (de)compress on-disk data to and from an in-memory representation. For this, efficiency directly depends on format's compression ratio, decompression speed, and transcoding performance. These trade-offs can be subtle and the line between an "in-memory" or "on-disk" format is often blurry. For example, in some cases a DBMS could improve performance by writing Arrow to disk or directly operating on Parquet in memory, avoiding transcoding costs and taking advantage of the features offered by each format.

Second, prior work has established that it is highly advantageous for a DBMS to "push down" computation as far as possible (e.g., to disk coprocessors or into the compressed domain) and to do so over as many data types as possible [24, 56]. Computation pushdown subsumes a number of related techniques. *Column pruning* and *data skipping* respectively enable a DBMS to avoid decompressing columns or rows that do not contain data relevant to a query answer (e.g., when executing a range query by skipping data regions that do not contain data within the range). Techniques such as *direct querying* enable a DBMS to retrieve query answers without an expensive decode or decompression step [5, 6, 59]. As we show in Table 1, the ability for common in-memory and on-disk formats to support these techniques is uneven; to maximize performance a DBMS should optimize for the resulting trade-offs.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611507

Finally, to maximize performance, modern DBMSs leverage modern techniques such as vectorized execution (e.g., SIMD) [27, 31, 33, 38, 40, 55] and query compilation [19]. Here again the ability for a format to support these techniques is uneven. For example, Parquet can vectorize operations over some query types whereas Arrow is inherently unencoded and more amenable to vectorized execution.

In summary, in this paper we present the first detailed, empirical evaluation of three popular and increasingly-adopted formats and evaluate their suitability to be used as a native format in a DBMS. Our hope is that this study can be used as a guide for system developers using these formats, as well as provide the community with directions to pursue for improving these common open formats.

Our contributions include:

- We succinctly summarize the design nuances and distinctions of three widely-adopted open columnar formats: Apache Arrow, Parquet and ORC (Section 3).
- We systematically identify and explore the high-level features that are important to support efficient querying in modern OLAP DBMSs (Section 4).
- For each format, we evaluate its ability to support efficient encoding, compression, and transcoding (Section 5) for both real-world, synthetic datasets, and various data types.
- We benchmark the ability of each format to support select-project (SP) operations found near the leaves of query plans. We evaluate these in isolation (Section 6) and in combination (Section 7) using TPC-DS query plan fragments and over various data types.
- We evaluate the ability for each format to take advantage of recent trends such as vectorization, query compilation, and direct querying (Section 8).
- We identify key opportunities to holistically co-design a unified in-memory and on-disk data representation.

2 BACKGROUND: COMPRESSION AND DATA ENCODING

Data systems employ compression algorithms to reduce on-disk or in-memory data sizes and improve bandwidth utilization [25]. Conventional compression has traditionally focused on minimizing file size. This focus on size alone, while appropriate for storage, overlooks DBMS query execution performance [33, 46, 51]. Conversely, in an analytical columnar DBMS, compressed size is usually balanced with the ability to query directly on the compressed data.

2.1 Compression

Because of their generality, byte-oriented compression techniques (e.g., Gzip [17], Snappy [23], and Zlib [22]) are widely used to reduce data size [5, 46]. They treat the input values as a byte stream and compress them sequentially. Byte-oriented compression is applicable to all data types and, in general, exhibits good compression ratios [33]. However, these methods are computationally intensive [5]. A data block needs to be fully decompressed before individual values can be accessed. This often introduces unnecessary overhead for query execution.

2.2 Encoding

For decades, many data engines used row-oriented storage formats for OLTP query workloads [3]. As more complex OLAP workloads became common, *columnar storage formats* began to predominate [6]. Columnar databases store data of the same type together [52], allowing systems to leverage *lightweight compression*, also referred to as *encoding* [5]. Encoding methods such as *dictionary encoding*, *run-length encoding*, and *bit-packed encoding* are typically designed to compress a specific type of data, enabling efficient compression and better record-level access relative to the general-purpose compression approaches described in the previous section.

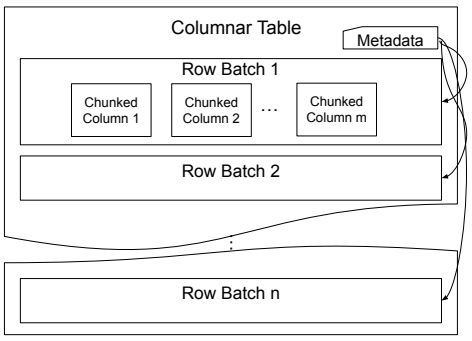


Figure 1: Columnar format layout.

Some encoding methods also support direct querying and data skipping to improve query performance [5, 33, 39]. Systems such as Redshift [26] and SQL Server [35] support many lightweight compression approaches that reduce the storage cost of data; at the same time, they apply compression-specific optimizations to improve query execution performance. Previous research [16, 32, 40, 41] has also demonstrated that, for specific datasets, good encoding achieves a comparable compression ratio with far fewer CPU cycles than does byte-oriented compression algorithms.

We next give an overview of several popular encoding algorithms referred to in this paper and highlight their applicable scenarios.

Bit-Packed Encoding (BP) works on numerical data. It finds the minimal number of bits needed to represent values and removes superfluous leading zeros. It works best when the target numbers have similar bit-width.

Dictionary Encoding (DICT) works on all data types. It encodes each distinct entry with an integer key and bit-packs the integers. Dictionary encoding works best when the dataset has small cardinality and many repetitions. Queries on dictionary encoded data can be applied either on the fully decoded data or directly in the encoded domain after query rewriting using dictionary translation.

Run-Length Encoding (RLE) works on data with many consecutive repetitions. It replaces a run of the same value with a pair consisting of the value and how many times it is repeated.

Hybrid Encodings are derived from the above encoding techniques. Dictionary run-length encoding (**DICT-RLE**) applies RLE on the dictionary encoded keys to further compress data. Bit-packed and run-length hybrid encoding is used as a default implementation for the Parquet RLE encoder. Hybrid encoding usually achieves better compression performance at the cost of performance.

3 COLUMNAR OPEN FORMATS

In big data environments today, there are many optimized data formats for columnar data storage and computation, as we show in Table 1. Interestingly, these data formats share the same basic

Table 1: A comparison of the features found in common open columnar data formats.

	Encoding Methods	Compression Codecs	Skipping	Direct Query	Primary Purpose	Representative Systems
Arrow	DICT	None	Chunk-level	None	In-Memory Compute	Dremio, Spark, Pandas, etc.
Feather	DICT	Zstd, LZ4	None	None	On-Disk Storage	Pandas
Parquet	DICT(-RLE), RLE, BP, Delta, etc.	Gzip, Snappy, Zstd, LZ4, (LZO)	Record-level	None	On-Disk Storage	Spark, Hive, Presto, etc.
ORC	DICT, RLE, BP, Delta	Snappy, Zlib, LZ4	Chunk-level	None	On-Disk Storage	Hive, Presto, etc.

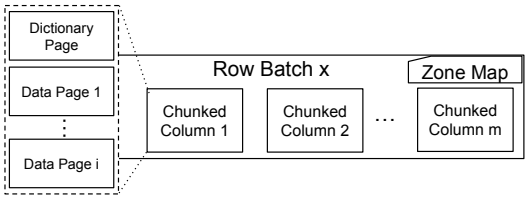


Figure 2: A Parquet row batch.

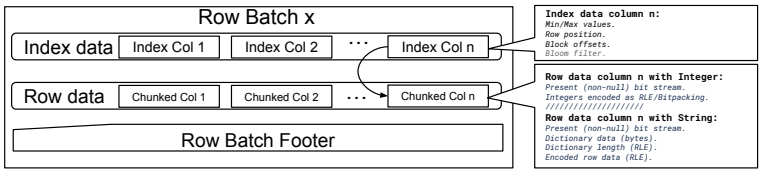


Figure 3: An ORC row batch.

Table 2: Column format name convention mapping.

	Row Batch	Chunked Column
Arrow	Record Batch	Chunked Array
Parquet	Row Group	Column Chunk
ORC	Stripe	Row Column

underlying design. Therefore, we begin by providing a "generic" architecture that summarizes the substantial commonality in modern columnar data format design (Section 3.1). Then, we drill into the idiosyncrasies found in Arrow, Parquet and ORC (Sections 3.2, 3.3, and 3.4, respectively). To ease the parsing for non-familiar readers, we take the liberty of providing a unified naming convention. Table 2 has the mapping between our naming and each format.

3.1 Open Columnar Formats 101

Figure 1 summarizes a generic columnar format design. Columnar storage formats physically arrange data such that all the records belonging to the same column are stored sequentially. To achieve better data access at scale, columnar formats partition columns into *chunks*. *Chunked columns* are not created arbitrarily; instead, row-level alignment is attained by first splitting a table horizontally into *row batches* where, within each batch, rows are then partitioned into column chunks. Metadata about the row batches (e.g., their location, number, length, compression algorithm, etc.) are stored either into the footer or in the preamble of the file.

3.2 Apache Arrow (Feather)

Apache Arrow [8] is a columnar data structure supporting efficient in-memory computing. Arrow can represent both flat and hierarchical data. Arrow is designed to be complementary to on-disk columnar data formats such as Parquet and ORC, and in fact it shares with them the same design depicted in Figure 1. On-disk data files are decompressed, decoded, and loaded into Arrow in-memory columnar arrays. Each row batch has a default size of 64K rows. Arrow column chunks have a *present* bit-vector signaling whether a value is null (or not), and, for strings, optionally a dictionary.

The Arrow columnar format has some compelling properties: random access is $O(1)$ for entries in the same chunked column, and each value cells are sequential in memory, so it's efficient to iterate over. Arrow also defines a binary serialization protocol for converting a collection of row batches that can be used for messaging, interprocess communication (IPC), and writing blobs into storage. Deserializing an Arrow blob has effectively zero cost.

Closely related to the Arrow format, *Arrow Feather* [10] is a column-oriented binary disk-based format, leveraging the same IPC as the in-memory Arrow format. Additionally, Feather adds dictionary encoding (for strings) and compression (Zstd, LZ4). Datasets stored in Arrow Feather are loaded in-memory as *Arrow Tables*.

3.3 Parquet

Parquet [12] is a columnar-oriented storage format inspired by the nested data storage format outlined in Google Dremel [44]. Parquet integrates many efficient compression and encoding approaches to achieve space-efficiency. A Parquet file is structured almost exactly as described in Section 3.1; however, as illustrated in Figure 2, each column chunk is partitioned into a *dictionary page* and series of *data pages*. Its file footer additionally contains *zone maps* (e.g., min, max, and number of NULLs) at the row batch, chunked column, and data page level. This enables efficient data skipping. Row batches have a recommended size of 512-1024 MB. Parquet applies dictionary encoding per data page and falls back to plain encoding when a dictionary grows larger than a predefined threshold. Parquet is designed to be space and IO-efficient at the expense of CPU utilization for decoding. It does not provide any data structures for in-memory computing.

3.4 ORC

Optimized Row Columnar (ORC) [11] is a storage format designed for read-heavy analytical workloads. ORC files are organized as in Figure 1 where the default row batch size is 250 MB. Differently than Parquet, as is shown in Figure 3, ORC organizes columns into an *index* that contains min/max values, bloom filters, etc., and *row data* with a *present* bit-vector indicating NULL entries. The chunked columns in the row data are formatted based on the encoding type.

ORC exposes a corresponding in-memory format, which contains a row-level index and NULL bit-vector data structures for fast querying and NULL checks. ORC supports dictionary encoding (at the row batch level) for string data. Similar to Parquet, ORC falls back to plain encoding when the number of distinct values is greater than a threshold (e.g., for Hive, 80% of the records).

Table 3: Default encoding by format and data type. Parquet uses DICT as default in the latest C++ API, while DICT-RLE was used in its legacy Java API. Int-RLE refers to the encoding where the decimal value is scaled to an integer and then encoded with RLE encoding.

	Integer	Double	String/Binary	Decimal
Parquet	DICT(-RLE)	DICT(-RLE)	DICT(-RLE)	DICT(-RLE)
Arrow	None	None	DICT	None
ORC	RLE	None	DICT-RLE	Int-RLE

3.5 Discussion

Overall, Parquet and ORC provide the most comprehensive compression support for common data types, whereas Arrow Feather supports the fewest. ORC provides more auxiliary information for query execution (e.g., its zone map and support for bloom filters). Arrow Feather applies the same compression type to all arrays in the same record batch, whereas Parquet is more granular and allows compression to vary across column chunks. This flexibility enables intelligent encoding and compression selection based on the data features or workload characteristics [33]. As summarized in Table 3, each format applies different default encoding strategies.

In terms of data access, both Arrow and ORC require data to be fully loaded into dedicated in-memory data structures (an Arrow Table or ORC `ColumnVectorBatch`, respectively) before further query execution can begin. On the other hand, Parquet exposes a *streaming API* that allows pipelining data parsing and query execution, leading to more optimization opportunities. However, Parquet does not itself provide any dedicated in-memory data structures.

4 METHODOLOGY

In the subsequent sections, we benchmark the performance of Arrow, Parquet, and ORC over (non-nested) relational data. This is not strictly an apples-to-apples comparison because each format was developed with a different use case in mind: Arrow eases the sharing of in-memory data across systems, Parquet is a generic on-disk format, and ORC is a storage format for relational big data systems. Nevertheless, this comparison is important for evaluating the design choices (e.g., encoding method, compression, implementation decisions) made by each format and to understand the limitations and opportunities when using these formats in analytical DBMSs.

Dimensions. To most fairly compare the formats, we evaluate each format across the following dimensions:

1. **Compression ratio.** Each format applies different encoding methods and supports different compression algorithms. The final achievable compression ratio is a result of these decisions, and so we evaluate each format using the variously supported encoding and compression algorithms (Section 5.1).
2. **Transcoding throughput.** While compression ratio alone is sufficient if we care only about minimizing disk or memory usage, this comes at the cost of having to compress, convert, and decompress (i.e., transcode) the data when accessing it (Section 5.2).
3. **Data access.** For each data type, what are the costs of accessing them? Data is often accessed by column (i.e., projected) or filtered using a predicate. We evaluate the costs of each format when applying simple data access operations (Section 6).
4. **End-to-end evaluation over subexpressions.** Since we care about the performance of each format when evaluating analytical queries, we explore the performance of each format over a set of query subexpressions drawn from TPC-DS (Section 7).
5. **Advanced features.** Given the many trade-offs baked into each format, we explore the extent to which we can extend them to support novel features such as computation pushdown into the encoded domain and hardware acceleration (Section 8).

To summarize our findings, in Table 4 we show the structure of our experiments and the overall best format for each dimension.

Setup. All experiments are performed on an Azure Standard D8s v3 (8 vCPUs, 32 GiB memory), premium SSD LRS, and Ubuntu 18.04. We test Apache Arrow 5.0.0, ORC 1.7.2, and the Apache Parquet Java API version 1.9.0. Where needed, we use the Apache Arrow C++ library to write in-memory Arrow tables to disk. We perform experiments using (i) the TPC-DS dataset at scale 10, (ii) the Join Order Benchmark (JOB) [1], (iii) the Public BI Benchmark (BI) [2], and (iv) real-world datasets drawn from public data sources including GIS, machine learning, and financial datasets (CodecDB) [33]. For all the experiments, we report numbers when the system caches are cold by default. For selected experiments we also report numbers when caches have been warmed up, i.e., to simulate frequently accessed datasets. Unless stated otherwise, we use each format's default settings. Different results could certainly be obtained if dataset-specific parameter tuning were applied to each format. However, such fine-grained configuration tuning is beyond the scope of the paper and left as future work.

5 COMPRESSION AND TRANSCODING

In this section, we evaluate the compression performance of Arrow, Parquet, and ORC (Section 5.1) and the related costs for transcoding data from compressed to in-memory formats (Section 5.2).

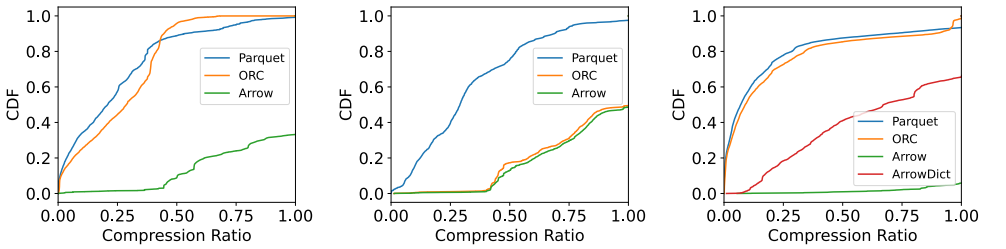
5.1 Encoding & Compression Performance

We first explore the compression performance of each format through three sets of experiments. In the first experiment (Section 5.1.1) we evaluate how each format's supported encodings perform over a set of real-world datasets. The last two experiments leverage TPC-DS [45] to illustrate the performance of each format when compression is applied on top of encodings. For the synthetic experiments on TPC-DS, we begin by evaluating compression algorithms over the full dataset (Section 5.1.2), and then explore how compression performance varies by data type (Section 5.1.3).

5.1.1 Encoding Performance over Real-World Datasets. In this experiment, we group each data column by data type, convert each column one-by-one into each format, and finally aggregate the statistics of the compressed columns. Table 5 and Table 6 show the overall compression performance and statistics over the ~31k columns in the CodecDB, Public BI and JOB, datasets. We further show the compression ratio CDFs for each data type in Figure 4 where we focus on the effective compression ratio range $(0.0, 1.0)$. Finally, Figure 5 shows, for each data type, CDFs that takes into account the number of distinct values in each column. To avoid

Table 4: Evaluation overview and key results.

Evaluation dimension	Best Overall	Key Advantage	Section
Compression ratio	Parquet	Comprehensive encoding and compression support	5.1
Compression throughput	Arrow Feather	Fast serialization	5.2.1
Decompression throughput	Arrow Feather	Fast deserialization	5.2.2
Projection evaluation	Parquet and ORC	Fine-grained skipping while loading data	6.1
Predicate evaluation	ORC	Fine-grained loading control with dedicated in-memory representation	6.2
Bitmap evaluation	ORC	Fine-grained loading control with dedicated in-memory representation	6.2.3
Subexpression evaluation	ORC	Fine-grained loading control with dedicated in-memory representation and efficient skipping	7
Direct querying	Parquet	In-memory mapping with data skipping and direct querying	8
Vectorized execution	Parquet	In-memory mapping with data skipping, direct querying, and SIMD support	8



(a) Integers (b) Floats (c) Strings

Figure 4: Column compression ratio CDFs over the CodecDB, BI and JOB datasets.

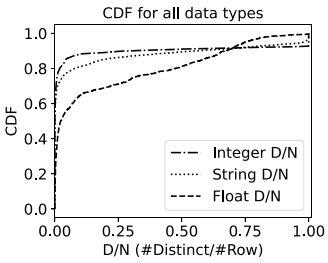


Figure 5: Distinct value CDFs.

Table 5: Total size (in GB) by format for columns in the CodecDB, BI, and JOB datasets. We serialize each column separately into each format and group their compressed size by data type. The raw dataset is in CSV format. For Arrow, we report with dictionary encoding enabled (Arrow DICT) and disabled (the default). We copy the file size (marked with *) from the Arrow default column for CR computation as there is no dictionary support for integer and float types.

Data Type	# Cols.	Raw Size	Parquet Size	ORC Size	Arrow Size	Arrow (DICT)
Integer	12k	57.3	9.8	13.5	59.3	59.3*
Float	7k	58.8	24.0	58.2	59.8	59.8*
String	13k	373.5	31.0	62.2	403.4	118.3
Total	31k	489.7	64.7	133.9	522.5	237.4
Compression Ratio (CR)			0.13	0.27	1.07	0.48

Table 6: Average and stddev compression ratios by data type.

Type	Parquet		ORC		Arrow		ArrowDICT	
	AVG	STD	AVG	STD	AVG	STD	AVG	STD
Int	0.25	0.27	0.26	0.18	1.41	0.84	-	-
Float	0.34	0.26	1.43	1.00	1.49	1.09	-	-
String	0.21	0.34	0.22	0.31	1.54	0.68	0.92	0.87

confusion, we do not apply any further compression after default encoding techniques are applied (we will explore how each format behaves when compression is enabled in the following sections).

As we can see in Table 5, overall, Parquet performs the best over the whole dataset and is able to reduce the size of the column data to about 13% of the original. ORC is able to compress the dataset to ~27%. By contrast, Arrow Feather—with default settings—exhibits a 7% *increase* in size compared with the raw text file. We found that this overhead is introduced by the format's metadata, which adds a four-byte length prefix to each variable binary entry (i.e., the string "abc" consumes seven bytes in total). It also pads numerical data types. On the other hand, with DICT enabled, Arrow Feather compresses string columns by 68% and the whole dataset by 52%.

For integers, ORC exhibits varying compression performance relative to Parquet. ORC achieves a better compression ratio for the CodecDB and JOB datasets (which contain a relatively higher number of distinct values), while it is worse for the BI dataset (which has a lower number of distinct values). This is because ORC applies RLE for integer columns (see Table 3), which performs better for columns with fewer distinct values, whereas Parquet applies DICT-RLE, which is slightly worse. Because of this, we observe a crossover point for the Parquet and ORC CDFs in Figure 4a.

For floats, as we can see from Figure 4b, Parquet outperforms ORC and Arrow Feather because of dictionary encoding. ORC and Arrow Feather perform similarly as they both use plain encoding. For strings, Parquet and ORC outperform Arrow. Interestingly, both Parquet and ORC fall back to plain encoding on some columns when dictionary encoding takes up larger space than plain encoding, but their dictionary encoding work differently: Parquet's plain encoding introduces a higher space cost for saving the string length values, while ORC's plain encoding uses RLE for string length values. However, Parquet's dictionary encoding is more effective than ORC because of the extra layer of RLE for the dictionary-encoded keys. That is why Parquet works better in terms of total compressed size (see Tables 5 and 6) while ORC works better in terms of the effectiveness (compression ratio < 1; see Figure 4c).

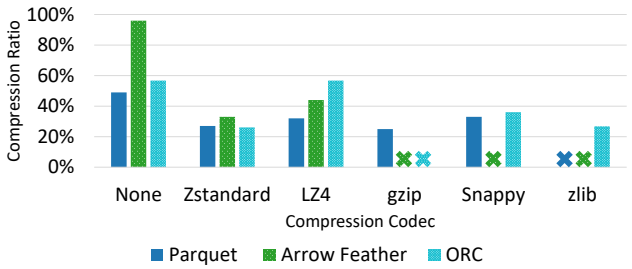


Figure 6: Compression ratio (compressed size / original CSV size) on TPC-DS (smaller is better). Uncompressed (None in the figure) only encodes using default settings. Not all formats support all compression algorithms.

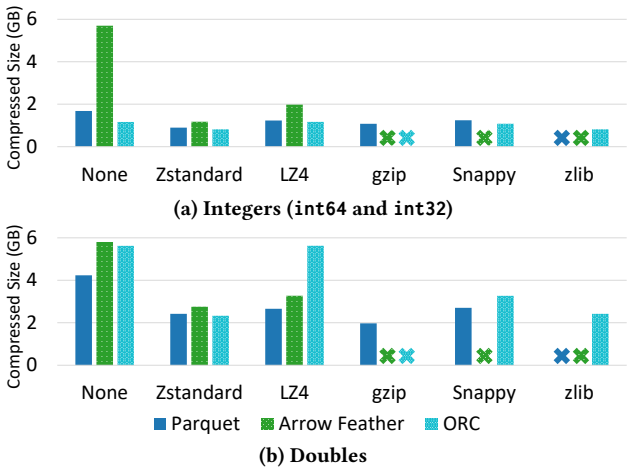


Figure 7: Total size on disk after compressing the numeric columns in TPC-DS.

5.1.2 Compression Performance. For this experiment, we report the compression ratio of each format on the full TPC-DS dataset when different compression algorithms are applied. We evaluate Zstandard (Zstd) at level 1 (we evaluate other levels later in this section), LZ4, Gzip, Snappy, and Zlib compression algorithms, and compare them against an uncompressed variant where data is encoded using the default settings. The results of this experiment are shown in Figure 6. In the uncompressed case, Parquet is about 2× better than Arrow Feather because Arrow Feather does not apply any encoding. However, when compression is enabled, Arrow performs within ~30% of Parquet. ORC achieves a similar compression ratio as Parquet, except under LZ4. In this case, ORC automatically disables compression because it detects that the LZ4 compressed data size is greater than the original data size.

Finally, we observe that different compression algorithms yield different compression ratios. For example, increasing Zstd's level from 5 to 9 yields more aggressive compression and achieves smaller sizes. However, this gain is minimal (< 1.5%) while the compression time increases by ~3× for Arrow Feather and ~2× for Parquet. We will show in Section 5.2 how decompression costs are also impacted by the choice of compression algorithm.

5.1.3 Compression Performance by Data Type. In this experiment, we look at the performance over various column types in the TPC-DS dataset. Specifically, we evaluate compression performance on

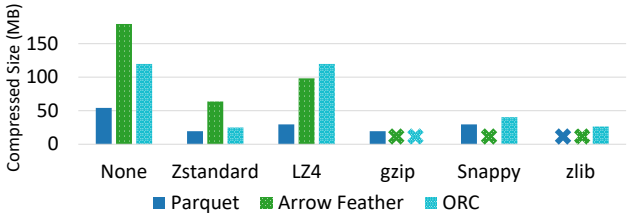


Figure 8: Total size on disk after compressing the string columns in TPC-DS.

integer (both `int32` and `int64`), double, and string data types. We extract all columns of a given data type, compress them, and report the aggregate sizes by type. The results are in Figures 7 and 8.

First, consider Figure 7a which shows the aggregate compression performance on the integer columns. ORC achieves slightly better compression performance than Parquet. This is because Parquet applies DICT and switches to plain encoding for some of the columns, whereas ORC always applies RLE. Arrow Feather does not encode by default. This leads to the worst compression ratio when compression is disabled. Nevertheless, all three data formats perform similarly when compression is enabled, except for LZ4, where Arrow Feather is almost 50% worse because it lacks encoding support for integers (we observe a similar result in Figure 6).

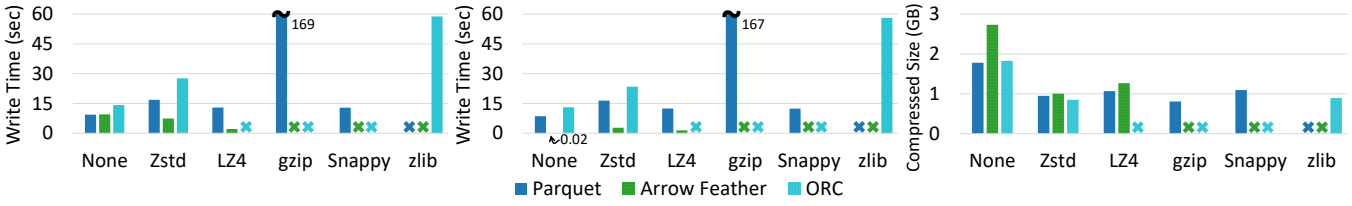
Next, Figure 7b shows the aggregate compression performance for the double columns. Parquet also applies DICT to this data type, whereas ORC and Arrow Feather do not encode at all. Because of this, Arrow and ORC have very similar performance both in the uncompressed and compressed setting, whereas Parquet is slightly better. The ORC outlier for LZ4 happens for the same reason as discussed in Section 5.1.2.

Finally, Figure 8 shows compression performance on string columns (both variable- and fixed-length). By default, Arrow does not encode this type, whereas ORC and Parquet apply DICT. Among all formats, Parquet has the best compression performance, followed by ORC and Arrow. ORC produces larger compressed sizes than Parquet, because: (i) ORC has a smaller default block size and thus pays more dictionary overhead per row batch; and (ii) it more frequently falls back to plain encoding because of its row batch-level dictionary encoding (versus the chunk-level used in Parquet). Again, LZ4 ORC disables compression because it offers no benefit.

5.2 Transcoding Overhead

In practice, storage formats are converted into (or from) an in-memory presentation on reads (writes). We now evaluate the overheads in transcoding (i.e., decompressing, converting, and compressing) each format. Specifically, Section 5.2.1 explores the time required to compress and serialize each format from a common in-memory representation, while Section 5.2.2 evaluates the overhead of loading data, i.e., deserializing and decompressing each format into an in-memory representation amenable to query execution.

5.2.1 Compression Overhead. Our first experiment in this section explores how long it takes to serialize (and compress) the data from an in-memory representation to each disk format. For this experiment we use the `catalog_sales` TPC-DS table. The `catalog_sales` is a large (~14M rows) and wide (34 columns) table containing integers and doubles. Its raw data size is 3GB. All



(a) Writing to disk **(b) Writing to null device** **(c) Compressed size on disk**

Figure 9: Write time from an Arrow in-memory table to each format stored either on disk or in memory.

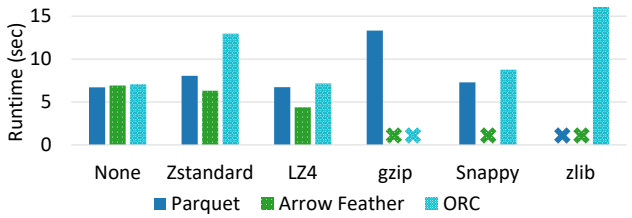


Figure 10: Runtime (in seconds) for decompressing the TPC-DS `catalog_sales` table from the on-disk formats into in-memory Arrow.

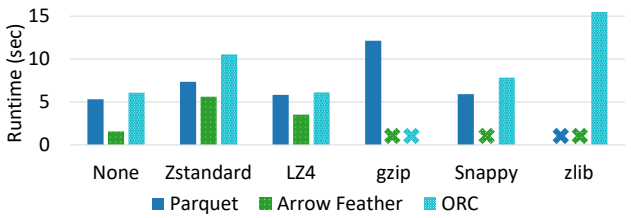


Figure 11: Runtime (in seconds) for decompressing the TPC-DS `catalog_sales` table from the formats in memory (ramdisk) into in-memory Arrow.

formats support serializing from an Arrow Table, and so we adopt it as our common in-memory representation.

Figure 9 shows the runtimes when: (i) writing to disk (9a); (ii) writing to the null device, which avoids any I/O overhead (9b); as well as (iii) the data sizes per format (9c). We omit the LZ4 and Snappy bars for ORC as the Apache Arrow C++ library has limited compression support for the ORC format. Starting with Figure 9a, we can see that Arrow Feather is the most efficient format in terms of compression and serialization runtime because it does not encode data. On the other hand, Arrow Feather's lack of encoding leads to almost a 50% larger footprint on disk (Figure 9c). Interestingly, ORC compression time is 50% slower than Parquet with comparable or slightly better compression ratio on disk (up to 15% better). We think that this is because of better Parquet support in Arrow; both projects share the same codebase and data structures.

Finally, we isolate the compression overhead in Figure 9b by avoiding disk I/O by writing to the null device. Here we can see a decrease in runtime for all the formats, although of different magnitudes. Arrow Feather has the biggest difference, thanks to its inherent zero-copy implementation in Arrow. The compression time for Parquet and ORC does not change substantially because the encoding and compression operations dominate the total runtime.

5.2.2 Decompression Overhead (i.e., table scan). In this experiment we investigate the overhead of loading the `catalog_sales` TPC-DS table from disk into memory. Our goal with this experiment is to simulate the overheads involved when a query processor is required to load and transform a compressed dataset into a plain in-memory format amenable to query execution. We start from data on disk in the Parquet, ORC, or Arrow Feather formats, and we report the time required to load the data and convert it into the Arrow in-memory format. The results are shown in Figure 10.

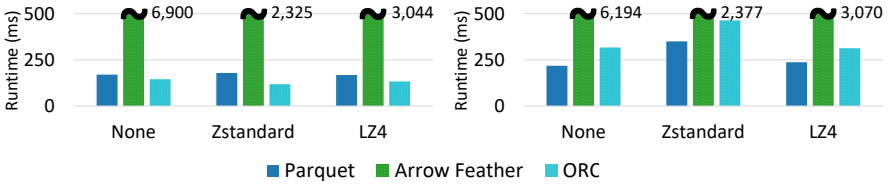
Interestingly, loading compressed data has 30% less overhead than the uncompressed case for Arrow under LZ4. This is because LZ4 requires less disk I/O (since the file on disk is smaller; see Figure 9c) while also providing "fast enough" decompression relative to the other compression methods. For the other cases, Arrow always exhibits the best performance: this is expected since it does not require decoding the data and its on-disk compressed size is reasonable. Parquet is slightly worse than Arrow because of the cost of decoding data, while ORC has the worst performance (it is particularly bad for Zstd and zlib). We think that this is due to decompression settings such as block size, buffer size, etc. In general, for formats that heavily leverage encoding (i.e., Parquet and ORC) data compression leads to a heavy penalty on read performance.

To isolate disk I/O from compression overheads, we load each compressed dataset onto a memory-resident disk mounted on `tmpfs`. As we can see from Figure 11, in all cases the runtimes decrease, especially for Arrow without compression. This result is intuitive because, for uncompressed data, the data size is much larger and disk bandwidth is saturated. Conversely, decompression is CPU-bound and not substantially impacted by the cost of bringing data into memory. Combined with previous compression experiments in Figure 9, this shows the benefits of Arrow as a fast inter-process format, when disk I/O and size are not the bottleneck.

To summarize, encoding and compression choices greatly impact performance, with formats like Parquet and ORC targeting size on disk, while Arrow targets raw read performance. To optimize both size and performance, formats should be carefully tuned to the workload and use case, and workload-aware compression selection is crucial. It remains an open question of how much computation can be pushed into the encoded space to minimize the decoding step while maximizing the compression ratio.

6 DATA ACCESS MICROBENCHMARKS

Having explored the overheads associated with encoding, compression, and scan operations, we next evaluate the performance of accessing data in the context of common relational operations found near the leaves of a query plan. Specifically, we explore the performance of projecting columns in a dataset (Section 6.1) and applying filters (Section 6.2). In this and subsequent sections we only consider Zstd and LZ4 compression since we evaluated the trade-offs of the other compression algorithms in Section 5.



(a) Integers (b) Doubles

Figure 12: Projecting numeric types on the `catalog_sales` table.

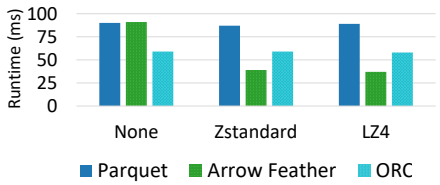
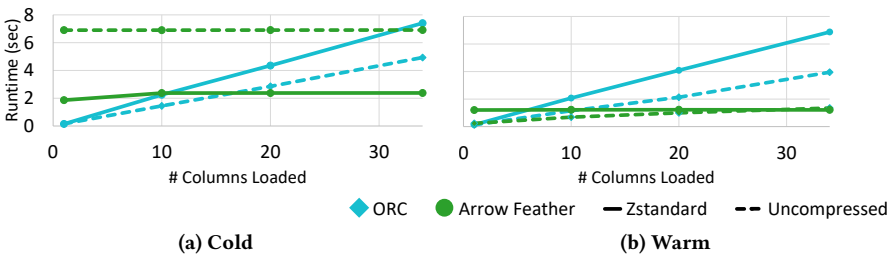


Figure 13: Projecting strings on the `customer_demographic` table.



(a) Cold (b) Warm

Figure 14: Profiling single column to full table sequential loading for ORC and Apache Arrow from the `catalog_sales` table.

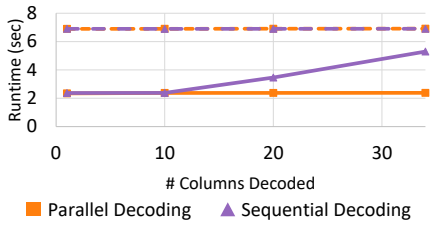


Figure 15: Arrow serial vs parallel (default).

6.1 Projection

We first explore projection performance for common data types. To do so, we: (i) load the data from disk, (ii) decompress and decode the projected columns, and (iii) convert to in-memory Arrow.

Figures 12a and 12b show the respective runtimes for projecting an integer and double column in the `catalog_sales` table of TPC-DS. Since `catalog_sales` does not contain any string columns, Figure 13 instead shows the runtime of projecting a string column drawn from the `customer_demographic` table. This table is narrower than `catalog_sales` and has 9 integer and string columns. The raw data size is 80 MB with ~2 million rows.

ORC is the most performant format for projections over integer columns because it applies RLE encoding, which yields a higher compression ratio and lower I/O costs. Parquet is slightly slower as it leverages DICT, which (i) inflates the compressed file size due to dictionary storage overhead; and (ii) slows down the loading process by introducing dictionary lookup overhead. Arrow Feather is by far the worst in this experiment since its API requires parsing the entire byte-array including all columns (not just the projected subset) from disk before column chunks loading can commence (we further discuss this in Section 6.1.1).

For doubles, Parquet offers the best performance, as we can see from Figure 12b. In this case Parquet's DICT encoding leads to a much smaller compressed size in this dataset (which has low cardinality; see Section 2.2). ORC does not encode doubles and is thus slightly slower than Parquet. Arrow Feather lags far behind for the same reason it did with integer columns.

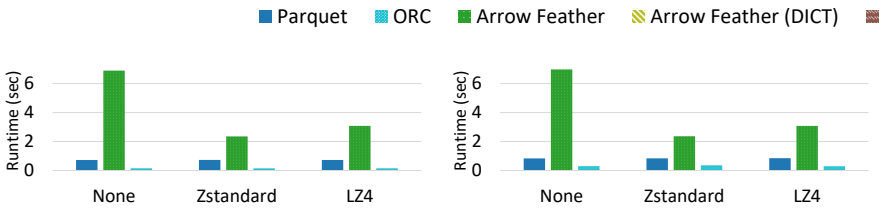
Figure 13 shows the results of projecting a string column.[1] Despite loading all columns into memory before projecting, Arrow Feather outperforms the other formats in this experiment. This is because by default Arrow Feather does not dictionary encode its data and is therefore able to entirely avoid the associated lookup overhead. ORC performs slightly worse than Arrow Feather because ORC separately RLE-encodes each string's lengths, introducing additional decoding overhead. Parquet, however, performs the worst because its application of DICT encoding inflates projection time relative to the modest I/O cost of loading the `customer_demographic` table. Finally, we want to point out that ORC is also faster than Parquet because of its API, which allows for efficiently transforming data into its dedicated in-memory representation, whereas Parquet deserializes data into memory using its relatively slow streaming style API with rudimentary data access control.

[1]The runtimes between the numeric and string experiments are not directly comparable since `catalog_sales` has an order of magnitude more records (and ~5× average row size in bytes) than does `customer_demographic`. Nevertheless, `customer_demographic` is the largest table in TPC-DS that contains a string column.

6.1.1 Profiling Data Loading. Our previous experiments show that Arrow Feather loading is far more expensive than ORC and Parquet. We observed that Arrow Feather, even when projecting a single column, requires parsing the entire byte-array. To better understand these trade-offs, we now explore Arrow's data loading code in deeper depth and contrast it with ORC.

We begin by evaluating the cost of loading a single column against the cost of loading the whole table. The results are shown in Figure 14. Overall, ORC performs best when extracting a single column. Relative to Arrow Feather, this occurs for several reasons. First, ORC offers the ability to perform fine-grained reads at the column level while Arrow Feather requires reading, decompressing and decoding the entire row batch before projecting the target column(s). This means that ORC's runtime is proportional to the number of columns extracted, while extracting one single column from an Arrow Feather file is only 2× faster than extracting the full table. Second, upon examining the Arrow Feather deserialization logic, we observed that it suffers from substantial synchronization overhead when parsing column chunks within each row batch. In particular, we found that the lock acquisition step consumed ~80% of the runtime for each row batch.

To better understand synchronization issues, we finally evaluate Arrow Feather table loading using its native data loading API in both sequential and parallel execution modes. Parallel loading mode leverages a global CPU thread pool to parallelize column



(a) Integers (b) Doubles

Figure 16: Filtering numeric types on the `catalog_sales` table.

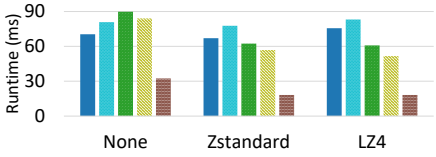


Figure 17: Filtering strings on the `customer_demographic` table.

decompression. This potentially results in a performance advantage relative to ORC, which serially decompresses columns. We see this improvement in Figure 15 where there is a large difference between the two modes when compression is enabled (the Zstd lines in Figure 15) and no difference when compression is disabled.

6.2 Filtering

In this section we evaluate each format's performance when applying filter operations. We separately consider predicate evaluation (Sections 6.2.1 and 6.2.2 respectively for numeric and string columns) and bit-vector evaluation (Section 6.2.3).

6.2.1 Numeric predicates. We evaluate two predicates over the `customer_sale` table (with respective selectivities of 65% and 30%):

CS_SHIP_DATE_SK > n (integer column filter)
CS_WHOLESALE_COST > n (double column filter)

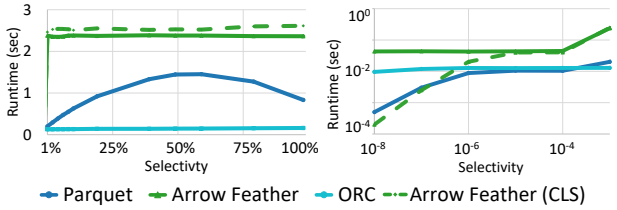
For each format, we load the data from disk, decode the target column to its in-memory representation when available (see Section 3), and evaluate the predicate to generate a bit-vector x (i.e., entry $x_i = \top$ when row i matches the predicate). For Parquet, which does not have a dedicated in-memory representation, we use its native API and interleave decompression with predicate evaluation.

The results for each predicate are shown respectively in Figures 16a and 16b. The trends are similar: overall, ORC outperforms both Parquet and Arrow Feather for each data type and compression scheme. Arrow Feather's performance is 3–4× worse than Parquet in the compressed case, but when uncompressed it further lags (to more than 7×) because the file is 2× larger than Parquet. This is due to the same reasons described in Section 6.1.1. Across all formats and for all expressions, we found that the majority of the time (i.e., >90%) is spent on data loading and decoding, whereas the contribution of the execution of the filter condition is minimal.

6.2.2 String predicates. We next evaluate a predicate on a string column in the `customer_demographic` table with 14% selectivity:

CD_EDUCATION_STATUS = n (string column filter)

The results are shown in Figure 17. Parquet is faster than ORC while Arrow with plain string encoding ("Arrow Feather") is slower in the uncompressed case, but faster than both when compression is enabled. This is because the `customer_demographic` table is small, implying that I/O is not a bottleneck for this experiment. As a result decompression dominates overall cost and Arrow Feather outperforms because it avoids the cost of decoding. String filtering on Parquet is faster than on ORC. In fact, ORC's bulk loading data access interface requires more string copying than Parquet because it materializes all strings into memory before filtering. Conversely, Parquet's streaming-style data access API does not require keeping



(a) Performance by selectivity (b) Low selectivity values

Figure 18: Bit-vector application performance by selectivity.

Table 7: Evaluated TPC-DS SP query subexpressions.

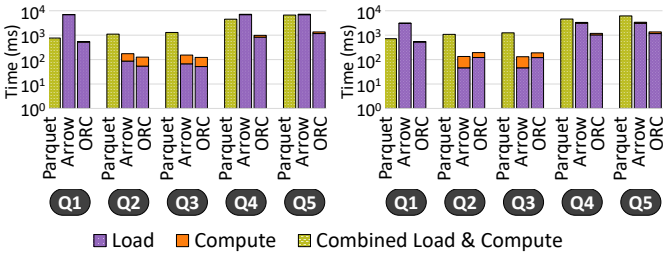
Q1	SELECT cs_ship_date_sk, cs_bill_customer_sk FROM catalog_sales WHERE cs_sold_time_sk=12032 AND cs_sold_date_sk=2452653
Q2	SELECT cd_demo_sk, cd_dep_college_count FROM customer_demographics WHERE cd_gender='F' AND cd_education_status = 'Secondary'
Q3	SELECT cd_demo_sk FROM customer_demographics WHERE cd_gender = 'M' AND cd_marital_status = 'D' AND cd_education_status = 'College'
Q4	SELECT cs_ext_sales_price, cs_sold_date_sk, cs_item_sk FROM catalog_sales WHERE cs_wholesale_cost>80.0 AND cs_ext_tax < 500.0
Q5	SELECT cs_ext_sales_price, cs_sold_date_sk, cs_item_sk, cs_net_paid_inc_tax, cs_net_paid_inc_ship_tax, cs_net_profit FROM catalog_sales WHERE cs_wholesale_cost > 80

all the strings in memory while filtering. Enabling Arrow DICT encoding ("Arrow Feather (DICT)") marginally improves performance because Arrow decodes everything when loading.[2]

6.2.3 Bit-vector evaluation. The previous sections produced a bit vector mask that indicates which entries match a given predicate. In this section we look into the performance of applying these bitmaps to produce a result. We start with Zstd-compressed data on disk and, for each format, load a column $C = \langle c_1, ..., c_n \rangle$ into an in-memory representation. We then mask C using a randomly-generated bit vector $B = \langle b_1, ..., b_n \rangle$ to produce a result $R = \langle c_i \mid b_i = 1 \rangle$.

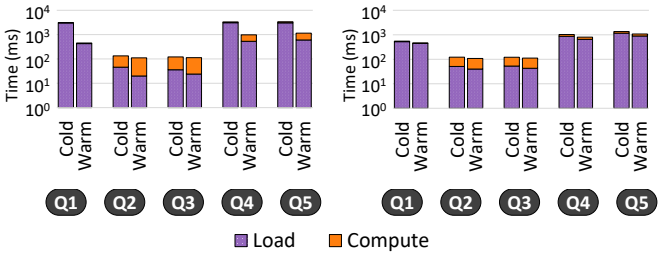
Figure 18a shows performance for each format at various selectivity levels (i.e., at selectivity s, $\sum b_i = s \cdot n$) on the CS_SOLD_TIME_SK integer column of the `catalog_sales` table. We can see Arrow Feather and ORC runtimes are approximately constant across all selectivity levels (though ORC is far faster). This is because both formats load all data into their in-memory structure before extracting the target records. Conversely, instead of fully loading all data, Parquet "pushes down" the operation by decoding only the target records that pass the filter condition. Because of this, Parquet runtimes vary for different selectivity levels. However, Parquet runtime is not a simple linear function of selectivity level. Instead, we observe the highest runtime for this format at ~0.5 selectivity, the point at which the largest number of branch mispredictions occur.

[2]We discuss the "Arrow Feather (Direct)" bar in Section 8.1.1.



(a) Uncompressed (b) LZ4

Figure 19: Log-scale runtimes by format for Table 7 queries with a cold cache.



(a) Arrow Feather (b) ORC

Figure 20: Table 7 runtimes for cold and warm caches on LZ4 compressed data (Parquet not shown; changes were negligible).

Overall, at all but the lowest selectivities ORC performs best. However, if we "zoom into" Figure 18a at the very lowest selectivities (i.e., we approach point selection) a different pattern emerges. As shown in Figure 18b, at extremely low selectivity levels (i.e., ≤ 0.001), Parquet performs better because it supports fine-grained record level data skipping. Conversely, ORC becomes better than Parquet at slightly higher selectivity (~0.01) since ORC provides a dedicated in-memory representation that efficiently loads batches. Specifically, ORC data loading consumes full data blocks, incurring extra overhead for queries with low selectivity where few entries are evaluated. On the other hand, this cost is amortized for queries with high selectivity where more data entries pass the predicate. We also implement an advanced Arrow variant that supports chunk level skipping (CLS), which we discuss in Section 8.1.3.

To summarize, trade-offs exist for simple data access operations, with no format being the best in every case. Data skipping is important, but it does not always help. Record-level data access APIs provide flexibility for data skipping but has reduced performance on queries with high selectivity compared to bulk loading APIs. To improve performance, on-disk formats should be more adaptive and co-designed with an in-memory representation.

7 LEAF SUBEXPRESSION EVALUATION

In this section we bring together the previous microbenchmarks and explore how select/project (SP) subexpressions found at the leaves of a query plan can be directly evaluated on each storage format. We use the standard API provided by each format. For Parquet, we use its streaming-style API to parse, decompress and decode the data entries while, interleaved, we evaluate the queries. For Arrow Feather and ORC, we load the data into their in-memory representations before applying the query.

We select five representative SP subexpressions from TPC-DS (see Table 7) representing a wide variety of use cases. They project few (Q1, Q3) and many (Q5) columns. They contain both equality (Q1–Q3) and range predicates (Q4–Q5). They contain predicates on integers (Q1), strings (Q2–Q3), and doubles (Q4–Q5). Queries have low (Q5), medium (Q2–Q4), and high (Q1) selectivities.

We execute each subexpression: (i) for each format, (ii) with and without compression (LZ4), and (iii) with and without clearing the system cache (to simulate the cases with repeated queries on the hot data, and infrequent queries on cold files, respectively).

The results are shown in Figure 19 (uncompressed vs LZ4), and Figure 20 (warm vs cold cache). Since, both Arrow and ORC provide a custom data loading interface (including parsing, decompressing, and decoding the data into their in-memory representations) before any query evaluation (see Section 3), we separately report "Load" and "Query" runtimes for each phase. Parquet pipelines data loading and computation, so we report only the total runtime for this format.

Overall, ORC performs best in terms of query performance because of its efficient in-memory mapping representation (lower loading time for large files) and more data-skipping opportunities resulting from a smaller row batch size. With the default setting, ORC and Arrow have 14,064 and 228 batches for the `catalog_sales` table, and 1,876 and 1 batches for the `customer_demographic` table, respectively. ORC's finer granularity allows it to skip more entries when no qualified item satisfies the filter condition. Conversely, Arrow loading time dominates runtime, slowing query evaluation, as we also observed in Figure 14. Parquet outperforms Arrow for Q1, Q4 and Q5 over large tables since Parquet file has a smaller size (i.e., less I/O) when loading the file. Parquet lags when compression is enabled as Arrow Feather sizes (I/O) decrease.

In Figure 20, we also see that other than Q2 and Q3 (which are both evaluated on the smaller `customer_demographic` table) both Arrow Feather and ORC are impacted by the system cache (significantly so for Arrow). This is because loading data from disk and building the required in-memory data structures is expensive (as we also discussed in Section 6.1.1).

In summary, smaller batches allow for more data skipping, but at the cost of space overhead and increased complexity. Workload-aware data partitioning can be used to balance these factors.

8 ADVANCED OPTIMIZATIONS

Given that there has been extensive discussion about pushing the limits of common open formats [7, 28, 59], the goal of this section is to evaluate the amenability of the Arrow and Parquet formats to support more advanced and emerging optimizations of execution engines. We will first discuss optimizations related to Arrow (Section 8.1) and then turn into Parquet (Section 8.2). While we focus on Arrow and Parquet because of space constraints and their relatively wide adoption, similar optimizations could also be applied to ORC.

8.1 Optimizing Arrow

We first evaluate the feasibility and effectiveness of integrating new optimizations into Arrow: namely its ability to support direct querying [6, 33] (Section 8.1.1), and data skipping (Section 8.1.3). We then

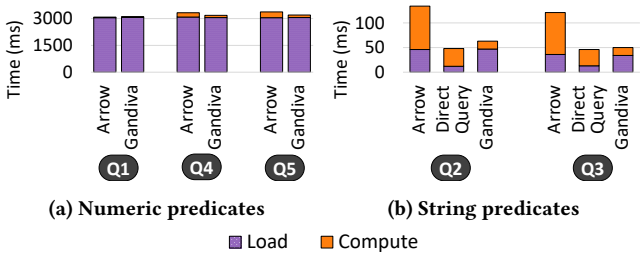


Figure 21: Arrow Feather runtime with and without direct query (warm cache execution).

compare the performance of this hand-optimized variant against Gandiva [19], a LLVM-based backend for Arrow (Section 8.1.2).

8.1.1 Direct Filtering over String Columns. We begin by exploring Arrow filtering pushdown (i.e., direct querying) into the encoded space for string columns (we separately evaluate a Parquet variant in Section 8.2.1). We modify Arrow to implement direct querying as follows. For each data chunk, we decompress and extract the dictionary from the metadata. We then map the string constant in the query predicate from the string domain into the encoded integer domain of the extracted dictionary. This process allows us to: (i) transform string comparisons into integer comparisons, which can be executed efficiently; and (ii) decode only the records admitted by the predicate.

We evaluate by repeating the experiment described in Section 6.2.2. The result is shown as the "Arrow Feather (Direct)" bar in Figure 17. Our results demonstrate a 2× to 4× improvement over other formats. The approach could be extended to range queries by employing an order-preserving dictionary (e.g., as explored in [41]).

8.1.2 Gandiva. Gandiva [19] is an LLVM-based execution backend for Apache Arrow. It is part of the Arrow project, and employs a number of optimizations (e.g., vectorization) applied via LLVM compiler passes. Gandiva further improves performance (especially for string and binary data types) by maximizing zero-copy operations.[3]

We test Gandiva by executing the queries evaluated in Section 7 (listed in Table 7). To do so, we construct expression trees for each query using `TreeExprBuilder` instances, which Gandiva transforms into machine code. As illustrated in Figure 21, Gandiva is able to slightly reduce computation time for all queries, while loading time (which is the dominant component of the queries with numeric predicates) remains unchanged. For queries with string predicates (i.e., Q2–Q3), we further compare with the optimized Arrow direct query variant described in Section 8.1.1. For these queries, the direct query variant achieves 3× speedup compared to vanilla Arrow and outperforms Gandiva as well. This is because direct query not only reduces compute time, but data loading also improves because the data decoding step is skipped.

Interestingly, Gandiva fails to generate vectorized versions for any of the evaluated queries. To test this aspect, we use Gandiva to execute a variant of Q4 that it was able to vectorize:

```
SELECT cs_ext_list_price - cs_ext_wholesale_cost -
       cs_ext_discount_amt + cs_ext_sales_price
FROM catalog_sales
```

[3]The Parquet and ORC APIs could benefit from reducing or eliminating redundant string duplication.

Gandiva's use of vectorization (and other LLVM optimizations) resulted in a 1.8× speedup (74ms vs 42ms) relative to normal Arrow.

Finally, we observe that Gandiva's compilation process is expensive relative to execution time. For the queries on the smaller dataset (i.e., Q2–Q3), the compilation time exceeded execution time (e.g., for Q2 compilation time was 103ms and runtime was 79ms).

8.1.3 Data Skipping. In Section 6.2.3 we showed how Parquet achieves excellent performance for low-selectivity filters by avoiding decode overhead for unnecessary records. We implement a similar technique for Arrow. Specifically, we augmented the bulk loading API in Arrow with the data skipping approach leveraged in Parquet. To do so, we modified the Arrow Feather API to support chunk-level skipping (CLS) where we only load the column chunks necessary to answer a given query. Because of its data layout, CLS is the most granular skipping we can employ for Arrow Feather.

To evaluate, we repeat the experiment described in Section 6.2.3, which requests a random set of row IDs. We show the result in Figure 18b as the "Arrow Feather (CLS)" series. As we see in the figure, this variant initially performs well but quickly degrades to perform similarly to unmodified Arrow. This is because the input is a bit-vector with random row IDs: even at extremely low selectivities, we quickly select at least one tuple per chunk, obviating any performance advantages.

8.2 Optimizing Parquet

In this section, we build on the lessons learned throughout the paper and augment Parquet with an efficient in-memory representation as well as vectorized instructions (Section 8.2.1). Finally, we put everything together and wrap up with an experimental evaluation comparing all optimizations (Section 8.2.2).

8.2.1 In-Memory Parquet and Vectorization. In columnar databases, it is common to encode data for better memory and bandwidth utilization. Therefore Parquet (and ORC) could potentially be leveraged as a useful *in-memory* data structure, without transcoding (to Arrow). Parquet's existing data access API either (i) fully deserializes data (precluding opportunities for fine-grained skipping or direct querying) or (ii) exposes record-level data access, which is typically much less efficient than batch loading. OLAP systems could benefit from an access pattern falling in between these two.

One example of this is CodecDB [33], which introduced a dedicated in-memory representation for Parquet. In CodecDB, Parquet data is lazily materialized (memory mapped) and fully decoded only when needed. This enables support for row batch-level, column chunk-level, and record-level skipping. Lazily-decompressed in-memory Parquet can be further optimized by leveraging vectorized instructions over in-place encoded data, as described in SBoost [31]. We implemented these optimizations based on the code as-provided by CodecDB and SBoost. We then compare the scalar performance with direct query, vectorization, and their combinations.

8.2.2 Results. We apply the above optimizations to Parquet and evaluate the performance over the queries listed in Table 7. The results are in Figure 22. As baselines, we show Parquet with its default streaming API ("Parquet"). We also show a variant in which we load Parquet into Arrow Table before query execution ("P-ArrowTable"); we included this second baseline because in Section 7 we observed

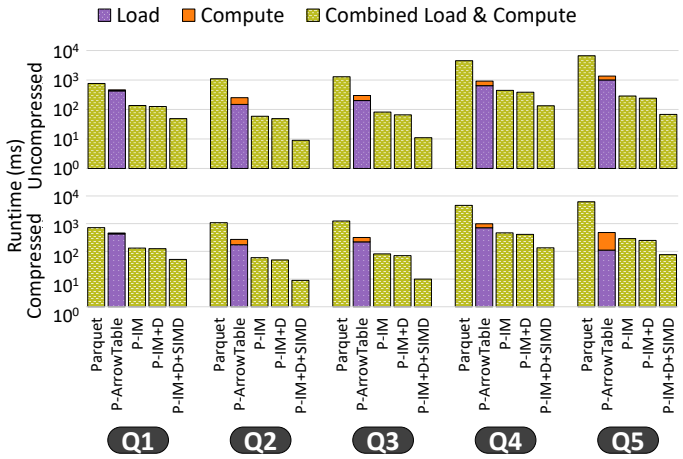


Figure 22: Parquet performance with various optimizations.

that loading data into an in-memory data structure can be quite efficient. Next, "P-IM" is the Parquet variant augmented with the in-memory format described in Section 8.2.1. "P-IM+D" further adds direct querying in the encoded domain. Finally, "P-IM+D+SIMD" enables SIMD instructions directly in the encoded domain.

As we can see from the figure, P-ArrowTable outperforms the streaming API. Nevertheless, Arrow still requires fully decoding the data into its table format, which limits the optimization opportunities. In fact, P-IM shows even better performance than P-ArrowTable, and achieves more than one order of magnitude improvement over the Parquet baseline because of its lazy materialization avoiding unnecessary decoding overhead from Parquet into Arrow. This can be further improved with direct query and avoiding decoding; P-IM+D is up to 60× faster than the Parquet baseline. Finally, if we enable SIMD execution (AVX_512) we observe up to 100× speedup compared with the Parquet baseline.

Overall, there is huge potential for query speedup when we push the query operator further down into the on-disk format and encoded domain, which demonstrates the feasibility of query push-down in the storage format when augmented with a corresponding optimized in-memory representation.

9 RELATED WORK

Columnar data format trade-offs. There has been substantial recent discussion both online (e.g., [3, 43]) and in the research community [29, 53] about the benefits and drawbacks of having multiple, often-overlapping open formats for representing columnar data. Differently than [53] our goal is not to propose a new format. Differently than [29] and other SQL-over-Hadoop evaluations [21, 48] our goal is not to evaluate the end-to-end performance of big data systems, but rather to understand how these format can be leveraged as native formats in analytical DBMSs (and how close they conform to columnar RDBMSs standards) [6, 20]. Our takeaways are consistent with the observations of Abadi [3]. Nevertheless, we provide an updated view of the trade-offs in these formats. Interestingly, we find that several limitations described in [3] persist to this day, despite their being highlighted more than six years ago.

Other encodings. In addition to the encoding methods discussed in Section 2.2, *delta encoding* is a basic encoding supported by many columnar data formats. Delta encoding works on integer data. Because differences between adjacent numbers are generally smaller than the numbers themselves, delta encoding greatly reduces data redundancy. Delta encoding works best when the numbers are large, but the value range is small. Direct querying on delta encoded data is challenging because sequential decoding is required to recover a specific record. Delta encoding variations, such as FOR and PFOR [36] use a fixed reference value instead of the previous value, which better serves the direct query on the encoded format. Even though Parquet and ORC support these delta-like encodings, the formats never elected to employ them in our experiments, presumably due to reduced performance or suboptimal encoding selection.

Other formats. Apache CarbonData [9] is an indexed columnar data format for analytics. Similar to Parquet, it uses compression and encoding to improve efficiency. Apache Avro [54] is a row-oriented storage format. It stores schema as JSON in the file header, making it an excellent choice for schema evolution tasks and write-heavy data operations such as whole-row consumption and processing. Avro also supports serialization and block level compression. We did not evaluate these formats in the paper since they are either not columnar or rarely employed by OLAP systems.

Other optimizations. The latest version of Parquet introduces the concept of an index page that contains statistics for data pages and can be used to skip pages when scanning data in ordered and unordered columns [13]. Even with the newly added index page to facilitate the access of data entries, Parquet's random-access operations are still costly in general compared with Arrow. Velox [47] is a recent effort trying to unify the execution layer across analytical engines. As also highlighted in this paper, Velox recognizes that Apache Arrow limited support for encodings is not a good fit for performant analytical engines. Velox therefore proposes improvements on top of Arrow for addressing this gap. It also highlights the need for a smart partition policy where data layout is organized for a given task (e.g., so that many data blocks can be skipped).

Other recent workload-driven data partition approaches such as Qd-tree [57], SDCs [42], Jigsaw [34] and Pixels [15] address the partition problem by focusing on data access patterns. Differently, in this paper we focus on workload-agnostic columnar formats without considering any workload-specific techniques. Workload-driven data partitioning is an exciting area (further motivated by our experimental results) and could be a promising direction for future work. The co-design of query engine and columnar formats should take workload-driven partitioning into consideration to enhance query performance (e.g., through efficient data skipping and improved compression).

10 CONCLUSION

In this paper, we evaluated three widely-used open columnar formats. We systematically evaluated them using micro-benchmarks including basic database operations and end-to-end query subexpression evaluation. We found trade-offs that make each format more or less suitable for use as an internal format in a DBMS. By applying various optimizations, we identified opportunities to more holistically co-design a unified in-memory and on-disk data representation for better query execution in modern OLAP systems.

REFERENCES

[1] Join Order Benchmark (JOB). https://github.com/gregrahn/join-order-benchmark. [Accessed: 2023].

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

[3] D. Abadi. Apache Arrow vs. Parquet and ORC: Do we really need a third Apache project for columnar data representation? dbmsmusings.blogspot.com/2017/10/apache-arrow-vs-parquet-and-orc-do-we.html. Accessed: 2022.

[4] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. 5(3):197–280, 2013.

[5] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[6] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980, 2008.

[7] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *NSDI*, pages 337–350, 2015.

[8] Apache Software Foundation. Apache Arrow. arrow.apache.org. Accessed: 2023.

[9] Apache Software Foundation. Apache CarbonData. carbondata.apache.org. Accessed: 2022.

[10] Apache Software Foundation. Apache Feather. arrow.apache.org/docs/python/feather.html. Accessed: 2022.

[11] Apache Software Foundation. Apache ORC. orc.apache.org. Accessed: 2022.

[12] Apache Software Foundation. Apache Parquet. parquet.apache.org. Accessed: 2022.

[13] Apache Software Foundation. ColumnIndex layout to support page skipping. github.com/apache/parquet-format/blob/master/PageIndex.md. Accessed: 2022.

[14] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *CIDR*, 2021.

[15] H. Bian and A. Ailamaki. Pixels: An efficient column store for cloud data lakes. In *ICDE*, pages 3078–3090, 2022.

[16] P. Boncz, T. Neumann, and V. Leis. FSST: fast random access string compression. In *VLDB*, volume 13, pages 2649–2661, 2020.

[17] P. Deutsch et al. Gzip file format specification version 4.3. Technical report, RFC 1952, May, 1996.

[18] Dremio. dremio.com. Accessed: 2022.

[19] Dremio. Gandiva. dremio.com/blog/announcing-gandiva-initiative-for-apache-arrow. Accessed: 2022.

[20] A. Ferrari and M. Russo. *The definitive guide to DAX: Business intelligence with Microsoft Excel, SQL server analysis services, and Power BI.* Microsoft Press, 2015.

[21] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full circle back to shared-nothing database architectures. *VLDB*, 7(12):1295–1306, 2014.

[22] J.-l. Gailly and M. Adler. Zlib compression library. 2004.

[23] Google. Snappy: a fast compressor/decompressor. google.github.io/snappy. Accessed: 2023.

[24] R. Gracia-Tinedo, M. Sanchez-Artigas, P. Garcia-Lopez, Y. Moatti, and F. Gluszak. Lamda-flow: Automatic pushdown of dataflow operators close to the data. In *CCGRID*, pages 112–121, 2019.

[25] G. Graefe and L. D. Shapiro. *Data compression and database performance.* University of Colorado, Boulder, Department of Computer Science, 1990.

[26] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923, 2015.

[27] B. Hentschel, M. S. Kester, and S. Idreos. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *SIGMOD*, pages 857–872, 2018.

[28] InfluxData. Querying Parquet with millisecond latency. influxdata.com/blog/querying-parquet-millisecond-latency, December 2022. Accessed: 2023.

[29] T. Ivanov and M. Pergolesi. The impact of columnar file formats on SQL-on-Hadoop engine performance: A study on ORC and parquet. *CCPE*, 32(5), 2020.

[30] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, and M. Zaharia. Analyzing and comparing lakehouse storage systems. In *CIDR*, 2023.

[31] H. Jiang and A. J. Elmore. Boosting data filtering on columnar encoding with SIMD. In *DaMoN*, pages 1–10, 2018.

[32] H. Jiang, C. Liu, Q. Jin, J. Paparrizos, and A. J. Elmore. PIDS: attribute decomposition for improved compression and query performance in columnar storage. *VLDB*, 13(6):925–938, 2020.

[33] H. Jiang, C. Liu, J. Paparrizos, A. A. Chien, J. Ma, and A. J. Elmore. Good to the last bit: Data-driven encoding with CodecDB. In *SIGMOD*, pages 843–856, 2021.

[34] D. Kang, R. Jiang, and S. Blanas. Jigsaw: A data storage and query processing engine for irregular table partitioning. In *SIGMOD*, pages 898–911, 2021.

[35] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server column store indexes. In *SIGMOD*, pages 1177–1184, 2011.

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

[37] T. Li, M. Butrovich, A. Ngom, W. S. Lim, W. McKinney, and A. Pavlo. Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats. *VLDB*, 14(4):534–546, 2020.

[38] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[39] C. Liu. Fast and effective compression for iot systems. 2022.

[40] C. Liu, H. Jiang, J. Paparrizos, and A. J. Elmore. Decomposed bounded floats for fast compression and queries. *VLDB*, 14(11):2586–2598, 2021.

[41] C. Liu, M. Umbenhower, H. Jiang, P. Subramaniam, J. Ma, and A. J. Elmore. Mostly order preserving dictionaries. In *ICDE*, pages 1214–1225, 2019.

[42] S. Madden, J. Ding, T. Kraska, S. Sudhir, D. Cohen, T. Mattson, and N. Tatbul. Self-organizing data containers. *Memory*, 1:2.

[43] W. McKinney. Some comments to Daniel Abadi's blog about Apache Arrow. wesmckinney.com/blog/arrow-columnar-abadi, November 2017. Accessed: 2022.

[44] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *VLDB*, 3(1-2):330–339, 2010.

[45] R. O. Nambiar and M. Poess. The making of TPC-DS. In *VLDB*, pages 1049–1058, 2006.

[46] J. Paparrizos, C. Liu, B. Barbarioli, J. Hwang, I. Edian, A. J. Elmore, M. J. Franklin, and S. Krishnan. VergeDB: A database for IoT analytics on edge devices. In *CIDR*, 2021.

[47] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. S. Sakka, K. Pai, W. He, and B. Chattopadhyay. Velox: Meta's unified execution engine. *VLDB*, 15(12):3372–3384, 2022.

[48] P. Pirzadeh, M. Carey, and T. Westmann. A performance study of big data analytics platforms. In *Big Data*, pages 2911–2920, 2017.

[49] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure data lake store: A hyperscale distributed file service for big data analytics. In *SIGMOD*, pages 51–63, 2017.

[50] S. A. Rodriguez, J. Chackrabroty, A. Chu, I. Jimenez, J. LeFevre, C. Maltzahn, and A. Uta. Zero-cost, Arrow-enabled data interface for Apache Spark. In *Big Data*, pages 2400–2405, 2021.

[51] M. A. Roth and S. J. Van Horn. Database compression. *SIGMOD*, 22(3):31–39, 1993.

[52] J. Shi. Column partition and permutation for run length encoding in columnar databases. In *SIGMOD*, pages 2873–2874, 2020.

[53] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler. Albis: High-performance file format for big data systems. In *USENIX*, page 615–629, 2018.

[54] D. Vohra. Apache Avro. In *Practical Hadoop Ecosystem*, pages 303–323. 2016.

[55] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang. Accelerating generalized linear models with MLWeaving: A one-size-fits-all system for any-precision learning. *VLDB*, 12(7):807–821, 2019.

[56] Y. Yang, M. Youill, M. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker. FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS. *VLDB*, 14(11):2101–2113, 2021.

[57] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-Å. Larson, D. Kossmann, and R. Acharya. Qd-tree: Learning data layouts for big data analytics. In *SIGMOD*, pages 193–208, 2020.

[58] X. Zeng, Y. Hui, J. Shen, A. Pavlo, W. McKinney, and H. Zhang. An empirical evaluation of columnar storage formats. *arXiv preprint arXiv:2304.05028*, 2023.

[59] F. Zhang, W. Wan, C. Zhang, J. Zhai, Y. Chai, H. Li, and X. Du. CompressDB: Enabling efficient compressed data direct processing for various databases. In *SIGMOD*, pages 1655–1669, 2022.