Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

A DuckDB extension for FastLanes

Author: Sebastiaan Eddy Gerritsen (2818056)

1st supervisor:	Prof. Dr. Peter Boncz
daily supervisor:	Dr. Azim Afroozeh
2nd reader:	Prof. Dr. Jacopo Urbani

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

March 31, 2026

"Talk is cheap. Show me the code."

, Linus Torvalds

Abstract

Parquet is the de facto columnar storage format in analytical data processing and is widely integrated into systems such as Apache Spark, Hadoop, Hive and many major data lake platforms. It is used across environments ranging from end-user devices to distributed clouds scaling to terabytes of data. However, Parquet's design reflects assumptions of the early 2010s. Its encoding schemes incur substantial overhead, it exhibits poor random access performance, and it does not exploit modern hardware capabilities such as SIMD or GPU acceleration.

This thesis introduces and evaluates an extension of FastLanes, a next-generation columnar format, into the analytical database DuckDB. It incorporates advances in encoding schemes, data layout, SIMD and GPU compatible code drawn from cutting edge research. This integration enables a practical assessment of the format within a real analytical engine and makes it accessible to end users.

Experimental results demonstrate that FastLanes reduces end-to-end query latency compared to Parquet and achieves competitive performance relative to other next-generation columnar formats, including DuckDB's internal storage format and Vortex. These findings illustrate the potential of FastLanes as a strong candidate for the next industry-standard columnar storage format.

Acknowledgements

I would like to extend my gratitude to Azim Afroozeh, who continuously guided me throughout this thesis and was a valuable sparring partner. I am also very thankful to Peter Boncz for offering valuable insights and direction on how to shape this work. Finally, this thesis was conducted at Centrum Wiskunde & Informatica in Amsterdam, which provided a great research environment filled with highly knowledgeable and passionate people.

Contents

List of Figures

LIST OF FIGURES

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 Context

Parquet is the current state-of-the-art columnar storage format for cloud-based big data (9). This open-source format provides cross-platform data sharing and is optimized for analytical workloads on big data. However, Parquet was developed in the early 2010s; since then, both hardware capabilities and workload characteristics have changed. In light of these developments, new formats propose to change the status quo. BtrBlocks (10) uses lightweight encodings by applying multiple lightweight encodings in a cascaded pipeline, as an alternative to heavier block-based equivalents such as Snappy (11) and LZ77-based algorithms (12) used by Parquet. Nimble (13) has light metadata organization to better suit wide workloads, such as machine learning. Vortex (14) extends cascaded encoding with parallelization, making the encodings suitable for SIMD- or GPU-based decoding. In addition, Vortex implements compression schemes for improved compression and decoding speed. Yet none of the aforementioned storage formats has emerged as a successor to the current state-of-the-art. This, in turn, raises the question of what the limiting factors for adoption are. We gather that Nimble is closely coupled with the overarching system Velox. For Vortex, we find that important features are still under development. In general, adoption seems to be a hard problem, given the numerous proposed alternatives to Parquet. This difficulty is further evidenced by the fragmented feature support found in Parquet "V2", where different implementations support different subsets of the specification, driving users to stay on an older established version. In this thesis, we propose the implementation of an extension into DuckDB (15) for a next-generation storage format FastLanes (8). FastLanes combines the aforementioned techniques of parallelized cascaded encodings and new compression schemes, informed by recent academic research. By implementing a reader and writer in DuckDB, we can evaluate its performance relative to Parquet. Furthermore,

as DuckDB is an embedded analytical database, and a significant fraction of big data workloads tend to fit on end devices (16), providing a DuckDB extension enables end users to independently verify its efficacy. The goal of this work is to implement an extension that, together with the key features of FastLanes, provides a foothold for lowering the barrier to industry adoption. These considerations lead to the following research questions:

RQ1. What is an appropriate extension architecture for integrating FastLanes into DuckDB with minimal overhead?

RQ1.1. Is the current FastLanes library API sufficient for efficient integration?

RQ1.2. What extensions or modifications are required or recommended for the FastLanes library?

RQ2. What is the read and write performance of FastLanes on OLAP workloads in DuckDB compared to Parquet, Vortex, and DuckDB's internal format?

1.2 Contributions

- **Literature Review** We provide a literature review that dives into the inner workings of the current industry standard Parquet and next-generation formats Vortex and FastLanes. We describe file layouts, encodings, and key differentiators between these formats.
- **DuckDB extension** We implement a FastLanes extension for DuckDB that adds table and copy functions for reading and writing FastLanes files.
- **Performance Analysis** We benchmark volumetric scans and TPC-H queries to gain insight into how file-format internals and extension implementations affect query performance.

1.3 Outline

We start by providing the background necessary to be able to understand the descriptions in later chapters. In Section 2.1 we give a primer on database workloads, followed by hardware considerations on modern systems in Section 2.2. We then move to common database architectures in Section 2.3, and how database systems store persisted data in Section 2.4, further expanded with descriptions on common encoding schemes in Section 2.5. We end with an overview of the internal structure of DuckDB in Section 2.6.

Chapter 3 encompasses the literature study and dives into Parquet in Section 3.1, Vortex in Section 3.2, and FastLanes in Section 3.3. It decomposes the different parts of each columnar format, including the binary layout, supported encodings, and encoding selection.

Chapter 4 introduces the extension mechanism present in DuckDB. It expands on the interface for the table function in Section 4.1 and for the copy function in Section 4.2. We conclude with Section 4.3, describing an optional supporting interface which standardizes and abstracts common functionality for the table function.

Chapter 5 describes the implementation of the FastLanes extension. In Section 5.2 we expand on the details of the reader, which integrates with the table function. In Section 5.3 we expand on the writer, which integrates with the copy function. In Section 5.4 we discuss patches that were applied to a fork of the FastLanes code in order to improve performance.

Chapter 6 details the evaluation results and accompanying analysis. The benchmarking setup is described in Section 6.1, with evaluations for the reader and writer present in Section 6.2 and Section 6.3 respectively.

Finally, we discuss limitations of the current implementation in Chapter 7, and conclude this work in Chapter 8.

2

Background

This chapter provides a concise background on the design considerations for the internal structure of a database management system, as well as design considerations for database storage formats.

2.1 Workloads

Traditionally, database management systems (DBMSs) were designed for on-line transaction processing (OLTP) applications, of which banking transactions or order entry systems are examples. These systems tend to update records often in isolated transactions, and query relatively few records compared to the entire dataset. As such, OLTP databases are designed for consistency and transaction throughput. In contrast, on-line analytical processing (OLAP) databases are designed for query throughput at the cost of transaction throughput. These workloads include ad hoc queries that may access large portions of the dataset with join or aggregate operations (17, 18).

2.2 Hardware

Processing analytical queries with low latency and high throughput on large datasets is a major challenge (19). As such, much work goes into utilizing novel hardware-level features in DBMSs. In this section, we discuss trends and considerations with regard to various hardware components: memory, CPU, and GPU.

2.2.1 Memory

For a general-purpose machine, we can identify a memory hierarchy as shown in Figure 2.1, which has multiple levels and where each level closer to the processor incurs less overhead but also has a lower available capacity (20). As databases are performance-critical software

programs, careful consideration must go into how the database affects and uses the different available layers.

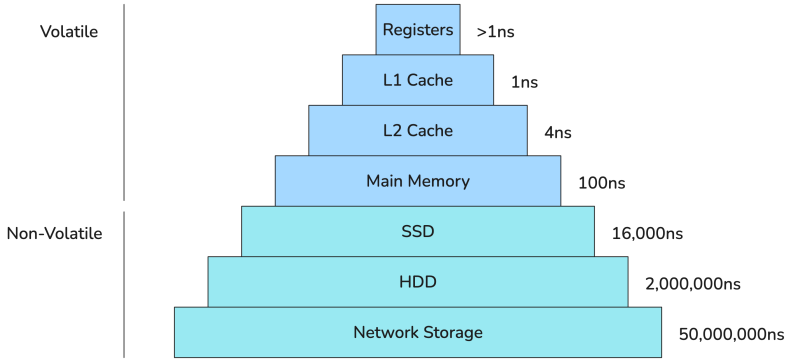


Figure 2.1: **Memory Hierarchy** - Generalized hierarchy for a general-purpose computer visualized as a pyramid of component blocks. Block widths indicate the non-proportional relative available capacity between components. Each block has an associated access time indication denoted in nanoseconds.

Historically, communication between the CPU and secondary non-volatile storage has been recognized as the major database performance bottleneck. However, due to advances in hardware and increased interest in OLAP workloads, this is no longer universally true. Boncz et al. (21) highlight the disproportionate advances between CPU and memory performance, where the speed of microprocessors has been outpacing memory speeds. Three areas of interest are identified regarding memory performance: bandwidth, latency and address translation. Latency and address translation times can be reduced using caches within the memory-hierarchy, given the relevant data is resident in the cache. The authors note that while bandwidth has enjoyed advances in capacity, latency has been almost at a standstill. They conclude that ineffective use of caches neutralizes all advances in CPU speed. Ailamaki et al. (22) further show that both OLTP and OLAP workloads experience significant bottlenecks in memory access, with memory-related CPU stalls accounting for 60-80% of query execution time. They identify the dominant components being L1 instruction and L2 data cache misses. It is of note here that L2 data thrashing can also affect the L2 instruction cache, as the L2 cache is usually a unified cache.

2.2.2 CPU

General-purpose CPUs have enjoyed improvements on internal techniques like prefetching and branch prediction. In addition, modern CPUs provide high computational capabilities through different sources of parallelism: thread-level parallelism, data-level parallelism

and instruction-level parallelism. Unlike internal techniques, database systems need to take these capabilities into account during their design and implementation to use them effectively.

MIMD Multiple Instruction, Multiple Data (MIMD) is offered by CPUs that have multiple independent and asynchronous processing elements (PE), or cores. Also called thread-level parallelism, it allows a CPU to execute different instructions on different pieces of data at any time. For DBMSs, MIMD is often realized by logically partitioning data objects, such as columns, which are then divided over multiple threads pinned to a PE. In this model the same operator in a query pipeline is executed simultaneously over disjoint data partitions (19). We cover how DuckDB deals with multiple PEs in Section 2.6.

SIMD Single Instruction, Multiple Data (SIMD) describes multiple PEs, or SIMD register lanes, that perform the same instruction on multiple different data elements in parallel. Modern CPUs have specialized circuits that allow SIMD operations to be performed in fewer cycles compared to their scalar counterpart. A drawback is that different platforms have different implementations and operation support for SIMD. These include but are not limited to Intel SSE, AVX512 and ARM Neon. Using SIMD requires explicit intrinsics for otherwise simple operations like addition and subtraction (19). This requires different implementations across platforms if a DBMS wants to be performant as well as enjoy cross-platform support.

2.2.3 GPU

As the growth of CPU performance has slowed in recent years, graphics processing units (GPUs) might offer an opportunity for DBMSs and data formats alike. Since 2008, GPU core counts have increased by x45 and memory bandwidth has increased by x23. While this high bandwidth and massively parallel architecture fits well with OLAP workloads, there are important challenges that have to be taken into account to effectively use GPUs within a DBMS. Paul et al. cite three reasons: First, GPUs differ significantly in hardware architecture compared to CPUs, requiring redesigns from the ground up to accommodate both architectures. Second, GPU hardware has been evolving rapidly; combined with a lack of studies on the impact of certain features, this makes effective system design hard. Third, GPUs have relatively small amounts of memory available, necessitating expensive remote access (23).

2.3 Processing Models

When a user submits an SQL statement, the DBMS converts this into a query plan, which consists of operators structured as a tree. Leaves produce data, propagating the results to their parents which propagate up to the root, where the root represents the answer to the query. An example query plan is given in Figure 2.2 for the SQL statement in Listing 1. The processing model defines how a DBMS executes a given query plan. This includes decisions on who drives the production of tuples and what kind of data is passed to operators (24). Different models have various trade-offs for different workloads. We consider three popular models in the remainder of this section.

```
SELECT E.name, D.dept_name
FROM E JOIN D ON E.dept_id = D.id
WHERE E.salary > 50000
```

Listing 1: An SQL statement, where E is an employee table and D is a department table.

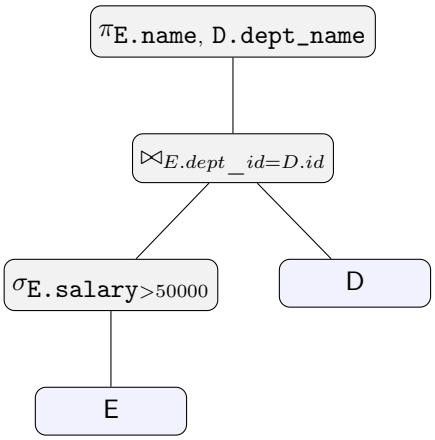


Figure 2.2: A query plan representing the SQL statement from Listing 1.

2.3.1 Iterator Model

The iterator model, also known as the Volcano model, is the most common model and used predominantly by OLTP DBMSs. This approach can also be described as a top-down or pull-based model. The root node kicks off execution by requesting the next tuple from its children with a `.next()` call. These children will then call their children with `.next()`, up to the leaves. Each invocation returns a single tuple, or `null` in case there are no more tuples left to process. When the leaves are reached, tuples are emitted and propagated through the operator nodes as far as possible back up to the root. This creates

a pipeline of operators, which allows for the computation of a query result with minimal intermediate state, reducing memory pressure. Some operators, such as a hash join or sort, require all their children to return a tuple before the operator itself can return one. These operators are called pipeline breakers, and reduce the efficacy of a pipeline model by increasing memory pressure and introducing synchronization points.

The iterator model has more performance implications; as operators in a plan run tightly interleaved, the set of instructions may become too large to fit into the cache. In addition, operators have their own state, the combined state of all operators in a query plan might not fit into the cache either. Besides cache performance, the overhead per tuple can be significant due to the function call overhead required for each operator (24, 25, 26, 27).

2.3.2 Materialization Model

The materialization model uses a push-based, or bottom-up approach. In contrast to the iterator model, where parents dictate the pace of a query plan execution, the materialization model is driven by its producers. Instead of producing intermediate results one tuple at a time, whenever an operator is reached it generates the entire result required by nodes higher up in the tree. This design also means that each operator in the tree is called only once.

As each operator is called only once, it can more easily amortize the overhead of operator setup and function calling with respect to the processed data. This results in a further improvement in instruction and data cache locality. However, this model is best suited for OLTP workloads. With OLAP workloads the state may become too large to fit in memory, spilling to disk and incurring significantly higher latency (24, 25, 26, 27).

2.3.3 Vectorization Model

Inspired by the iterator model, the vectorization model follows a top-down, pull-based model using `.next()` function calls. However, instead of doing tuple-at-a-time pipelining, multiple tuples are returned or processed. These tuples are organized in batches, or vectors, and the operators are optimized to process multiple tuples at a time.

Using batches instead of single tuples allows for a reduction of function calls within the tree, benefiting OLAP workloads that often scan large amounts of tuples. Batches also allow for easier use of SIMD intrinsics, better exploiting modern CPU capabilities. Special care has to be taken when constructing query plans however, filtering operators may reduce the effectiveness of batching, since rows within a batch can be invalidated. If

filtering is highly selective, performance may degrade close to tuple-at-a-time execution (24, 25, 26, 27).

DuckDB started out with a variant of the vectorization model, having `GetChunk` calls operating in a similar fashion as described earlier. However, due to difficulty in parallelizing this model, DuckDB now employs morsels, as outlined in Section 2.6 (28).

2.4 Data Layouts

Data Layouts are used when persisting data to disk. As both OLAP and OLTP have differing query patterns, database design optimizes for these patterns by storing data in distinct layouts. This section discusses well-known approaches and their derivatives, concluding with a brief on columnar formats.

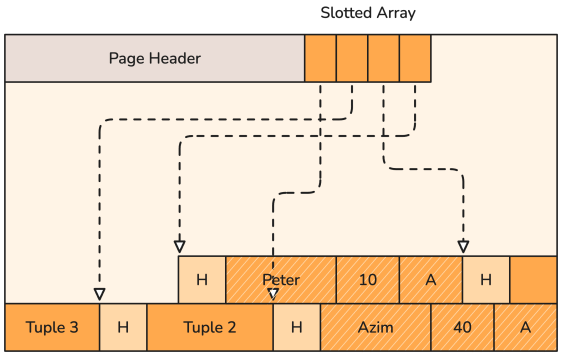
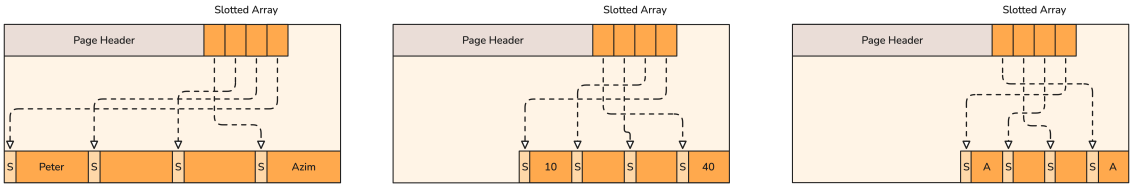


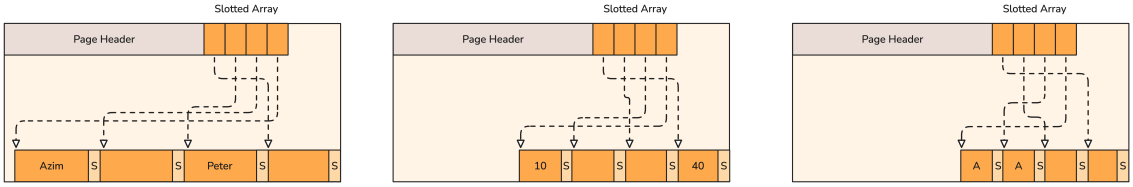
Figure 2.3: **Slotted Page** - The internal layout of a slotted page, at the start of the page is a header. Following is the slotted array which contains pointers to the start of the records contained in the slotted page. Records consist of a record header and the tuples with their respective values.

2.4.1 NSM

The N-ary Storage Model, also known as slotted pages, stores records in sequential pages on disk. The size of a page is a multiple of 4`KB`. This size is consistent across a table or the entire database, but the default value can differ between databases. While the exact internal page structure may differ between implementations, most follow a layout as showcased in 2.3. The page starts with a header, containing metadata like the available space, a checksum and information for identification of the page. This is followed by a slotted array, containing offsets and sizes of each record stored in the page. Each offset points to the start of a record, which consists of a header and the tuple data. The header may contain a bitmask to indicate `NULL` values. The tuple data is the contiguous and



(a) DSM layout with records split into subrelations with clustering on the surrogate.



(b) DSM layout with records split into subrelations with clustering on the attribute values.

Figure 2.4: **DSM Layout** – The internal page layouts for both clustering approaches. Each page follows a structure similar to an NSM page. Offsets in the slotted array point to an attribute and surrogate pair in the page, representing a sub-relation of the record.

sequential stored column elements for a record. The slotted array and the tuples start at separate ends of the page, and grow inwards as more records are inserted. The advantage of this layout is easy insertion of variable width records, as well as independent compaction for each page in case of record deletion. Furthermore, this design loads multiple values of a record into cache if the value block size is smaller than the cache block size, thus favoring OLTP access patterns over OLAP access patterns. In case the record size is larger than the cache block size, a predicate evaluation would result in a cache miss per record (1, 29, 30).

2.4.2 DSM

In an effort to deepen the analysis of the characteristics of storage models across different dimensions, Copeland et al. (31) propose a fully Decomposed Storage Model (DSM). This is constructed as a transposed storage model where each attribute carries the record identity, also called the surrogate, separately. DSM saves two copies of a record. One copy is clustered around the surrogate and the other around the attribute value. The resulting page layouts are shown in 2.4. Each copy stores attributes of a record in separate pages, so-called sub-relations. Sub-relations are then (partially) reconstructed on demand using a join operation, depending on the attributes required for the resulting relation. In contrast to NSM, DSM does not require the use of inverted files or other indexing techniques to optimize record retrieval, as both ways of the binary relation of the attributes are stored in sub-relations, making indexes part of the storage format itself. This removes the need for

manual tuning of the indexes which would be required with NSM formats. Furthermore, when utilizing a small fraction of the attributes in a relation during a sequential scan, DSM exhibits higher cache and I/O performance compared to NSM due to higher spatial locality (29, 31).

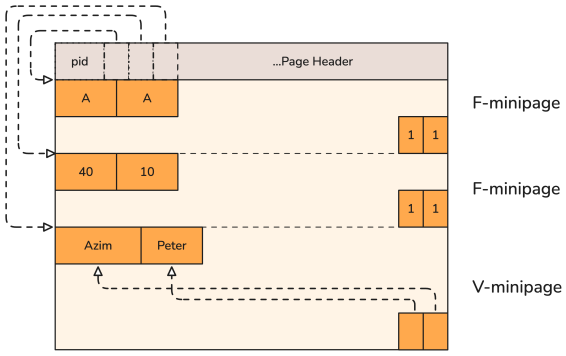


Figure 2.5: **Pax Page** - The internal layout of a PAX page, at the start of the page is a header, this contains a page id and offsets to the minipages contained in the page, among other metadata. Following are three minipages, equal to the amount of attributes for a record in the relation. The first two minipages contain fixed-width values, formatted as F-minipages. The third minipage contains variable-width values, formatted as a V-minipage.

2.4.3 PAX

While DSM improves on cache performance, the format suffers from performance degradation when many attributes are involved in the resulting relation. This is caused by the reconstruction cost when joining the required sub-relations into a record that can be used to derive the relation result. Considering this limitation of DSM, and the performance characteristics of NSM, Ailamaki et al. (29) propose a storage layout called Partition Attributes Across (PAX). PAX partitions each page into minipages equal to the degree of the relation stored. A header at the beginning of the page contains offsets to the start of each minipage, along with additional metadata on how to parse the data contained in each minipage. A minipage can be either of two types; F-minipages are used for fixed-width attributes. Values are inserted at the start of the minipage, with a presence bit vector at the end to indicate `NULL` values. V-minipages are used for variable-width attributes. Values are inserted at the start of the page, with a vector of offsets at the end of the minipage, pointing to the end of each value. This design gives PAX improved spatial locality over NSM, while not requiring expensive join operations as in DSM due to all attributes being available on the same page. The authors report a 75% reduction in L2 cache stall time for data fetches, which further reduces L2 cache misses on instructions as the cache is no

longer thrashed with unreferenced data. This translates into an 11–48% speedup when running TPC-H on non-memory resident data (29).

2.4.4 Columnar Formats

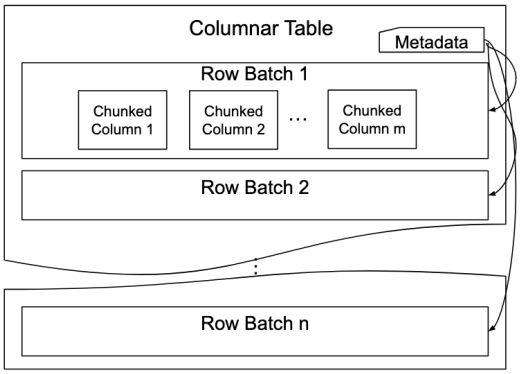


Figure 2.6: General columnar format layout. Image from Liu et al. (1).

The aforementioned data layouts are mostly used within the context of a format internal to the accompanying database. However, driven by the increasing need for systems capable of handling big data analytics, columnar formats have been externalized to accommodate these systems (32). One of the first instances of such a format is RCFile (33), designed to interoperate within the Hadoop ecosystem using the distributed file system HDFS. Inspired by PAX, RCFile applies the concept of "first horizontally partition, then vertically partition". But instead of doing this on page granularity, the entirety of the table is considered. Tables are horizontally partitioned into equally sized row groups, in which each column is stored one after the other, with all values of a column being stored contiguously. Each column is compressed using the general-purpose compression algorithm Gzip. The major benefit of RCFile is that it can skip over columns not required to resolve a given query, saving on I/O and bandwidth. After two years, Meta announced a refinement of RCFile named ORC (34). ORC followed a similar layout but improved on RCFile by retaining type information of the columns, whereas RCFile would treat all data as an opaque blob of bytes. ORC uses the type information to apply lightweight encodings for improved compression, after which it can optionally compress further using general-purpose compression (35). Another addition to the ORC format was the inclusion of minimum and maximum metadata at each row group and file, allowing the reader to skip entire sets of rows using pushdown filters. Shortly after ORC was released, Twitter and Cloudera announced Parquet, which too follows a similar layout to RCFile and ORC. A

generalization of this layout is given in Figure 2.6. Currently ORC and Parquet are the de facto industry standards. They enjoy wide compatibility with data processing platforms.

2.5 Encodings

Studies have found that compression in database systems can improve performance by reducing the data size, in turn reducing the pressure on memory bandwidth and I/O (36). With the advent of column-oriented databases new possibilities arise for the application of compression algorithms. Whereas with row-stores compression is not trivial, with the most notable approach being the use of a dictionary within a page. Columnar layouts, in contrast, store attributes from the same domain together. This makes it easier to exploit the properties of the domain, as values close together are highly related, allowing further compression across rows. In addition to improved compression, the overhead of iterating through values of a column-store tends to be lower due to the use of vectorized code. A further optimization possible with column-oriented compression is to operate directly on the compressed data by database operators.

In the remainder of this section, we will go over the encoding algorithms used by the storage formats discussed in the remainder of the thesis. We start with a short introduction to general purpose compression, which is followed by a case-by-case study of LWCs.

2.5.1 General Purpose Compression

General purpose compression (GPC) schemes operate over a bit or byte stream of data, making them type-agnostic and thus applicable to a wide range of data sources. Algorithms used in columnar formats include Snappy (11) and Zstd (37), both of which are based on Lempel-Ziv compression (38), which is the most widely used technique for lossless file compression. Specifically, Snappy and Zstd are based on a Lempel-Ziv variant known as LZ77. The differing factor is that Zstd combines LZ77 with entropy coding for increased compression, whereas Snappy does not in return for speed. As there are many variations of Lempel-Ziv encodings, we build an intuition for these encodings by illustrating LZ77. The main idea behind LZ77 is to have two contiguous sliding windows over the input stream, the search buffer and the look-ahead buffer. The encoder then finds the longest sequence of symbols in the search buffer that matches the symbols in the look-ahead buffer. Based on this match, the encoder outputs a tuple containing the length and offset (pointer) which locates the match, and a symbol which corresponds to the first symbol in the look-ahead buffer that is not part of the match. After finding a match, the window, consisting of the

search and look-ahead buffer, is shifted as many symbols as the length of the match plus 1. When the look-ahead buffer becomes empty the input stream has been fully processed (12). An example is given in Figure 2.7.

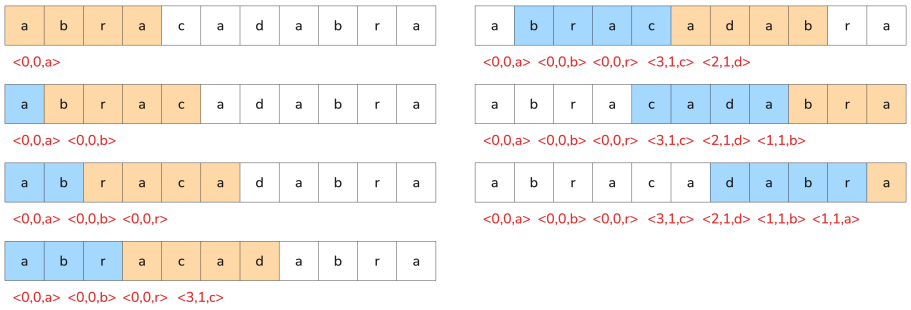


Figure 2.7: Example of LZ77 compression over the string 'abracadabra'. Blue blocks on the input stream denote the search buffer, orange blocks denote the look-ahead buffer. The red symbols denote the output in the form <offset, length, next symbol> after each step in the algorithm.

From the example can be gathered that the string 'abracadabra' could be encoded more efficiently by using a larger search buffer, when using a search buffer of size 7 the repetition 'abra' can be encoded as <7, 4, EOF>. This highlights the tradeoff between compression ratio and performance, a larger search buffer offers greater chance of finding longer matches but also increases the search space. As a consequence of how LZ77 (and similar) algorithms operate, decoding the entire block is required as the encoded output uses back-references that may lead (recursively) to the first encoded symbol of the block. This implies that GPC schemes are ill-suited for random access. In addition, GPC schemes do not support execution over compressed data, in contrast to lightweight encodings where certain schemes do allow for compressed execution (36, 39). Zeng et al. find that the decompression overhead of GPC schemes tends to dominate the I/O and storage savings, as storage media gets faster and cheaper over time (32). Based on these findings, and the aforementioned characteristics, this work does not explore the notion of GPC further.

2.5.2 Lightweight Compression

Constant Encoding A straightforward encoding is to collapse all values into a single value when all values are the same. An example is given in Figure 2.8. As the encoding does not store a value count, it requires the layout implementing constant encoding to have a predefined number of elements per compression block (40).

Figure 2.8: A simplified example for compression with constant encoding.

Bit Packing When integer values do not use up the entire range of their representation, the required space for storage can be reduced through bit packing. The main idea is to determine the required bit width of the largest value in the set, and subsequently "cutting off" the unused bits by placing the bit width bits of each integer after each other. The bit width is equal to $\lceil \log_2(M+1) \rceil$, with M being the maximum value in decimal notation. An example is given in Figure 2.9, where four 8-bit integers with a bit width of $\lceil \log_2(3+1) \rceil = 2$ are bit packed into a single 8-bit integer. Of note is that outliers in a block, even a single occurrence, reduce the compressibility for the entire block as the bit width follows the bit width of the largest value in the block. To reduce the effect of outliers, a block can be split into multiple smaller sets (chunks), with each chunk having its own bit width (40).

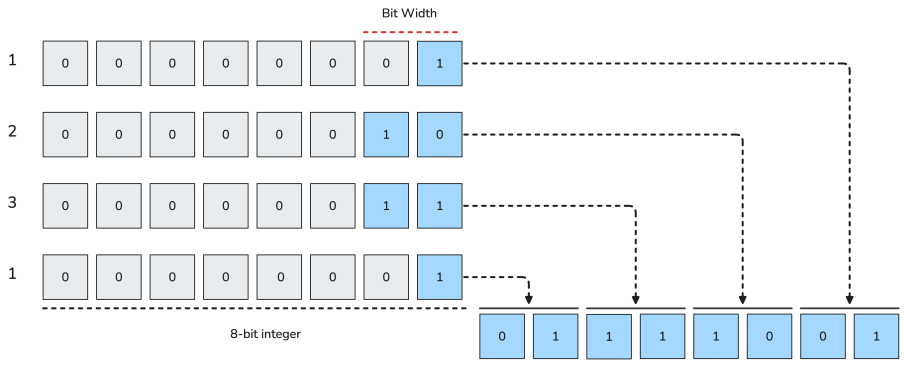


Figure 2.9: A simplified example for bit packing the set of integers 1,2,3,1, represented as 8-bit integers with bit width 2, into a single 8-bit integer.

Run-Length Encoding Run-length encoding (RLE) collapses consecutive occurrences of the same value into a single value representation. Usually stored as a tuple comprising the value and the running length, where each element in the tuple is a fixed number of bits. An example is given in Figure 2.10, where 2 runs totaling 5 values are compressed into 2 tuples. As RLE uses consecutive values, sorted columns and columns with a low amount of distinct values are well suited for this compression method (36, 40, 41). A variant of RLE is Run-End encoding (REE) which stores values as two arrays, one containing the values and the other containing the logical index where the run for the respective value

ends. The benefit is that REE allows for random access, where for RLE it would require decoding from the start of the block. A disadvantage is that REE stores absolute index values, which tend to be larger compared to run lengths.

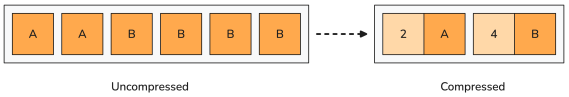


Figure 2.10: A simplified example for compression with RLE encoding.

Delta Encoding Delta encoding tries to reduce the amount of bytes required to represent values by storing the difference (delta) between consecutive values instead. Each compression block maintains a base value, which is an uncompressed value at the start of the block, then each subsequent value is stored as the difference from the preceding value. Delta encoding is well suited when a domain contains large values, where a significant part of a value contains redundant information. This can occur with time data, such as dates and timestamps. An example is given in Figure 2.11, after applying delta encoding the bit width of the values has been reduced, offering an opportunity for further compression using bit packing. A drawback of Delta encoding is that it increases data dependencies between values. In the worst case one needs to fetch all tuples when interested in the last encoded value in the compressed block (42).

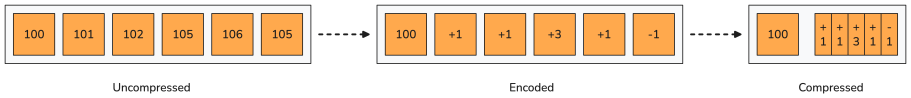


Figure 2.11: A simplified example of delta encoding, followed by bit packing.

Frame of Reference Similar to Delta encoding, Frame of Reference (FOR) can be of benefit when a domain has large values with a redundant factor. Unlike Delta encoding, however, FOR exploits clustered data by representing values from a block as a constant base plus an offset. This makes FOR better suited than delta encoding in case where the range of values is relatively small to a common base, but where deltas between consecutive values are large. An example is given in Figure 2.12, showcasing the result of encoding the same source block as in Figure 2.11 with FOR (41). In contrast to delta encoding, FOR does not introduce dependencies between values, making it generally better suited for random access (43).

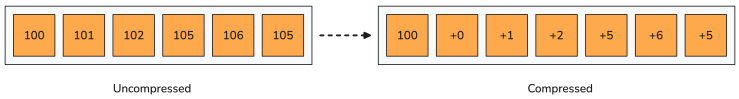


Figure 2.12: A simplified example of FOR compression.

Dictionary Encoding Dictionary encoding replaces values in a block with keys that require less space, often integers, where the keys map to the original values in a separate segment. Dictionary encoding is thus most effective when the number of distinct values is low, so that multiple key replacements will point to the same value in the dictionary. An example is given in Figure 2.13, where string values are replaced by integer keys mapping to values in a dictionary (43).

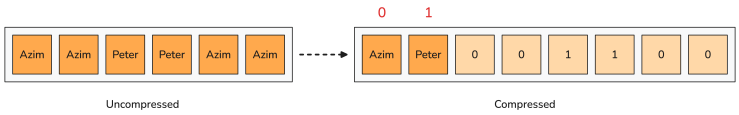


Figure 2.13: A simplified example of dictionary compression.

Fast Static Symbol Table While dictionary encoding is effective in distributions where a large fraction of string values are equal, it is not able to effectively compress strings that are similar but not fully repeating. In contrast, Fast Static Symbol Table (FSST) is able to compress strings that share common sub-strings. FSST builds a symbol table containing sub-strings leading to the highest compression factor of the block, where symbols range in length from 1 to 8 bytes. Each symbol is associated with a 1-byte code. This code is then used to replace the associated sub-string across the values in the block. As a code must be contained in a byte there is a maximum of 255 symbols that can be stored in the table, with the code 255 reserved for the escape sequence. The escape sequence is used as prefix for sub-strings which are not stored in the symbol table, but are inlined between the codes of a value instead. An example compressing URLs with FSST is given in Figure 2.14. An important aspect of FSST is that the symbol table is static during decompression, allowing strings to be decoded independently and making the encoding suited for random access patterns (2).

Adaptive Lossless Floating-Point Compression Adaptive Lossless Floating-Point (ALP) compression targets IEEE 754 (44) doubles with a lossless transformation into a representation that is more compressible by other LWCs. IEEE 754 represents 64-bit floats

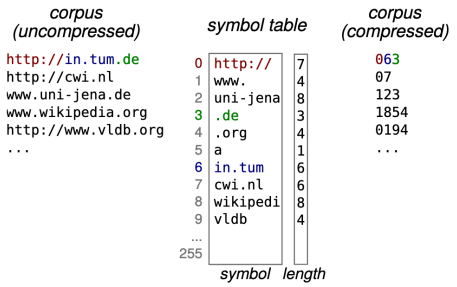


Figure 2.14: An example of FSST compression, where strings are divided into symbols of length 1-8 and symbols are replaced by 1-byte codes. Image from Boncz et al. (2).

(doubles) as a sign, an exponent e and a fraction, which corresponds to the binary scientific notation. While this representation is efficient for computation, it complicates compression as arithmetic over doubles introduces rounding errors, which can result in a lossy encoding of the original representation. To attain lossless compression while also exhibiting a good compression ratio, ALP implements two strategies: ALP and ALP_{rd}. ALP is able to switch between these strategies by sampling on a row-group granularity.

ALP encodes doubles in a block by finding a shared exponent e and factor f such that for each double n the encoding step becomes $ALP_{enc} = round(n \times 10^e \times 10^{-f})$ resulting in an integer d, and decoding becomes $ALP_{dec} = d \times 10^f \times 10^{-e}$. For the encoding, a higher exponent e increases the chances of a successful (lossless) encoding, as the difference between the real and exact value due to rounding errors becomes negligible. While a higher exponent increases the success rate, it results in tails of 0-digits when combined with low-precision decimals. To combat this, a factor f is introduced to cut these tails so that the resulting integers can be stored in a smaller bit representation. When encoding doubles in a block not all encodings are guaranteed to succeed, therefore ALP verifies the transformation for every value by encoding and decoding, comparing the result to the original representation. Whenever an encoding has failed the corresponding double is stored uncompressed in an exception segment. As ALP mainly changes the binary representation and does not compress the data, the authors combine ALP with FFOR, which is an implementation of FOR fused with bit-packing to save on a load and store instruction.

In case the aforementioned encoding is not desirable, either due to the high amount of exceptions or the size of the resulting integers, ALP switches to ALP_{rd}. This encoding exploits a low variance in the bit-representation of a double. ALP_{rd} determines with sampling a position p such that the highest $64 - p$ bits exhibit low variance across the

doubles in the block. Position p is then stored once per block, and the right side of each double is bit-packed. For the left side of the double a combination of a skewed dictionary and bit-packing is used. A skewed dictionary is an encoding which is able to handle exceptions (45).

2.6 DuckDB

DuckDB is an in-process DBMS designed for OLAP workloads. It caters to end users who need interactive data analytics, similar to users of Pandas, but with improved performance and consistency guarantees. Another use case is to allow edge devices to preprocess data locally before sending it to a centralized service.

2.6.1 Vectors

DuckDB takes inspiration from the vectorized processing model, as such it defines `DataChunk` objects which flow through its operators. Each `DataChunk` consists of a set of vectors with a default size of 2048, representing a horizontal slice of a set of columns. DuckDB supports compressed execution, which allows the query engine to operate over compressed data. To support this, a distinction is made between a vector's logical data and physical representation. Currently five different physical representations are supported: flat, FSST, constant, dictionary and sequence vectors.

Flat vectors have an identical logical and physical representation, with all elements stored as a contiguous array. Constant vectors are stored as a singular value. Dictionary vectors consist of a selection vector that stores indices and a child vector that stores the values. Sequence vectors are stored with a base value and an increment step. FSST vectors store compressed strings as a contiguous array of bytes, along with a decoder instance that can reinterpret the compressed bytes into UTF-8.

Vectors within DuckDB are represented by a `vector_type`, `type`, `data`, `buffer`, `validity` and `auxiliary` properties. The `vector_type` indicates the physical representation, used to interpret the buffers associated with `buffer` and `auxiliary`. The `type` represents the logical type of a vector. This retains semantic information like dates, and is able to represent complex types such as a struct or list. The `buffer` can be instantiated with an instance of `VectorBuffer`. Depending on the vector type this can hold different data. For example, a flat vector would store values whereas a dictionary vector would hold a selection vector. The `data` property then offers an untyped pointer to `buffer` through which data is read or written. Similarly, `auxiliary` can also hold a `VectorBuffer` to store additional data,

which would be the child vectors containing values in case of a dictionary vector. Last, `validity` represents a validity mask over the data stored in the vector.

German Strings To store strings in vectors, DuckDB uses the layout depicted in Figure 2.15, first introduced in the Umbra system (3). This layout was inspired by the realization that most strings stored in databases are short. Furthermore, strings in databases tend not to be changed often, allowing for optimizations by using immutable representations. The layout consists of a 4-byte header and a 12-byte body that can have two differing representations. The header stores the length of the string, which can represent a maximum length of just under `4GB`. Whenever a string can be stored in 12 or fewer bytes it is inlined directly in the body, otherwise the body stores a 4-byte prefix along with 8 bytes for a pointer referencing the complete string. In case of short strings, directly inlining saves a pointer indirection. For long strings the inlined prefix can aid in operations like equality and lexicographic sorting, providing a fast path saving on a pointer dereference. The layout is sometimes referred to as "German Strings", as it was developed by the database group at TUM. There are some differences however, as Umbra is a disk-based DBMS it uses storage classes to indicate the lifetime of each string. DuckDB is memory-based and as such does not implement storage classes (46).

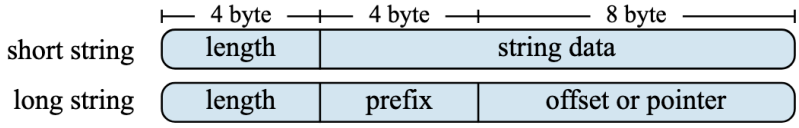


Figure 2.15: String layout within DuckDB vectors. Image from Neumann et al. (3).

2.6.2 Query Lifecycle

In this section we sketch a rough outline of the query lifecycle within DuckDB. As an extension indirectly interfaces with this lifecycle, understanding the application flow helps reason about the correctness of additional functionality. We distinguish the following phases in order: parsing, binding, constructing the logical plan, optimizing the logical plan, constructing the physical plan and execution.

Parsing The first step within the lifecycle of a query is the parsing of the textual SQL statement, such as in Listing 1, into an abstract syntax tree (AST). DuckDB borrows from the design of PostgreSQL to parse the textual statement into an intermediate AST, saving

on custom grammar and parsing logic. If the parsing step fails, DuckDB tokenizes the textual statement and retries parsing on each partition. When a retry fails, it is delegated to the available extensions, if any are registered, which are given an opportunity to parse the segment in question. The resulting PostgreSQL-compatible tree is then immediately transformed into a DuckDB-style AST, by mapping each PG-style node to an `SQLStatement`. As the PostgreSQL parser can return multiple trees, there can be multiple `SQLStatement` representing different DuckDB-ASTs.

The specific instance of a `SQLStatement` determines how the respective tree is structured. A common variant, `SelectStatement`, contains a `QueryNode` that is an abstract base class modeling `SELECT`-style query bodies. Another example is `CreateStatement`, which holds catalog information instead of a `QueryNode`. In short, these statements contain logical references to resources required to resolve the respective query.

Binding After all statements have been generated, each statement is handed off one-by-one to the planner. The planner first produces a bound AST based on the incoming AST, adding semantic meaning to the structural information present in the AST. This entails looking up AST identifiers for tables and columns in the database catalog, resolving the logical references to the physical location of these objects. It further verifies that the types of different objects are compatible with each other, with respect to the operator that is performed over them. For example, it will determine that `E.salary < 50000` is a boolean expression. The resulting AST is represented by the bound counterpart of elements in parsed AST, such as `BoundSelectNode`.

Logical Plan The bound elements in the AST are then transformed into a logical query plan by producing a tree of `LogicalOperators`. Each `LogicalOperator` represents an action that needs to be taken within a query plan. For example, the operator `LogicalGet` is a leaf node which should take a bound table reference and produce a stream of rows. As the query plan is a direct translation from the bound AST, parts of the query plan may be inefficient and can benefit from a rewrite. The optimizer takes this logical query plan and applies heuristics and cost-based optimizations. An example is predicate pushdown, this entails pushing down a filter condition as far as possible in the plan tree. The benefit being that early filtering results in later, possibly expensive, operators having to process less rows. A cost-based optimization is join-reordering, where different join orders are evaluated, selecting the order with the smallest intermediate result. The result of the optimizations is a new optimized logical plan, which is logically equivalent to the original.

Physical Plan The optimized logical plan is subsequently converted into a physical plan. While the logical plan determined what actions had to be taken, the physical plan is an executable definition of these actions. Each `LogicalOperator` is translated into one or more `PhysicalOperator`s. For example, `LogicalJoin` may be mapped to `PhysicalHashJoin`, which is efficient for large unordered sets. However, in case of a non-equijoin, it might be mapped to a less efficient `PhysicalNestedLoopJoin` instead.

Execution With the physical plan, the executor walks the tree of `PhysicalOperator`s and cuts these into source-operator-sink chains called pipelines. Each pipeline contains exactly one data source, zero or more intermediate operators and one sink. When all pipelines are defined, the executor asks each pipeline if all its operators support parallel execution. If they do, the pipeline is split into multiple tasks. Otherwise the pipeline runs as a single task. Tasks are then handed to the task scheduler, which feeds them to the thread pool. During the creation of pipelines the executor maintains a dependency graph between pipelines, influencing the order tasks are scheduled. The executor then enters a loop, each iteration it pulls a chunk from the source with `GetData`, processes it through intermediates with `Execute`, finally handing it to the sink with `Sink`. During execution, operators can return codes like `BLOCKED` or `NEED_MORE_INPUT` to reschedule tasks. When the source signals exhaustion, the executor asks each operator if it still has buffered work to flush. Finally, the sink finalizes, which could be either materializing buffered rows or committing writes.

3

Related Work

This chapter explores current columnar formats in detail. For each columnar format, we analyze the data layout, the implementation, and the associated reader and writer. This is followed by a discussion of the columnar formats covered, giving a taxonomy of the different feature sets and their respective strengths and weaknesses.

3.1 Parquet

Parquet (9) version `1.0` was released on July 30, 2013, as the result of a collaboration between Twitter and Cloudera (47). The initial goal was to provide an open-source columnar format for the Hadoop ecosystem, but the format has since grown into an industry standard with compatibility across a wide variety of frameworks and ecosystems. The format has been designed with complex data structures in mind, and therefore builds on the concepts introduced in the Google Dremel paper (48). It is further built to support both LWC and GPC schemes on a per-column granularity. In the subsequent sections, we take the latest version of Parquet (`2.11.0`) as a reference when describing the format itself. However, there exist many different reader and writer implementations of the specification with distinct feature sets. This makes the notion of a V1 or V2 ambiguous. Therefore, the later sections on the reader and writer focus specifically on the features supported by the implementation of DuckDB (49).

3.1.1 Layout

The layout of a Parquet file is represented in Figure 3.1. The file starts and ends with the same magic number `PAR1`. After the first magic number, the data contained in the format is represented by `ColumnChunk` blocks. Each row group is represented by a set of contiguous `ColumnChunk` blocks, and each `ColumnChunk` contains all entries of a column per row group.

Each `ColumnChunk` subsequently consists of a contiguous set of `DataPage` blocks. Inside each `DataPage` is a header containing information on the page type, the uncompressed size, the compressed size, an optional CRC32 checksum, and data associated with the specific page type. The file metadata resides after the `ColumnChunk` blocks at the end of the file. It contains information on the column types, optional key/value pairs, and metadata per row group. Putting this metadata at the end of the file allows for a single-pass write. After the file metadata, the length and the trailing magic number follow.

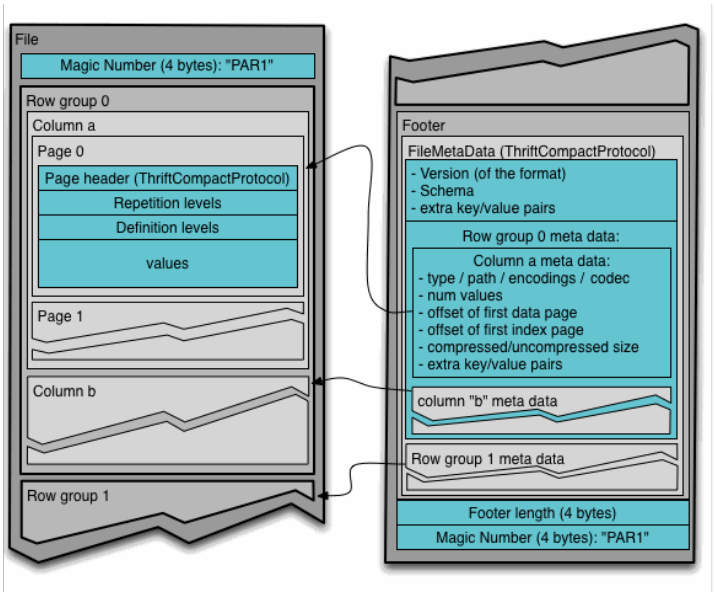


Figure 3.1: Internal layout of a Parquet file. Image taken from Apache Parquet documentation website (4).

Apache Thrift Apache Thrift (50) is a software library with code-generation tools that, through an Interface Definition Language (IDL), can generate code in a multitude of programming languages. Parquet defines its metadata structures using Thrift, and (de)serializes the metadata using Thrift Compact Protocol. The Compact Protocol encodes data into a sequence of fields, with each field having a header that includes a type specifier and a unique field identifier. To save space, booleans are inlined into the field header by having type codes for the respective values `BOOLEAN_TRUE` (`0x01`) and `BOOLEAN_FALSE` (`0x02`). Field IDs are further stored as a delta from the previous field ID if the delta is smaller than 15, otherwise they are stored as a variable-length integer (varint). Further, signed integers are zig-zag encoded to reduce the space requirement of small negative numbers. Zig-zag encoding maps signed integers to unsigned integers by placing the sign in

the least significant bit. This ensures that small negative and positive values both require few bytes when encoded as varints (51, 52, 53).

Pages Pages are the smallest granularity on which the Parquet format operates. There are four types of pages: `DATA_PAGE`, `INDEX_PAGE`, `DICTIONARY_PAGE`, and `DATA_PAGE_V2`. The `DATA_PAGE` and `INDEX_PAGE` are deprecated.

All page types share a common base header struct, with the `PageType` property determining which additional data is included in the `PageHeader`. The `uncompressed_page_size` is the size of the uncompressed page in binary representation excluding the header. Readers use it for pre-allocating the required memory for decoding in advance. The `compressed_page_size` is similar to the previous value, but records the compressed and possibly encrypted size. When a page is uncompressed, `compressed_page_size` is equal to `uncompressed_page_size`. The `compressed_page_size` can be used by the reader to skip over the body of a `DataPage`, based on other information stored in the header.

The `DATA_PAGE_V2` represents a data page that contains the repetition and definition levels, and the stored values. The repetition and definition levels of this page are always `RLE` encoded, in contrast to `DATA_PAGE` where levels could be either `RLE` encoded or bit-packed. Further, in V2, GPC no longer compresses both the levels and the values. Leaving the levels uncompressed allows a reader to interpret the levels without decompressing the entire page.

The `DICTIONARY_PAGE` represents a dictionary page. There can only be one such page per `ColumnChunk`, and it should always be at the beginning of the `ColumnChunk` block. Whenever a dictionary page is present, each value in subsequent data pages is written as the key associated with the value defined in the dictionary.

Indexes Parquet can enable more I/O-efficient point lookups and range scans through optional `ColumnIndex` and `OffsetIndex` metadata. These index structures are located just before the footer and are referenced per `ColumnChunk` through the `column_index_offset` and `offset_index_offset` fields present in the footer. The `ColumnIndex` provides per-page statistics such as minimum and maximum values, null presence, and ordering information. When a predicate is applied, a set of candidate pages from a `ColumnChunk` is constructed based on the `ColumnIndex`. For sorted columns, candidate pages can be identified using binary search, while unordered columns require a linear scan. When a `ColumnIndex` is present, a corresponding `OffsetIndex` must also be present. The `OffsetIndex` maps data

pages to row index ranges, which are used to reconstruct fields over different columns based on the candidate pages (54).

Bloom Filters Parquet supports Bloom filters at the `ColumnChunk` level. A Bloom filter provides a probabilistic membership test with a tunable false positive probability but guarantees no false negatives. This makes them well-suited for deciding whether an entire `ColumnChunk` can be skipped without fetching its pages. Bloom filter data is stored separately from the row groups and referenced through the `bloom_filter_offset` in the footer, usually placed after the row group data and before the page index structures. They are especially valuable for high-cardinality columns where dictionary encoding is inefficient and page-level statistics have little selectivity. For example, in a column of UUIDs, most values will be unique. Dictionary encoding would result in a larger size compared to plain encoding. Page statistics for min and max would cover a significant range as UUIDs are generated (pseudo)randomly. Now predicate pushdown of the form `WHERE uuid = 'xxx...'` most likely cannot rule out any page as every min/max range overlaps the query. With a Bloom filter, the reader would test for membership in the `ColumnChunk` block. If the value is not present, the block can be skipped. If the value might be present, the data pages of the respective block are fetched (55).

3.1.2 Encodings

The GPC encodings supported by Parquet are: `SNAPPY`, `GZIP`, `LZO`, `BROTLI`, `ZSTD` and `LZ4_RAW`. GPC is performed at page granularity. This allows both for partial and parallel decompression based on the query (56). The supported LWC encodings are: `PLAIN`, `RLE`, `DELTA_BINARY_PACKED`, `DELTA_LENGTH_BYTE_ARRAY`, `DELTA_BYTE_ARRAY`, `RLE_DICTIONARY`, `BYTE_STREAM_SPLIT` (57).

Nested Encodings To support nested types, Parquet borrows from the approach described in the Google Dremel paper (48). The authors describe a columnar storage format for nested data by dissecting nested records into separate columns using repetition and definition levels. Each primitive leaf path in the schema of a nested object is stored as its own column.

Repetition levels encode the depth of repeated fields, telling the reader whether the current value continues the same container or starts a new one. For example, considering an object with path `Name.Language.Code` that occurs multiple times. When the path `Language.Code` occurs for the second time within the same `Name` field, it would be denoted

with a repetition level of 2, as the field `Language` has repeated. When the entire path `Name.Language.Code` occurs for the second time it would result in a repetition level of 1, as the field `Name` has repeated.

Definition levels specify why a certain value is missing. It does so by adding a `NULL` entry in the column slice corresponding to the respective field. The definition level is the number of fields along the path that are defined (non-null). For example, consider an instance of the path `Name.Language` with the `Code` field missing. The last defined field in the path is `Language`, setting the definition level to 2.

The benefit of the aforementioned approach is that it allows for projection and predicate pushdown on nested fields, saving on the amount of data that needs to be read. Furthermore, as leaf fields are stored in separate columns they retain the benefits of a columnar format, such as vectorized processing and compression over the same domain.

Parquet applies this approach by storing the columns as separate `ColumnChunk` blocks in the file. Within each `DataPage`, repetition and definition levels are stored in the body after the `PageHeader`.

3.2 Vortex

Vortex is a next-generation columnar format. It has been developed by SpiralDB (58) and was initially released as a way to work with compressed Arrow arrays on-disk, in-memory, and over-the-wire (59). It has since expanded its scope as an open, extensible columnar format, with the project being donated to the Linux Foundation and becoming an incubation stage project (60). Vortex offers pluggable encodings and compression strategies that are able to be cascaded. It is designed for random access via SIMD and SIMT (GPU) and can support pushdown on compressed data. Vortex is implemented in Rust, and the latest version is `0.51.0`, with backwards compatibility enforced for all versions since `0.36.0` (14).

3.2.1 Layout

The Vortex file format has a very small definition, comprising magic numbers, segments of binary data, a version tag and postscript data. To understand how Vortex functions in a more typical scenario, we include Figure 3.2, highlighting the binary contents with a more elaborate use of Vortex layouts. A Vortex file always starts and ends with the same magic number `VRTX`. Prior to the trailing magic number are the version tag and the Postscript length. The postscript contains the location information for the DType, Layout, Statistics

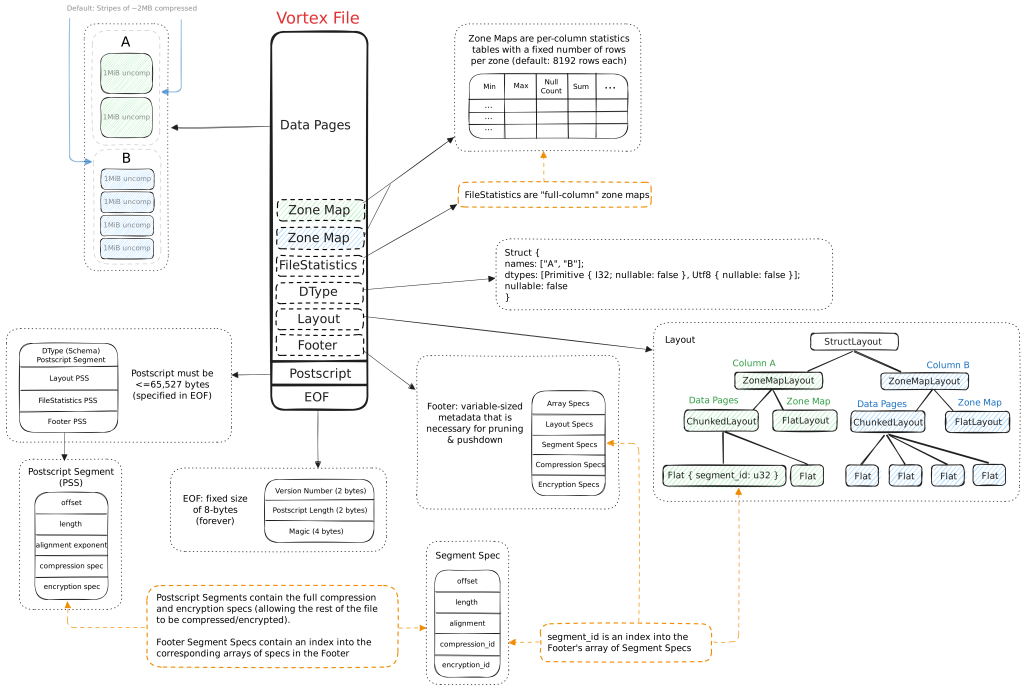


Figure 3.2: Internal layout of a Vortex file. Image taken from Vortex documentation website (5).

and Footer segment. Each segment in the footer contains an offset and a length, along with specifications for compression and encryption that might have been applied to the segments. The specifications are defined as inlined strings (`vortex.<some-scheme>`). These strings map to a runtime registry, defined in code with `VTableRegistry`, which functions as a hash map where values point to the respective implementations. The postscript is fixed-size (64 kB), with EOF included, so all necessary footer information can be read in two round trips.

The footer segment functions as a registry of which its values are referenced by other metadata structures in the file, such as layout nodes and Vortex arrays. During a read, the registry uses its values to construct a `VTableContext` by translating the string literals (e.g. `vortex.flat`, `vortex.chunked` for layout entries) into the respective runtime implementations, similar to the postscript. The `VTableContext` is then indexed by identifiers in the other metadata structures, resulting in behavior similar to dictionary encoding. The registry in the footer is composed of five topic-specific registries, each building their own `VTableContext`. The `array_spec` and `layout_specs` list every scheme encountered in the remainder of the file. The `segment_specs` is an ordered map of all physical blobs referenced by the layout tree. Each entry in this registry records the absolute byte offset from the start of the file, the length in bytes, and alignment information.

The `segment_specs` thus acts as a mapping from logical ids (indexes) to their respective physical locations. The `compression_specs` lists every block-based compression scheme used in the remainder of the file. The idea is that after Vortex arrays have been encoded using LWCs, a block-based compression scheme can be applied to the associated buffers before they are flushed as segments to disk. However, currently this behavior is not fully implemented and the Vortex array buffers are always flushed directly to disk. The `encryption_specs` list encryption schemes in the remainder of the file and function in a similar manner to `compression_specs`. Encryptions too have not been implemented to date. Both `compression_specs` and `encryption_specs` are already defined, however, to ensure compatibility with future Vortex format versions.

The layout segment contains a recursive data structure, representing a tree of layouts describing the physical layout of Vortex arrays. Each object in the tree contains an identifier for the layout type, which can be one of the following: `Flat`, `Chunked`, `Struct`, `Zoned` or `Dict`. These identifiers are stored as integers, and index into the `layout_specs` described earlier. Each node also records the number of rows it covers (for row-based pruning), opaque metadata required by that specific layout, its child layout objects, and a list of segment identifiers. Those segment identifiers are indices into the footer's `segment_specs` array, tying the logical node back to the concrete byte ranges that store its buffers.

The DType segment stores the logical schema represented by the file once, as a union with variants like `Null`, `Primitive`, and `Struct` (the latter is also present in Figure 3.2). Depending on the variant, properties include field names, nullability, and the physical type. The schema carried by DType is threaded through layout serialization and, more importantly, through deserialization so each child layout can be instantiated and validated by the correct runtime implementation in accordance with its physical type. Separating the DType from the layout prevents every node in the layout tree from inlining the same schema metadata. In addition, the DType segment is optional and can be provided externally at read time, allowing the schema to live in a separate metadata layer.

FileStatistics FileStatistics is an optional metadata structure referenced by the postscript. It contains an `ArrayStats` entry per top-level field in the schema (or a single entry if the root type is not a struct). Each entry carries global statistics such as min, max, null_count, is_constant, among others. These statistics are then consulted at the beginning of a scan, before touching the layout tree, to determine if a file or specific columns can be skipped given a predicate. Scanning can then continue for the remaining columns, possibly checking additional more fine-grained statistics deeper in the format.

FlatBuffers FlatBuffers is a cross-platform serialization library (61), originally developed by Google for performance-critical applications. It requires no temporary data structures, allocations, or copies during read and write operations. It does so by storing the data as a binary buffer containing nested objects, organized using offsets. Binary buffers are written and interpreted by a schema defined in an intermediate representation (IR). The IR is then used to generate code in a supported target language which provides writing and accessor functions based on the schema. A benefit is that much of a binary format can be encoded in the generated code, saving on byte size of the resulting binary buffers. Furthermore, a schema can generate read functions that do not need to parse data to access a buffer.

Editions Vortex proposes the use of "Editions" to group feature sets contained in the file format. Features include the encodings, layouts, compression schemes and encryption algorithms. Editions follow the format `YYYY.MM.DD`, allowing writers to be configured on a moving scale. For readers, the minimum version required solely depends on the features used and not necessarily the edition the writer used.

Vortex further introduces the notion of backwards and forwards compatibility. With backwards compatibility readers using newer versions or editions are able to read files written by older writers. Forwards compatibility is currently mostly conceptual. The idea is to embed WebAssembly kernels, or references to a hosted location, during writing of the encodings used. The reader would then use the embedded kernels to decode unsupported encodings (62).

3.2.2 Vortex Layouts

Vortex's key property is that it doesn't hard-wire one on-disk layout; it stores a tree of layout nodes describing how data is partitioned and organized. Because that tree is serialized as metadata inside each file, writers are free to choose the layout strategy they need, and the resulting file remains self-describing. In effect, a Vortex layout is an embedded physical query plan for Vortex arrays on disk. Vortex offers a set of different layout types, forming the building blocks of layout trees and which can be composed by a writer depending on their desired partitioning and pruning strategy.

- **FlatLayout** is the only layout that owns on-disk bytes. It contains a single segment id which indexes into `segment_specs` from the footer. The referenced segment contains all required buffers and metadata for (a part of) a serialized Vortex array.

- **StructLayout** is a container for child layouts. It is used to present multi-column schemas or complex types. Each child corresponds to one field's layout and can enable column projection.

- **ChunkedLayout** is used to partition child layouts row-wise, storing their row counts so the reader can map global ranges to per-chunk ranges. This layout results in a structure similar to that of a row group.

- **ZonedLayout** is a wrapper over a data child layout and a zonemap child. The zonemap child contains a FlatLayout with statistics of the child layout. The writer can decide which statistics to collect, and these statistics usually include values such as min and max. Zonemaps are useful for predicate evaluation, allowing coarse-grained row-level pruning.

- **DictLayout** has two children: one for the dictionary values (usually a FlatLayout), and one for the codes (which can be of arbitrary complexity). During writing, the layout strategy maps each incoming value to a dictionary id, streaming the deduplicated dictionary into the values child and the codes into the other child. At read time the runtime materializes the values once and joins them with the codes to form a logical dictionary array. In short, DictLayout is the building block for dictionary-encoded columns, letting writers store the dictionary and codes with independent layout strategies.

3.2.3 Encodings

Vortex currently supports `Zstd` for GPC, and the following LWC encodings: `ALP`, `BitPacked`, `ByteBool`, `DateTimeParts`, `DecimalByteParts`, `Delta`, `Dict`, `FSST`, `FOR`, `PCO`, `REE`, `Sequence`, `Sparse`, `ZigZag`. Most of these encodings have been discussed in Section 2.5, so we discuss only the newly introduced encodings here.

`ByteBool` represents each boolean as a separate `u8` with a validity array. Of note is that while a Vortex file is able to write and read this encoding, it will not be written by the current writer as `EncodeVTable = NotSupported`.

`DateTimeParts` is a structural decomposition of a UNIX-epoch-based date-time value into days, seconds and sub-seconds arrays. The days array is nullable, and if it is `NULL`, this implies that the associated indices in the other arrays are also `NULL`. The decomposition then allows the compressor to choose the most optimal encoding for each array independently.

`DecimalByteParts` splits a `Decimal` type into a most significant part (MSP) and optional two or three lower parts. MSP stores the top bits, and the lower parts store the remaining bits in order. When MSP is `NULL` the associated lower parts are also implied to be `NULL`. The encoding exploits data where the MSP has a small range, and each section can be encoded independently by the compressor. Of note is that the lower parts are currently unused, which makes `DecimalByteParts` similar to a `PrimitiveArray`, with the exception that values are stored as a base value and scale factor instead.

`PCO` is an implementation of Pcodec by Loncaric et al. (6). Pcodec is a format and algorithm for losslessly compressing float or integer sequences. It does so by expressing numbers in latents, which are designed to approximate independently and identically distributed (IID), nearly uniform unsigned integers. Latents are then further compressed with binning. In Figure 3.3 the processing steps are visualized. First, Pcodec performs a transformation (e.g. XOR, FOR) over the distribution resulting in one or two latent variables. The transformation can be one of four modes, each exploiting different characteristics in the distribution, the mode being chosen by sampling numbers and comparing the resulting bit sizes. Next it uses delta encoding within each variable to remove dependencies between variables, so that the resulting latents are closer to IID. Finally it performs binning over the resulting variables. While the aforementioned steps reshaped the distribution, binning actually reduces the data size. Histogram bins are constructed over each latent variable, with each latent subsequently being encoded by its entropy-coded bin and offset into that bin. Pcodec formats the results into a header with one or more chunks. A chunk is the unit of compression and so each chunk has its own mode, delta encoding and binning configuration. Each chunk consists of metadata and one or more pages. As pages store initial states for entropy and delta encoding they can be decompressed in any order. The design of Pcodec sacrifices compression and decompression speed in favor of increased compression ratio. It offers a better compression ratio compared to GPCs, and in case GPCs match the compression ratio Pcodec offers better speeds. Vortex uses Pcodec in conjunction with Zstd in a separate compressor `CompactCompressor`, as Pcodec is not compatible with cascaded encodings.

`Sequence` encodes a sequence for an array A as the equation $A[i] = base + i * multiplier$. Only the base, multiplier and length need to be stored, in contrast to other encodings such as delta or FOR which result in per-element residuals. A limitation is that it requires the data to follow a linear pattern, as any deviations break the model.

`Sparse` encodes either a constant array when all values are equal, or as a fill-with-patches layout otherwise. A constant array is represented by a value and the length. In a fill-with-

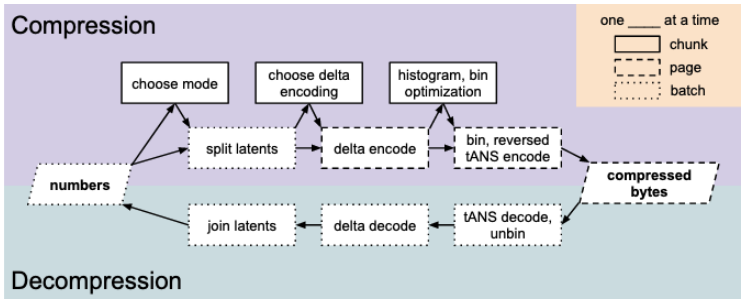


Figure 3.3: The processing steps involved in Pcodec compression and decompression. Image from Loncaric et al. (6).

patches layout, a fill is represented by a scalar value, and patches are represented by an array of indices and an array of values. The indices indicate the logical index for an associated value in the values array.

3.2.4 Encoding Selection

Vortex currently has two strategies for selecting encodings, by using either `CompactCompressor` or `BtrBlocksCompressor`. When writing a Vortex file `BtrBlocksCompressor` is the default, but can be overridden through explicitly defining another compressor in `WriteStrategyBuilder`.

`CompactCompressor` compresses numeric types, currently 16, 32 and 64-bit integers and floats, using `PCO`. `PCO` offers a configuration for the compression level that controls the number of histogram bins used when encoding latents. For all other types the compressor uses Zstd. It similarly offers a compression level configuration.

`BtrBlocksCompressor` is a compressor based on the work of Kuschewski et al. (10). In this work the authors propose `BtrBlocks`, an open columnar compression format for data lakes. `BtrBlocks` divides each column into fixed size blocks, similar to other formats such as Parquet. It then uses a predefined set of LWC schemes to compress the blocks on an individual basis. The distinguishing factor is that `BtrBlocks` does not do a single encoding pass, but it can recursively encode the output of a previous encoding, also known as cascaded encoding. To determine the appropriate encoding for a block the authors propose a sampling method that estimates the compression ratio, choosing the highest performing encoding. First statistics like min, max and unique count are collected from a block to rule out nonviable schemes through heuristics. Next 1% of the data is sampled in 10×64 non-overlapping randomly chosen chunks to preserve spatial locality and acquire a low bias representation. These chunks are then compressed using the left-over encodings, and the encoding with the highest compression ratio is then used to compress the entire

block. If the output is in a compressible format, the process starts again from statistics gathering. The authors define a max recursion depth of three, no reasoning is given but a possible justification would be that the data does not significantly reduce in size after a third pass, with respect to the additional cost in encoding and decoding speed. Vortex applies the cascaded encodings with its own set of encodings as defined in Section 3.2.3. With encodings applied on a per array basis. Similar to the original paper, Vortex sets the maximum recursion depth to three.

3.3 FastLanes

FastLanes (63) is a next-generation columnar format developed at CWI (64). It was designed with evolved features from existing data formats. Because FastLanes does not depend on GPCs and uses data-parallel lightweight cascaded encodings, it offers up to 40% better compression and 40x faster decoding compared to Parquet. In addition, through expression encoding FastLanes is able to exploit relationships between columns for increased compression. FastLanes is implemented in C++, does not require external libraries, and is SIMD-friendly with zero explicit SIMD instructions. This makes the code portable and allows the compiler to auto-vectorize depending on SIMD support on the target platform. FastLanes is in active development with the most recent release being version `0.1.4`, the code is open-source and accessible via its GitHub repository.

3.3.1 Layout

Internally, a FastLanes file starts with a header followed by a sequence of segments, with at the end a table descriptor and footer. A visual representation of the layout is given in Figure 3.4. In the remainder of this section we describe the metadata structures in order of scope, finishing with a discussion on the representation of data within a segment.

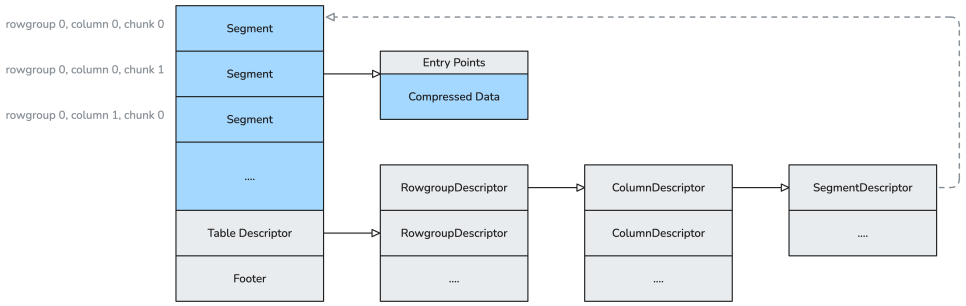


Figure 3.4: Generalized internal layout of a FastLanes file. Blue denotes the presence of table data, gray denotes FastLanes specific metadata.

Metadata The header is the entry point of a FastLanes file. It contains magic bytes representing "FastLane", the version of FastLanes with which the file was written and 8 configuration flags. Currently one configuration flag is defined, the others are reserved for future use. The flag `inline_footer` determines if the table descriptor is inlined in the FastLanes file or defined as a separate file. The location of the external footer is currently limited to the parent path of the FastLanes file, with a fixed name "table_descriptor.fbb". In case the table descriptor is inlined, the footer provides the same magic bytes and an offset and size of the table descriptor.

The table descriptor acts as the container for all metadata required to interpret the sequence of segments in a FastLanes file. As such, it contains the total binary size of all segments, and a vector of row group descriptors.

Each row group descriptor contains the binary size, the offset in the file, tuple count, vector count and a vector of column descriptors. Since FastLanes encoding and decoding operate over a predefined vector size, adding a tuple count is essential to be able to determine the number of rows which are valid in the last vector of a row group. In contrast, the vector count can be derived from the tuple count but is added explicitly for convenience.

The column descriptor is the most complex metadata object within FastLanes, it contains metadata used for row group pruning, currently consisting of the max value and a null count. In addition, it contains the binary size, the offset, a data type, index and column name. The data type represents the semantic meaning of the data stored in the associated segments, this is especially important as some encoding operators can erase the ingested type during compression. For example, the signedness of integers is removed when encoding with `FFOR`, transforming everything to unsigned integers. During decoding the data type stored in the column descriptor is then used to determine if the original type was signed or not. In case a column stores decimals, an additional object is stored alongside the data type containing the scale and precision of said decimals. To support complex types, a column descriptor can contain a vector of 'child' column descriptors, creating a recursive structure. Regarding segments, column descriptors store a vector of segment descriptors, a modified form of Reverse Polish Notation (RPN) to describe expressions, and an expression space. The expression space is constructed during writing and includes all tried encoding variations on the respective column with the resulting size. This may help with analysis of the encoding behavior, but it is not required for correctly reading a FastLanes file. The definition and use of expressions is worked out in Section 3.3.2.

The segment descriptor describes two areas within a segment: the block of compressed data with an offset and size, and an entry points array with offset and size. The entry points

array lives at the start of a segment, it provides the starting offset for each vector within the data block. The entry points array itself is indexed by storing the entry value bit-width in the descriptor. The ability to fetch individual vectors of values allows improved random access performance compared to page-based solutions, as there is less data transferred on a per-row access (7).

3.3.2 Expression Encoding

FastLanes uses an abstraction named *Expression Encoding* to describe how compressed data should be interpreted. It uses a form of Reverse Polish Notation (RPN) to describe a program, or expression, used to encode and decode a column. The operators represent the actions to take, whereas the operands represent the data the actions should be performed over. Both are encoded as integers, but operators represent an enum from Table 3.1 while operands reference segments in a FastLanes file. Unlike classical RPN, operators may have arbitrary arity, as each operator's implementation dictates what arguments it consumes and what auxiliary metadata it expects. Here arguments denote information coming from the stream of operands, such as the column reference when using the expression `EQUALITY`. Auxiliary metadata supports the function of operators, and includes statistics or dictionaries. Together the operators and operands can be interpreted in either direction: to compress incoming vectors into a FastLanes compatible layout, and to materialize the compressed data back into the original vectors. To build an intuition on how this abstraction is applied, we describe both encoding and decoding with an example.

Encoding When a candidate expression is considered during the selection stage (Section 3.3.5), it is first materialized into an executable plan by an interpreter. The interpreter instantiates a fixed sequence of C++ operator objects that share transient buffers, and reserve space for their output segments by appending operand tokens to the respective column descriptor. Then, each vector is run through the materialized plan, filling the buffers. After all vectors have been processed, each operator finalizes row group scoped state after which a flush operation collects the segments, which are then inspected by the Wizard.

For `EXP_DICT_I16_FFOR_U08` the interpreter would emit the following operator pipeline: `enc_dict_opr<i16>`, `enc_dict_map_opr<i16,u8>`, `enc_analyze_opr<u8>` and `enc_ffor_opr<u8>`.

- `enc_dict_opr<i16>` constructs a row group scoped segment to hold key values during interpretation, these keys are then loaded in from an earlier statistics gathering step during finalization. It does nothing during the per-vector execution phase.

- `enc_dict_map_opr<i16,u8>` translates for each vector the current set of values into the corresponding dictionary ids, writing the results to an owned transient buffer instantiated during interpretation.
- `enc_analyze_opr<u8>` reads for each vector the transient buffer of `enc_dict_map_opr<i16,u8>` with the current dictionary ids, determining the correct bit-width and base values.
- `enc_ffor_opr<u8>` uses the transient computed values of `enc_analyze_opr<u8>` by running a FFOR kernel, of which the result is appended to a bit-packed segment. In addition, both the base and bit-width are appended to their respective segments.

This example also illustrates the two execution scopes: some operators only do work once per row group, while others run every vector and mutate only in-memory buffers. Only the operators with persistent artifacts flush segments to disk; the rest exist to feed those producers with transient state.

Whenever an operator requires transient state, its constructor walks back through the `PhysicalExpr` via a visitor so it can grab typed pointers into the upstream operator's buffers. This gives a consumer-driven dependency direction, explicitly encoding intra-operator dependencies.

Decoding Decoding reuses the same RPN program but instantiates decoder operators instead: the interpreter rewinds the operand indices and builds a `PhysicalExpr` whose operators read the segments laid out during encoding. For `EXP_DICT_I16_FFOR_U08` the pipeline becomes: `dec_unffor_opr<u8>` followed by `dec_dict_opr<i16,u8>`. Note that this pipeline is simpler, as it does not require any analysis but only decodes compressed values.

The un-FFOR operator retrieves the segments based on the operand tokens added during the encoding process, which results in three segments: bit-packed values, bit-width and bases. During per-vector execution it invokes the un-FFOR kernel, recovering the dictionary ids in a transient buffer. The dictionary operator uses the remaining operand token to point at the key segment, zips those keys with the decoded ids, and materializes the original 16-bit values. Because the encoder already stored operand tokens in the same order the descriptors were flushed, the decoder can deterministically find every segment without extra metadata.

Note that when executing a pipeline we can choose up to which operator we uncompress the source vector, by which we can emit partially decompressed data to a target system.

The extension makes extensive use of this capability, as DuckDB supports compressed vectors itself, saving on a potentially redundant decompression-compression step.

3.3.3 Unified Transposed Layout

Section 2.2.2 discussed how SIMD can improve performance, but also highlighted challenges in using it effectively. FastLanes operates over vectors of a set size, lending itself well to the use of SIMD. However, as mentioned before, the SIMD landscape is comprised of heterogeneous ISAs. Thus, using explicit SIMD intrinsics would explode the complexity of the FastLanes code over time, exacerbated by the high likelihood of future SIMD expansions. To support SIMD, FastLanes proposes the use of a virtual 1024-bit register, of which its instruction set is implemented in scalar code. Importantly, the supported instructions are comprised of the common denominator of all SIMD instruction sets. This allows the compiler to auto-vectorize the instructions based on the target platform without requiring knowledge on per-platform SIMD specifics.

In addition to a virtual register FastLanes introduces a SIMD-friendly data layout. When values exhibit sequential dependencies between each other they end up in adjacent lanes in a SIMD register. In case of `DELTA` this makes a SIMD addition impossible, requiring more expensive lane-crossing operators. To reduce the effects of sequential data dependencies FastLanes combines two ideas: (i) instead of having one entry point per vector there are multiple, where each entry point covers an equal part of the vector, and (ii) the values in a vector are reordered into SIMD register-sized chunks, with no dependencies between values in a chunk and where dependencies between chunks map to the same SIMD lane. This translation is visualized in Figure 3.5. Generally, the resulting matrix of vertically stacked chunks has T rows and S columns, where each row represents a chunk and each column a SIMD lane. For the 1024-bit register T is the value bit-width, with S being 1024/T representing the amount of lanes.

The aforementioned layout has a limitation in that the ordering of values depends on T. This is important in the context of databases, where tables often consist of multiple columns, and where columns can have differing bit-widths. When the data layout of a column is transposed, it should be transposed the same way across all other columns to correctly reconstruct tuples in the relation. However, using the same order for different bit-widths may result in underutilization, because not all lanes may be filled when working with smaller bit-widths. Therefore Afroozeh et al. propose a layout that is compatible with all lane-widths, consisting of transposed tiles of 8x16 values, with 8 tiles per vector of 1024 values. The choice for 16 lanes in a tile gives that at the widest bit-width, a single

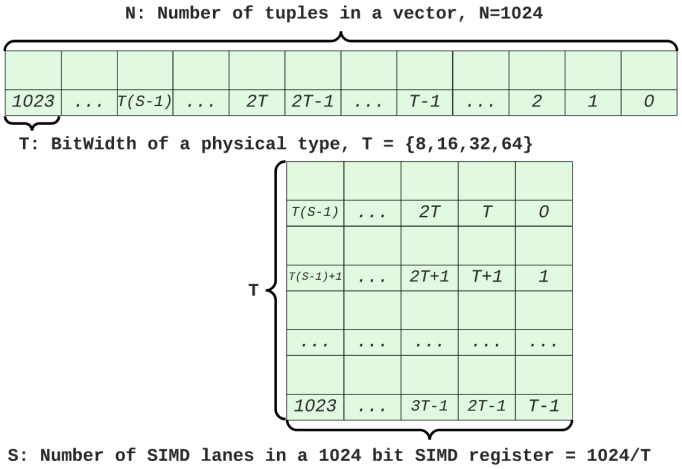


Figure 3.5: Transposed layout of a FastLanes vector, where ordering depends on the widths of S and T. Image taken from Afroozeh et al. (7)

row fills exactly one 1024-bit register. However, when working with smaller data-types, a single row may not fill the 1024-bit register. In this case multiple rows are grouped into a single register, sourced from independently processable tiles. This gives rise to the ordered sequence 04261537, where each digit represents a tile ID. Figure 3.6 highlights how for any lane-width values remain data-parallel when decoding. A consequence is that since data is stored in a transposed order, an untranspose step is required in case a query needs to preserve the logical order, for example when using `ORDER BY`.

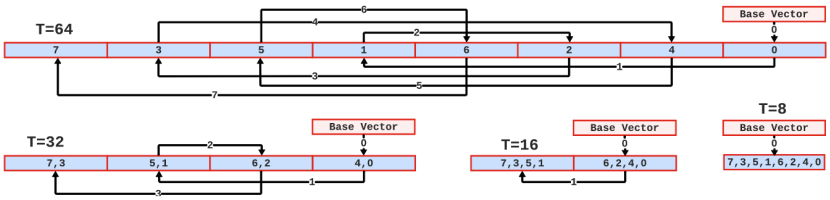


Figure 3.6: The Unified Transposed Layout 04261537 tile order for all lane widths, highlighting universal data-parallelism. The blue-red boxes represent one or more 8x16 tiles. Image taken from Afroozeh et al. (7)

3.3.4 Encodings

FastLanes uses operators as a model for its expression encoding mechanism. Operators represent a data structure containing functions for encoding and decoding, storing potentially partially encoded data. Operators are able to receive and pass on their state, forming a pipeline of transformations. FastLanes does not determine the set of operators

to combine for a particular column at runtime. Instead, based on benchmarking the Public BI dataset, it has predetermined a set of operator combinations which are most effective in terms of compression ratio and decoding speed. All combinations supported by FastLanes are represented in Table 3.1, for both `INTEGER` and `UX` the supported bit-widths are `8,16,32,64`. As most operators have LWC counterparts, we defer the reader to Section 2.5 for the respective definitions. In the remainder of this section we discuss operators not covered by previously discussed LWCs, or that do not directly map to an LWC.

string columns	numeric columns	floating-point columns	correlated columns
CROSS_RLE_STR	CONSTANT_INTEGER	ALP_DBL	EQUALITY
FSST_DICT_STR_FFOR_SLPATCH_UX	FFOR_SLPATCH_INTEGER	DICT_DBL_FFOR_SLPATCH_UX	
FSST_DICT_STR_FFOR_UX	DICT_INTEGER_FFOR_SLPATCH_UX	RLE_DBL_UX	
CONSTANT_STR	DICT_INTEGER_FFOR_UX	DICT_DBL_FFOR_UX	
EXTERNAL_FSST_DICT_STR_UX	CROSS_RLE_INTEGER	CONSTANT_DBL	
FSST_DELTA_SLPATCH	FFOR_INTEGER	ALP_RD_DBL	
FSST_DELTA	EXTERNAL_DICT_INTEGER_UX	EXTERNAL_DICT_DBL_UX	
FSST12_DICT_STR_FFOR_SLPATCH_UX	RLE_INTEGER_UX	CROSS_RLE_DBL	
FSST12_DELTA_SLPATCH	RLE_INTEGER_SLPATCH_UX	RLE_DBL_SLPATCH_UX	
RLE_STR_SLPATCH_UX			
FSST12_DELTA			
RLE_STR_UX			

Table 3.1: Supported expressions in FastLanes v0.1, grouped by encoding target type (8).

`FFOR`. This is a variant of `FOR` with fused bit-packing. It saves a SIMD store and load instruction between the FOR and bit-packing loop by immediately packing a calculated delta, without ever writing the intermediate delta to memory.

`SLPATCH`. Patching removes outliers from a vector and stores these as exceptions in a separate data structure. `SLPATCH`, also known as SelectionVector, uses separate arrays for exception values and exception positions. A vector is reconstructed by first looking up the size of the exceptions list, then iterating over the exceptions positions and exceptions values array up to the size. The exception value then overwrites the original value in the vector on the exception position for each iteration. FastLanes specifically uses `SLPATCH` as this technique is able to be data-parallelized on the GPU. This can be realized by writing exceptions first in order of lane, and second in order of position. Combined with additional metadata on the range of exceptions for each lane, every lane can decode its exceptions without dependencies on other lanes (65).

`GLUE`. The `GLUE` operator describes how two sources of bit-packed data should be merged. In case of ALP_{rd} encoding the double is split into two bit-packed arrays as described in Section 2.5, the `GLUE` merges the left and right side bits back into doubles. Currently this operator is only used for ALP_{rd} and as such is not independently defined in the code.

`Transpose`. The transpose operator puts a vector in the UTL as described in Section 3.3.3. This operator is only applied during encoding, leaving the untranspose operation to

the caller. In the current implementation of FastLanes only encodings that benefit from the UTL are transposed, making untransposing for these encodings necessary to correctly restore tuples. Future versions of FastLanes will most likely transpose all encodings, with an optional untranspose that is applied to all encodings in one go.

`Cast`. The `Cast` operator transforms the type of a column into a narrower type, supported casts are: string to integer, double to integer, and integer to a narrower integer type. The resulting type is not stored in the schema, thus care should be taken on the expected types when materializing values.

`FSST12`. This is a variant of `FSST`, but using 12-bit instead of 8-bit codes. This results in a larger maximum symbol table size of 4,096 symbols, where FSST has a maximum of 255.

`FastLanes-RLE`. FastLanes takes a different approach compared to classical RLE to improve SIMD acceleration. A vector is encoded by producing an array containing run values, and an index array that monotonically increases by 1 whenever a new run starts. The indices then map the run value to the correct location in the vector. The index array is then `DELTA` encoded and bit-packed. As `FastLanes-RLE` incurs a 128-byte overhead per vector, classical RLE is supported through the `Cross-RLE` operator in case a row group contains very few RLE values.

Multi Column Compression (MCC) FastLanes introduces a new category of compression schemes for columnar formats that exploits the relation between two columns. In addition, MCC includes schemes that split one column into multiple sub-columns which can then be encoded individually. Currently two MCC schemes are supported: columns which are equal and columns that have a one-to-one mapping between them. In the first case, only one column is stored on disk which is then referred to by both column descriptors. For the second case both columns are dictionary encoded, but one column only stores a reference to the indices of the other column. Due to how unique values are discovered, the indices are guaranteed to align with the dictionaries of both columns.

3.3.5 Encoding Selection

Within a FastLanes file, each row group is encoded independently. To select the optimal encoding for each column, FastLanes can be described to use a multi-stage funnel. The funnel performs relatively inexpensive checks, before resorting to more computationally intensive methods.

The process can be described in two phases: (i) structural checks that can be applied to entire columns, and (ii) sampling-based selection in case the column was not pruned in the earlier stage. For both these phases it uses statistics gathered earlier in the write process (described in Section 5.3.1). These statistics include cardinality, null count, and max values for numerical columns. For string-based columns, it includes the maximum string length in bytes. For every column a bidirectional map is constructed, which tracks the unique values and their respective counts.

These stages can be bypassed or influenced by the user. If a specific schema is forced, the selection logic is skipped entirely, and the user-defined encodings are applied. If a schema pool is provided, only the sampling-based selection is executed with the provided schema pool, using the encodings from the pool as the candidates. The remainder of this section describes the two stages in more detail.

Structural Selection The structural stage consists of four checks, applied sequentially. Once a check has determined an appropriate encoding for a column, the column is considered "resolved" and is no longer considered in subsequent steps and the sampling stage.

First, a constant check identifies columns containing only a single unique value. This is determined by checking if the bidirectional map has size 1 in case of numbers, or by checking the flag `is_constant` for strings. If so, a variant of `CONSTANT_*` encoding is applied.

Second, an equality check performs a brute-force comparison between all columns of the same data type. If two columns are identical the second column is encoded with `EQUALITY`.

Third, a null check identifies sparsely populated columns. If the percentage of null values in a column exceeds 95% it is encoded with a variant of `NULL_*`. It stores only the defined values and their positions in the binary format.

The final check determines if two columns have a one-to-one mapping. It does this by looping over every column pair. For every pair it compares the distinct-value ratio to a predefined maximum to determine if dictionary encoding can be applied. If so, a bijection test is performed which has specializations to support all column combinations of string and primitive types. The test first performs validity checks by comparing row count and null maps, after which it builds two hash maps modeling the relation in both directions by scanning all rows. When an existing key maps to a different value, or vice versa, it aborts. If this operation completes the relation is bijective, resulting in the second column being encoded with an external dictionary operator.

Sampling-Based Selection When a column could not be pruned in an earlier step FastLanes determines the appropriate encoding through sampling. To reduce the search space, and prevent incompatible encodings being applied, encodings are separated into encoding pools based on their data type. For each type, there is a general purpose pool (e.g., `I64_POOL` for 64-bit integers, containing RLE, FFOR, etc.) and a pool containing dictionary encodings. In addition to the data type, the dictionary pool uses the cardinality of the column to determine the smallest representation to encode indices.

After determining the set of encodings a subset of vectors is selected which will be encoded with all selected encodings. The subset of vectors is selected based on a so-called three-way approach: taking the first, last and middle vector for a given column. The exact set of vectors differs per row group and sample size, for a row group of size 64 the ordered list of vector indices would be $\{0, 63, 32, 16, 48, \cdots, 1, 62\}$. The reasoning behind this approach is that data in a table often exhibits locality, while also having a gradual change in the distribution across a column. All selected encodings are applied to the selected vectors in a column, of which the resulting size is extrapolated to estimate the total encoded size of the column. The expression used is then chosen by comparing all estimated column sizes and selecting the expression associated with the smallest size.

4

DuckDB Extensions

DuckDB offers an extension mechanism for introducing additional functionality for specific use cases. This allows DuckDB to keep a small binary size with no external dependencies, while not constraining the potential applications where and how DuckDB can be applied.

Extensions are dynamically loaded using `dlopen`, returning a handle based on a provided file path (66). This handle is subsequently used in `dlsym`, which retrieves a pointer to a function based on the provided handle and function name (67). Using `dlopen` allows DuckDB to load additional functionality at runtime without the need for further compilation. DuckDB provides the commands `INSTALL` and `LOAD` in the runtime CLI. `INSTALL` fetches a zipped file containing the extension binary, then decompresses the file and stores it in local storage. A repository can optionally be provided with `FROM`; it can be either a direct URL or an alias, and it is only required for non-core extensions. `LOAD` loads the stored file and dynamically links it using the aforementioned functions. An example is given in Figure 2.

```
INSTALL arrow FROM community;
LOAD arrow
```

Listing 2: Installing a community extension in DuckDB.

Template Repository For developing extensions, the DuckDB team has made a template repository available (68). The repository provides a skeleton that can be extended with functionality specific to the extension. It further provides a build system that automatically generates a DuckDB binary with the extension linked.

4.1 Table Function

Table functions in DuckDB produce `DataChunk` objects, representing rows and columns of a table. Table functions are defined through the `TableFunction` class, which is used for both internal and extension scan operators[1]. We cover parts of the interface relevant to the FastLanes extension below.

During the binding phase of the query processing pipeline, DuckDB calls the `bind` function, which sets the names and return types of the table. Upon completion it returns a `FunctionData` object, which can contain arbitrary information required by the specific table function implementation down the line.

After binding, a logical plan is constructed using the `statistics` and `cardinality` functions for optimization. `cardinality` returns the expected number of rows produced by the table function. For numeric columns, `statistics` provides per-column min/max values, distinct count, and null count. For string data, `statistics` keeps track of per-column min/max values, whether Unicode characters are present, and the maximum string length. This data can then be applied to more accurately estimate costs, for example by improving filter selectivity estimates. This in turn leads to better join orders with smaller intermediate results, which reduces the number of rows processed to answer a given query.

Based on the logical plan, the physical planner is initialized, and global initialization is performed by the `init_global` function. It sets up a state shared between threads, and is used to keep track of the progress within the table function. The `init_local` function similarly sets up a tracking state, but is thread-local.

Using the physical plan, `function` is called in the execution phase. This function is responsible for filling `DataChunk` objects with data structured with the same schema as defined during the bind phase. `function` can execute in parallel, and is supported by the global and local states to synchronize between threads. During the execution phase, `table_scan_progress` gives feedback on the percentage completed.

Partitioning When a table is split into multiple files based on partition keys, for example using HIVE, the functions `get_partition_info` and `get_partition_data` can be used to improve performance. When filters are applied to partition keys, files can be skipped based on the filename or directory path, reducing I/O operations. `get_partition_info` determines the partition properties of a set of columns supplied by the query optimizer.

[1] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/include/duckdb/function/table_function.hpp`

This can be one of the following: `NOT_PARTITIONED`: no guarantees can be made on the layout of the columns; `SINGLE_VALUE_PARTITION`: each partition has exactly one unique value combination of the columns; `OVERLAPPING_PARTITIONS`: the partitions overlap at the boundaries; `DISJOINT_PARTITIONS`: the partitions are disjoint but can contain multiple value combinations of the columns. `get_partition_data` gets called during execution to retrieve the column values of the current partition, as these are left out during writing and embedded in the directory path instead[1]. In addition, `get_partition_data` allows the execution pipeline of DuckDB to track progress using `batch_index`. The semantics of `batch_index` are defined by the table function, and thus can represent different granularities such as row groups or files.

Pushdown DuckDB can push down filters into the scan operator of a table function. Depending on the implementation, this can reduce the amount of data read and the materialization cost. During planning, DuckDB tries to push a list of expressions into `pushdown_complex_filter` if it is defined. The table scan function can then consume expressions by removing them from the incoming list[2]. Expressions can be of arbitrary complexity, and can span multiple columns. If the expression list is not empty after executing `pushdown_complex_filter` the optimizer tries to push the remainder as generic expressions into the table function. With `pushdown_expression` the table function can indicate if it supports a generic expression by returning a boolean value. The supported generic expressions are then added as a `TableFilterSet` to the table initialization function input[3]. A generic expression is more limited compared to expressions used by `pushdown_complex_filter`, as the former operates over single-column boolean predicates while the latter uses arbitrary logical subtrees. When a table function supports neither expressions nor generic expressions, or when not all expressions are supported, the optimizer constructs a `LogicalFilter` which applies the expressions on the output of the table function[4]. Filter support has to be explicitly enabled using the `filter_pushdown` flag in the table function.

Next to filters, DuckDB also supports projection pushdown. This behavior is enabled by setting the `projection_pushdown` flag to true. If enabled, DuckDB will produce projection ids that map into the physical column layout. The table function then only populates the columns as indicated by the given projection ids. If it is false, DuckDB requests all

[1] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/parallel/pipeline_executor.cpp#L126-L184`

[2] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/optimizer/pushdown/pushdown_get.cpp`

[3] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/optimizer/filter_combiner.cpp`

[4] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/optimizer/filter_pushdown.cpp`

columns and constructs a logical projection that filters out columns based on the table function output. When combined with the `filter_prune` flag, the resulting projection ids are further filtered for columns that are only required for predicate evaluation and not for the query result [1].

Late Materialization Late materialization first selects a set of rows using only lightweight columns, and only when the row set is known fetches the relevant parts of the remaining columns. When the `late_materialization` flag is enabled, the optimizer will try to rewrite the logical plan into a two-phase fetch[2] joined by a row-id. First it constructs an LHS, which represents a light pass using the minimal amount of columns to evaluate the top operator (e.g. ORDER BY). The LHS is used to determine the set of row-ids used to answer the query. Next the RHS is constructed, which is equivalent to the original logical plan but with a row-id column added. Then a join is performed over the row-id column of the LHS and RHS such that the RHS returns only the rows selected by the LHS. After the join the results are ordered and materialized. In order for the join to work correctly, the table function must implement `get_row_id_columns` and `get_virtual_columns`. As late materialization depends on ordering it needs row-ids that are stable across scans, where row-ids should be uniquely identifiable across files, row groups and other possible partitions processed in the scan. In addition, DuckDB fetches these row-id columns only from the virtual column set. This guarantees that for both the LHS and RHS the row-id column is available regardless of the projections.

4.2 Copy Function

Copy functions provide SQL semantics for interacting with files through tables or table-producing functions. Copy functions are defined through the `CopyFunction` class and support two variations: `COPY FROM` and `COPY TO`, for importing a file into a table and exporting a query result into a file, respectively [3]. We describe the relevant parts of the interface by describing the lifecycle of both variations.

[1] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/include/duckdb/execution/operator/scan/physical_table_scan.hpp`

[2] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/optimizer/late_materialization.cpp`

[3] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/include/duckdb/function/copy_function.hpp`

COPY TO During binding of the copy function, `copy_to_select` can be used to intercept the select list and push new projections on top of the existing plan[1]. The select list is a set of bound column references based on the `SELECT` statement. For example, performing `SELECT * FROM tbl` over a table `tbl` with an id and name column will yield a select list of `[id, name]`. The select list can then be modified based on the copy target and supported types. One of the use cases is casting unsupported types into `VARCHAR` to prevent a potentially lossy conversion. Taking the previous example, with id being an unsupported type, this would result in the modified select list `[CAST(id AS VARCHAR), name]`. After optionally adding projections over the select list, the column names and types are resolved with `copy_to_bind` similar to the table function.

After binding, a physical plan is constructed that takes into account the mode of parallelism. The target mode is indicated by the `execution_mode` function, and can be either `REGULAR_COPY_TO_FILE`, `PARALLEL_COPY_TO_FILE` or `BATCH_COPY_TO_FILE`[2]. When the insertion order is not enforced, `PARALLEL_COPY_TO_FILE` offers the highest throughput by out-of-order encoding and writing of rows. If insertion order is enforced and a batch index is available `BATCH_COPY_TO_FILE` still offers parallelism in encoding but with sequential batch writes. If insertion order is enforced and there is no batch index available, such as with HIVE, `REGULAR_COPY_TO_FILE` will perform a single-threaded sequential write.

Next, `copy_to_initialize_global` and `copy_to_initialize_local` initialize the shared and local thread states. The general model follows encoding parts of a table per thread in a local buffer, subsequently committing to a file by using a writer object stored in a thread-safe data structure in the global state.

Since the execution phase differs based on the parallelism mode, we describe the flow of `PARALLEL_COPY_TO_FILE` here and dedicate a separate paragraph to `BATCH_COPY_TO_FILE` below. `REGULAR_COPY_TO_FILE` follows the same model as `PARALLEL_COPY_TO_FILE` but with only a single thread. At the start of the execution phase, DuckDB instantiates the maximum number of worker threads supported by the source operator. Then each worker will start to pull morsels from the source operator and run it through any intermediate operators defined in the pipeline, finally calling `copy_to_sink`. Workers will keep calling `copy_to_sink` until the `DataChunk` objects produced by the source operator are exhausted. The `copy_to_sink` function is responsible for encoding the incoming `DataChunk` objects, and flushing them once a predefined threshold has been met. When the source is exhausted,

[1] https://github.com/duckdb/duckdb/blob/v1.4-andium/src/planner/binder/statement/bind_copy.cpp#L238-L269

[2] https://github.com/duckdb/duckdb/blob/v1.4-andium/src/execution/physical_plan/plan_copy_to_file.cpp

each worker calls `copy_to_combine`, which should encode and flush the remainder of the last `copy_to_sink` call. After all workers have finished and called `copy_to_combine`, the pipeline finalizes and calls `copy_to_finalize`, providing a window in which to format related metadata and facilitate a graceful shutdown for the writer.

Batching If order should be preserved, `BATCH_COPY_TO_FILE` offers parallelism when the target format supports horizontal partitioning of the table. Instead of the sink, combine and finalize model, batching exposes a new API through `prepare_batch`, `flush_batch` and `desired_batch_size` functions. With `desired_batch_size` an indication can be given on how many rows a batch should contain. During execution, `prepare_batch` runs in parallel to encode `ColumnDataCollections`. These can be viewed as grouped `DataChunks` with the same schema, but with additional features such as being spillable to disk in case of memory exhaustion. After a morsel has run through `prepare_batch`, `flush_batch` is called with an encoded `ColumnDataCollection` according to the current `batch_index`, committing the data to disk[1].

Internally the source of the pipeline will produce a set of `DataChunks` with a shared `batch_index` for each morsel, according to the batching granularity of the source operator. Once the batching logic of the source operator has determined that the current batch is finished, the resulting set of `DataChunks` is handed to the global state[2]. Repartition logic then merges and splits the global batches into new batches that are approximately the size specified by `desired_batch_size`[3]. These repartitioned batches are subsequently picked up by workers, which can encode to the target format in parallel and possibly out of order. During `flush_batch` order is enforced again, using the monotonically increasing `batch_index` from the repartitioning step. As batching heavily relies on the `batch_index`, it is required that the source operator implements `get_partition_data`, such that batch boundaries and the current batch are defined.

File Rotation To control file size and use parallelization across files, DuckDB offers the option to rotate the current file using the `rotate_files` and `rotate_next_file` functions. `rotate_files` acts as an indicator of whether file rotation should be supported by returning a boolean. If enabled, `rotate_next_file` will be called before each sink or combine

[1] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/execution/operator/persistent/physical_batch_copy_to_file.cpp#L367-L386`

[2] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/parallel/pipeline_executor.cpp#L126-L184`

[3] `https://github.com/duckdb/duckdb/blob/v1.4-andium/src/execution/operator/persistent/physical_batch_copy_to_file.cpp#L411-L513`

function call to determine if a threshold, as defined by the copy function, has been met[1]. When a thread hits the threshold it will swap the global file lock with a new file lock for the next file to write. For the previous global file lock the thread waits until it can get an exclusive lock on the file, after which it calls the finalize function. When a thread is awaiting the exclusive file lock, only threads already having a shared lock on the file can commit their results, releasing their shared locks afterwards. The setup is non-blocking, as threads not owning a shared lock can continue to acquire a shared file lock for the newly assigned global lock. Currently file rotation is not supported in combination with order preservation and `BATCH_COPY_TO_FILE`[2], requiring manual management of files within the copy function.

COPY FROM The copy-from function inserts data from an external source into a predefined table. When defining a query as in Listing 3 with the target table `people` having columns `id`, `name` and `createdAt`, DuckDB will rewrite this to the query given in Listing 4. When the source table has more columns than required by the target schema, a subset of the columns is selected in the `SELECT` statement during rewrite[3]. During binding, `copy_from_bind` is invoked with the target table's expected names and types. It is responsible for resolving the file schema, mapping/ordering columns and preparing cast expressions where source and target types differ. At execution time, the table function referenced by `copy_from_function` produces `DataChunks` in accordance with the behavior described in Section 4.1.

```
COPY people(id, name)
FROM read_parquet('s3://bucket/people.parquet')
```

Listing 3: Query using the copy from function.

```
INSERT INTO people(id, name)
SELECT id, name
FROM read_parquet('s3://bucket/people.parquet');
```

Listing 4: Rewriting result of Listing 3.

[1] https://github.com/duckdb/duckdb/blob/v1.4-andium/src/execution/operator/persistent/physical_copy_to_file.cpp#L497-L535

[2] https://github.com/duckdb/duckdb/blob/v1.4-andium/src/execution/physical_plan/plan_copy_to_file.cpp#L27C1-L35C3

[3] https://github.com/duckdb/duckdb/blob/v1.4-andium/src/planner/binder/statement/bind_copy.cpp#L353-L408

4.3 MultiFileInfo

Aside from the ability to hook into DuckDB directly with the table and copy function interfaces, DuckDB provides an abstraction over these interfaces that provides a standardized implementation for certain optimizations and behaviors, such as projection pushdown and managing multiple files. This abstraction is contained in the templated `MultiFileFunction` class, which extensions interact with by implementing the `MultiFileReaderInterface` struct. Benefits of basing an extension on this interface are standardization, a reduction in complexity of custom logic, and the ability for further extensions to more easily reuse and reason about existing extensions. The latter is relevant in case of table formats, like Iceberg[1], Delta Lake[2] and DuckLake[3], which are able to reuse significant parts of extensions using the `MultiFileFunction` abstraction, easing integration of new columnar formats into existing table formats. In the remainder of this section we first cover the general architecture of the `MultiFileFunction` class, followed by a lifecycle description, finishing with details on partitioning, optimizations, and user feedback. For clarity, we indicate with the colors teal and orange functions that should be implemented by the extension in question. Teal denotes `MultiFileReaderInterface` functions, orange denotes functions from `BaseFileReader`.

Overview The `MultiFileFunction` templated class wraps a normal DuckDB table function, acting as the glue between the DuckDB function catalog, the format-specific extension code, and multi-file management. Format-specific behavior is supplied through the `MultiFileReaderInterface` struct. The implemented virtual methods are called by `MultiFileFunction` and model the lifecycle of a format-specific table function. The `MultiFileReader`, instantiated by `MultiFileFunction`, interacts with the files by creating `BaseFileReader` instances that produce `DataChunks`.

Lifecycle During binding, `MultiFileFunction` constructs a generic `MultiFileReader` and `MultiFileReaderInterface` instance via `InitializeInterface`. The reader then creates a file list from the input, which can lazily expand based on glob patterns. Both the generic multi-file options and the format options are instantiated (`InitializeOptions`) and populated: generic options are handled by `MultiFileReader` (e.g., HIVE configuration), and format-specific options are handled by `ParseOption`. Next, the bind state is

[1] https://iceberg.apache.org/
[2] https://delta.io/
[3] https://ducklake.select/

initialized (InitializeBindData) and a determination is made about whether schema information should be fetched from the first file (BindReader) or through a custom bind call (used by table formats). By default schema derivation will always happen on the first file, requiring a custom implementation of `MultiFileReader` injected in the `MultiFileFunction` through the `get_multi_file_reader` table function property otherwise. After the schema and options are fixed, FinalizeBindData can synchronize reader derived information, such as row group size, back into bind data.

During logical optimization the planner will try to consume expressions by invoking `pushdown_complex_filter` and applying them to the file list based on HIVE partitioning, returning a filtered file list. If there is no HIVE partitioning this process is skipped.

At execution start, the file list is filtered again using canonical table filters coming into the global initialization function. These may be dynamic filters which would not have been available during previous phases[1] (e.g. parameters which are now bound). It then instantiates a list of reader slots. When using a table format, pre-seeded readers are copied into the list, otherwise it uses the initial reader used during bind. Any further readers are created lazily based on worker request. Next, the format's global state is created with InitializeGlobalState, and the scan thread budget is set to the scheduler pool size, capped by the format readers MaxThreads. Finally, the returned projection is determined and any additional virtual columns required by the reader are added.

Each worker then initializes a thread-local state via InitializeLocalState. The worker briefly acquires the global lock of `MultiFileFunction` and enters an infinite loop. At every iteration the worker checks if there are any files left to read by comparing the value `file_index` to the length of `readers`. `file_index` starts at zero and advances only when the current `file_index` is not able to give out more batches. As such, when `file_index` < `readers` there is still work left to be picked up for the worker. The `file_index` is used as index into the `readers` list to access the global reader for the associated file. When the file is open, the worker tries to initialize a local reader for the next batch with TryInitializeScan. If there are no more batches left in the current file the `file_index` is incremented and the file is closed (FinishFile). Otherwise, the worker has succeeded in claiming a batch. If the worker has sourced a batch from a different file compared to the previous call, it will instantiate the schema that the reader will output before finalization. If the file is not yet open, the worker tries to open the file by calling CreateReader, subsequently initializing the global reader for the associated

[1] https://github.com/duckdb/duckdb/blob/v1.4-andium/src/execution/operator/scan/physical_table_scan.cpp#L28-L30

file (InitializeReader), and if the file should not be skipped, does optional preparations (PrepareReader). If the current file's batches are exhausted and there are no more files to read, FinishReading is invoked before exiting the loop.

After acquiring a batch, the worker starts filling DataChunks up to the standard DuckDB vector size (2048) by calling Scan on each MultiFileScan invocation. The scan produces an intermediate chunk, which is finalized into the target schema by applying expressions over the chunk. These expressions are responsible for a range of functions, such as column reordering, adding virtual columns and computing derived fields[1]. When the intermediate chunk is empty (i.e. it produced no rows) after a scan, the batch is considered completed and the worker will request the next batch as outlined previously.

Partitioning A batch_index is maintained in the global table function state, managed internally by MultiFileInfo. Every time a worker gets assigned a batch by succeeding in TryInitializeScan, the batch_index is incremented. This increment is secured behind a global lock, workers first exhaust the available batches in the opened file, only after advancing to batches in the next file by increasing file_idx. Thus, the batch_index is guaranteed to be monotonically increasing with order preserved across files and batches. During execution, MultiFileGetPartitionInfo determines the guarantees of the partition schema based on a set of column ids, if SINGLE_VALUE_PARTITION cannot be guaranteed the function falls back to no partition. Through MultiFileGetPartitionData, the pipeline executor receives both the assigned batch_index and the constant values of the partition columns identified by their column ids.

Optimizations MultiFileScanStats gathers statistics by delegating to GetStatistics, which is called only when both the file list contains a single file and an initial reader is defined. It is disabled for multiple files to prevent the opening of many files, of which the cost is further exacerbated in the case of a table format due to networked storage. Cardinality estimation in MultiFileCardinality is first tried based on the file list, however, the only implementation currently returns a nullptr, so it always asks the format plug-in through GetCardinality for a cardinality estimation based on the bind data and file count.

User Feedback During execution, MultiFileProgress keeps track of the total progress of the scan, returning a percentage of the total work completed. This percentage is calculated by adding 100% for every completed file, and getting a percentage through

[1] https://github.com/duckdb/duckdb/blob/v1.4-andium/src/common/enums/expression_type.cpp

GetProgressInFile for every file being worked on. The total is then divided by the total number of files to be scanned. To aid in debugging, MultiFileFunction provides two utility functions. These functions output information when configuring the CLI with PRAGMA enable_profiling;, or when using EXPLAIN in front of the query in question. First, MultiFileGetBindInfo provides information on the file paths used in the scan, with additional format-specific information added through GetBindInfo. Second, with MultiFileDynamicToString the user receives feedback on how many files the scan accessed. Note that information passed through get_bind_info of the table function does not reach the user-facing output, except for catalog information of the function in question, making any information fetched from GetBindInfo invisible[1].

[1] https://github.com/duckdb/duckdb/blob/v1.4-andium/src/planner/operator/logical_get.cpp#L27-L32

5

FastLanes Extension

5.1 Introduction

This chapter marks the transition from background and format analysis to the main contribution of this thesis. We translate the concepts introduced in the previous chapters into the architecture and implementation of a FastLanes extension for DuckDB. The central question is not only how FastLanes files can be made readable and writable from DuckDB, but also how this can be done in a way that fits DuckDB's execution model while introducing as little overhead as possible. This directly addresses **RQ1**.

A second design consideration is where the boundary between DuckDB and the FastLanes library should be placed. In some cases, the existing FastLanes API is sufficient and can be wrapped directly by DuckDB-facing code. In other cases, however, the API is too limited to support efficient integration. More strongly, we find that the pre-existing API does not provide the abstractions required to implement writing support inside DuckDB. This chapter therefore also addresses **RQ1.1** and **RQ1.2**: whether the current FastLanes library API is sufficient for efficient integration, and which extensions or modifications are required or desirable.

The implementation described in this chapter focuses on the interfaces that are essential to make FastLanes usable within DuckDB. These are the table-function path for reading, the copy-function path for writing, and a set of targeted patches to the FastLanes library that improve compatibility and performance. Across these changes, we aim to minimize modifications to the FastLanes library itself, as larger deviations reduce the likelihood of successful upstream integration in the future. Accordingly, several additional integration points were considered but left out of scope for this work, including DuckDB's `FileSystem` API, memory allocation through DuckDB's buffer manager, and tighter coupling to DuckDB's threading abstractions.

The remainder of this chapter is structured as follows. We first describe how FastLanes is connected to DuckDB's table-function interface for reading. We then describe how FastLanes is integrated into DuckDB's copy path for writing. Finally, we discuss the patches applied to FastLanes, distinguishing between changes that are required for correctness and changes that primarily improve performance. The extension code is open source and available on GitHub[1].

5.2 Reader

5.2.1 Configuration

The reader can be invoked using `read_fls('path')` in a query. The path value supports globs (`'/**/*'`), but should only ever select FastLanes files (`'/**/*.fls'`). Currently two configuration options are supported: `explicit_cardinality`, which allows the user or a table format to explicitly set the number of rows in the files to be read. The `file_row_number` flag constructs a virtual column that provides a unique identity for rows in a file; when combined with the file index, it provides a unique identity across files. Options are added by expanding the `named_parameters` object using `string` keys and setting `LogicalType` values, exposing the option through `read_fls('path', key=value)`. Each option is parsed in `ParseOption` by comparing against the incoming key value and casting the associated value to its `LogicalType`, inserting each result in `FastLanesFileReaderOptions`.

Parallelization Parallelization is determined in `MaxThreads`. It returns the number `initial_file_n_rowgroups` if we use an initial reader, otherwise the table function becomes single-threaded. If there are multiple files, it returns the maximum number of threads supported by the hardware. The latter is done to prevent a low parallelization in case the first file happens to be disproportionately small compared to the other files.

5.2.2 Binding

Upon entering `InitializeBindData` we move `FastLanesFileReaderOptions` into a new struct `FastLanesReadBindData` so that the options stay available during the scan. If `explicit_cardinality` is set we further calculate the `initial_file_cardinality` by dividing by the file count. The resulting `FastLanesReadBindData` is then passed to the constructor of `FastLanesReader` in `BindReader`. In case we have an initial reader we use this to further set the `initial_file_cardinality` and `initial_file_n_rowgroups` in

[1] `https://github.com/sebastiaan-dev/duckdb-fastlanes`

`FinalizeBindData`. At the end of the binding phase a file-scoped instance of `FastLanesReader` is created. In its constructor we derive a `TableMetadata` object from the target file path, wrapping file metadata in getter methods usable by the FastLanes extension.

Type Promotion The metadata is passed to a `SchemaBuilder`, which, upon being built, tries to resolve the physical types of each column across the row groups in the file. The physical representation for the same column can differ between row groups, while the destination vectors in DuckDB only support a single physical representation per row group. As such, we try to resolve any discrepancies using type promotion. Each type is grouped based on the casting support in the decoding kernels. For example, `uint16` can be safely cast to `int32` but not to `uint8`. Within each group every type is given a relative rank based on the width of its physical representation. Type promotion then selects the smallest type that can safely serve as a common decoding target for all physical representations of the column. If no safe promotion exists the scan fails with a runtime error. The final promoted types are then mapped to their DuckDB equivalents and stored alongside the column names in the reader.

Statistics As a final step in the constructor of `FastLanesReader` the row group statistics are gathered in `RowGroupStatistics`. Currently per column min and max values are supported. These values are stored as type-erased binary blobs, and are cast back into their respective types using the physical representation of the column.

5.2.3 Scanning

Initialization First, in `TryInitializeScan` workers check if there is any work to be done in the current file by comparing `next_rowgroup`, representing the consumed row groups plus one, to the size of the row-group indices to be scanned. If `next_rowgroup` is still a valid index in `rowgroups_to_scan` the worker consumes the row group, otherwise the current file has no work left.

In case the row group can be consumed, the worker requests a `RowGroupReader` from `FastLanesReader`. Then, `cur_vector` is set to zero, keeping track of the progress in the current row group, and is used to request chunks from the `RowGroupReader`.

If there are filters, each filter will be referenced by `FastLanesScanFilter` in the `scan_filters` vector. The `FastLanesScanFilter` acts as a wrapper over the filters provided by DuckDB, which may contain additional cross-vector state. In addition, an adaptive filter is constructed which can dynamically reorder the present filters based on performance. Finally,

a `filters_by_col` vector containing vectors of pointers to `FastLanesScanFilter` is constructed, giving each decoding kernel access to the filters present for the respective column.

The last step in initialization constructs a `ColumnDecoder` for each projected column (the `column_ids` property). In case the decoders already exist, which happens when a worker receives the next batch from the same file, they are reset to prevent conflicting states across row groups. Then, we initialize each `ColumnDecoder` by calling its `Init` function. Using the operator variant for the respective column, as determined by FastLanes, cross-vector state is set up.

Producing Chunks After initialization, the extension enters an infinite loop in `Scan`. The loop is exited either when there are no more tuples left in the row group, or when there is at least one tuple present in the `DataChunk` at the end of the loop. The purpose of the infinite loop is to prevent returning a `DataChunk` with a cardinality of zero, while there are still more tuples left in the row group, causing an early exit from the batch. This can occur due to the filtering step finding no matches in the current set of tuples.

Inside the loop we first determine how much work there is left to do. FastLanes uses a vector size of 1024, whereas DuckDB defines vectors with a default size of 2048 elements. To fill a DuckDB vector as much as possible, we determine if we can pull two yet to be processed vectors from the FastLanes file, which are then written to the same DuckDB vector at their appropriate offsets (0 and 1024). If we do not fill the vectors up to the default size, it results in a combination of: (i) increased physical operator executions, and (ii) `DataChunk` merges performed by DuckDB. Both of these being detrimental for performance.

For each FastLanes vector that can be fitted, we fetch all vectors of the projected columns that reside at the same index in the row group through `get_chunk`. Then, we iterate over the set of projected columns to index into the `DataChunk` and `expressions` vector. The former giving the destination vector and the latter giving the source expressions for the respective column. The index of the `DataChunk` and `expressions` vector both map to the same logical column, due to a mapping operation performed when creating a new row group reader. Next, we decode into the current DuckDB vector using the `Decode<Pass>` function from `ColumnDecoder`. The decoder function is templated so that one implementation can handle multiple situations, without runtime branching on offset logic.

5.2.4 Decoding

To decode a FastLanes vector into a DuckDB vector we use a separately instantiated thread-local `ColumnDecoder` for each column. A `ColumnDecoder` delegates the incoming decode request to a `KernelDecodeVisitor<Pass>`, injecting column-scoped state with `ColumnCtxHandle`. The visitor unwraps the operator variant, calling the appropriate `Decode` implementation of `KernelTraits`. Each decode kernel provides a typed implementation of the interface shown in Listing 5. When a `ColumnDecoder` initializes it delegates to `KernelInitVisitor`, which, similar to the earlier visitor, unwraps the operator variant, calling the appropriate `Prepare` implementation.

```
template <typename OpT>
struct KernelTraits {
        static void Prepare(ColumnCtxHandle&,
        LogicalType&,
        OpT&,
        const std::vector<FastLanesScanFilter*>*) {
                throw std::runtime_error("Not implemented");
            }

        template <Pass PASS>
        static void Decode(ColumnCtxHandle&, Vector&, idx_t, OpT&) {
                throw std::runtime_error("Not implemented");
            }
    };
```

Listing 5: The generic interface for each decode kernel.

Column State The `ColumnCtxHandle` provides a type-erased store for column-scoped state. Using `Emplace<ColumnCtxBase>()` in `Prepare` constructs a typed object for the respective column in the current row group. During the `Decode` call of the same column the state can be accessed through `Expect<ColumnCtxBase>()`. The inserted object can be an arbitrary type, but must inherit from `ColumnCtxBase`.

Casting As stated before, physical representations for the same column can differ across row groups. To ensure values are accessed and interpreted correctly we provide visitors for the common operations: copy, assign and untranspose. Each visitor provides a fast path for when the values can be directly copied, or a slower path in case casting is required. Note that when unsigned integers coming from FastLanes need to be copied to a signed destination of the same size, we can do so without a cast. As the discrepancy is a result of

optimization within FastLanes, it can never happen that the value of the unsigned integer exceeds what can be represented with a signed integer.

5.2.5 Filtering

The reader supports three granularities of filtering: file, row group and row level. File level filtering is supported implicitly by using `MultiFileReaderInterface`. We thus cover only the latter two, as these require a custom implementation.

Chunk Filtering After filling a `DataChunk` for all projected columns the `FilterExecutor` performs a filtering step, if filters are present. It does so by looping over a range, bounded by the amount of filters. Using the current index of the iteration we request a filter from the adaptive filter. The result is a `FastLanesScanFilter`, containing a `filter_index`, `TableFilter` reference, `TableFilterState` and `FilterCtxHandle`. Corresponding to the location in the output `DataChunk`, immutable information on the filter such as its type, and the compiled predicate with scratch buffers respectively. The `FilterCtxHandle` provides a type-erased metadata store, which provides a link between decoders and filter execution. The target vector of a filter in the `DataChunk` is wrapped in a unified vector format, which provides a singular interface to interact with different DuckDB vector types. The DuckDB provided function `FilterSelection` then applies the filter on the vector using the normalized view, assigning the surviving rows to a selection vector. In case the filter resulted in changes, the `DataChunk` is modified using the selection vector to represent only the surviving rows across all columns.

Since DuckDB does not have knowledge on encoding specifics from the decoding kernels, introducing an encoding aware fast path which interprets the filter manually may improve performance. As such, each decode kernel has access to the set of filters applied to the respective column. A decode kernel can choose to add additional metadata alongside a filter with `Emplace<FilterCtxBase>()`, with the actual type being a struct that inherits from `FilterCtxBase`, such as `FSSTDictFilterCtx`. The base class enforces an enum property `DirectFilterKind`, used during filter execution to downcast to the internal type. When a filter contains a valid instance of `FilterCtxBase`, the fast path is used, skipping the earlier described `FilterSelection` by exiting the respective iteration early.

We decided to implement this optimization in the extension rather than in the FastLanes library itself, because the current FastLanes library does not yet expose a predicate-pushdown interface that DuckDB can invoke directly. This makes the extension responsible for applying DuckDB's `TableFilters` directly to FastLanes vectors. In the longer term, a

more appropriate design would be to move predicate evaluation into the FastLanes library itself. Besides yielding a cleaner architectural boundary, this would also allow predicate pushdown to be implemented across cascaded encodings, whereas the current extension-level optimization can only operate on the final representation in the decoding chain. This direction is consistent with the work of Dunamalijevs, who studies predicate pushdown in FastLanes itself and shows that FastLanes-native predicate evaluation is feasible, including on compressed representations such as `DICT` and `DICT + RLE` (69).

Row Group Filtering We introduce a `RowGroupFilter` owned by `FastLanesReader`. When a worker initializes a scan, it first requests the list of row group indices to scan. If the filter has already computed this list, it returns the cached result. If it has not been computed before, which is true only on the first scan initialization for a file, the list is built.

Using the statistics gathered in `RowGroupStatistics`, we iterate over all row groups. For each row group we evaluate every filter; the only filter type currently supported is `CONSTANT_COMPARISON`. If a given comparison is compatible with the available column statistics, we evaluate it at row group scope. A row group is skipped when at least one filter can indicate that the range of values in the row group does not intersect the query. The resulting list is subsequently stored for all future requests on the same file.

5.3 Writer

In this section we describe the implementation of the FastLanes writer. We start with a description of the current flow of writing a FastLanes file using the default library code. This is followed up with a description of the modifications made to FastLanes itself. Finally, we cover the extension code implementing the copy function. As the copy function API in DuckDB does not have similar auxiliary structures as the table function API with `MultiFileInfo`, we implement the copy function API directly.

5.3.1 Original FastLanes Writer

The FastLanes writing capability operates through the `Connection` class, which provides the externally callable API and holds configuration variables that may change the behavior of the writer, such as if the footer should be inlined or put into a separate file. We identify the following phases in encoding a source file into a FastLanes file: loading a source file, preparing the in-memory table, encoding selection and encoding.

5.3.1.1 Loading a Source File

The writer supports two types of source files, `csv` and `json`, through the functions `read_csv` and `read_json` respectively. When either function is called, FastLanes starts the construction of an in-memory table. First it locates the source and schema file on disk, the schema is a `json`-formatted file that contains name and type information for each column. Then, it uses a file parser and streams the resulting tuples into a newly constructed in-memory row group. Each value of the tuple is iterated over, and ingested using `Attribute::Ingest`, after which a tuple count tracker is incremented. When the tuple count reaches the capacity of the row group, which is set to 64 vectors of 1024 values, the row group is moved into the table object and a new local row group is instantiated.

After ingesting all tuples a final check is made to see if the last row group has been pushed into the table. If this is not the case it means that the default capacity was not reached, and the last value in the row group is duplicated and added to the row group until the tuple count is divisible by 1024, after which it is also pushed into the table object.

Instantiating a Row Group When a row group is instantiated it uses the supplied metadata to initialize each column in memory, based on the `DataType`. The instantiated columns are then moved into a type-erased vector, which can later be accessed again using an `std::visitor`.

Ingest The `Attribute::Ingest` function takes a column reference from the type-erased vector and selects between two different strategies based on the column type: `TypedIngest` for numerical columns, and `FLSStringIngest` for string-based columns.

The `TypedIngest` strategy detects if the incoming string values represent `NULL` values, tracking the location in an array and incrementing a null counter. When a value is valid it is cast into the target physical type, if the value is `NULL` the last valid value is used instead. Values are never left uninitialized so that encoders see full vectors. The materialized value is then appended to the column's data vector and associated per-column statistics are updated, which are later used to select encodings.

The `FLSStringIngest` strategy mirrors that logic for variable-length data: it records null values, appends the current prefix sum to an offset array, copies the raw bytes into a shared buffer while updating the length array, and tracks the maximum bytes-per-value plus whether every entry is numeric. At the same time it feeds buffers that later drive FSST compression, and uses the "last seen" string for NULL slots.

5.3.1.2 Preparing the Table

Table preparation encompasses a four-phase, per-row-group pipeline: `Init`, `Cast`, `Finalize`, and `GetStatistics`.

The `Init` phase does housekeeping by setting the indexes of the `ColumnDescriptor`s based on the in-memory column order, so that later routines (i.e `Cast`, `Wizard`) can look up columns by index.

The `Cast` phase is only executed when the user has not forced a schema or schema pool (used to predefine the encodings used). It revisits every `ColumnDescriptor` and checks if the current physical type should be transformed to another physical type which can more optimally represent the same data. An example is the `is_numeric` flag which can be set by `FLSStringIngest`, indicating a string column can be converted to a primitive numerical type (such as int, uint, etc.). After all potential transformations are determined each column with the `should_be_cast` flag set will produce a new in-memory column with the target type.

The `Finalize` phase gathers statistics for the numerical columns such as min, max and creates a bidirectional map acting as a dictionary. For the string column it creates pointer arrays based on the byte and length arrays. Gathering statistics up front and creating views over buffers prevent needing to do these computations, and thus re-scanning the same data over and over, during encoding selection.

The `GetStatistics` phase is a final statistics gathering pass over all columns. Currently it only considers string columns, checking if all values are equal (and thus the entire column being equivalent to a constant) and building a dictionary for string values.

5.3.1.3 Wizard

The Wizard is responsible for determining the optimal encoding for each column on a per-row group basis. This process has been described in detail in Section 3.3.5.

5.3.1.4 Encoding

After all encodings have been determined `Encoder::encode` encodes all row groups, flushing the results to disk. First it creates a buffer and a file handle, subsequently looping over all row groups. In each iteration the RPN stack is fetched for each column from the `ColumnDescriptor`. Following, the RPN stack is translated to a `PhysicalExpr` and subsequently executed on a per-vector basis. A more detailed description with examples on expression encoding is present in Section 3.3.2. After all vectors for a column have finished

encoding they are flushed in order to the local buffer. The local buffer is flushed to disk when all columns for the respective row group have finished encoding. The resulting buffer sizes are tracked between flushes and referenced in the metadata, so a reader can determine the location of each row group and column.

5.3.1.5 Limitations

The current FastLanes writer has limitations when it comes to integrating it into a copy function in DuckDB. For starters, the only source types supported are `csv` and `json` files, this differs significantly from the `DataChunks` DuckDB produces. Second, `Attribute::Ingest` accepts only string values, which means every value would have to be serialized before FastLanes parses it back to its original type. Third, the entire source file is loaded as a table in memory, and only afterwards is encoded and flushed to disk. This puts significant pressure on memory, and may lead to out-of-memory (OOM) exceptions or the use of swap. Fourth, there is no way to seed the `ColumnDescriptor`s with the logical types from the DuckDB schema. Fifth, encoding is fully single-threaded, so we cannot use the parallel modes available in the DuckDB copy function. Based on these limitations we modify the existing writer implementation, which is described in Section 5.3.2.

5.3.2 Modified FastLanes Writer

In this section we introduce the modifications made to the FastLanes writer. We describe only the most significant modifications for brevity.

5.3.2.1 Streaming Interface

The original FastLanes writer is only capable of sourcing data from external files. To align the interface with the copy function interface from DuckDB we introduce a streaming-oriented API built around two new classes, `FileWriter` and `RowGroupWriter`. A `FileWriter` is created through a builder that specifies the schema, target path, vector and row-group sizes, optional forced operator pools, and whether the footer is inlined. When `FileWriter::Open` is called it allocates a `TableDescriptorT`, opens a handle to the destination file (and optionally to the footer file) once, writes the file header, and instantiates an offset tracker so subsequent row groups flushes are written to the correct location in the destination file. Each row group is then produced by a `RowGroupWriter`. The writer can either wrap an existing in-memory `Rowgroup`, which stays compatible with the existing `read_x` interface, or it can construct one incrementally via `WriteColumn`. The caller can signal the end of a row group by calling `RowGroupWriter::Finalize`, which ensures the tuple count

is divisible by the FastLanes vector size (1024) and invokes the same path as Sections 5.3.1.2, 5.3.1.3, 5.3.1.4. With the only difference being that they are modified to work on a row group scope, and where encoding modifies a buffer owned by `RowGroupWriter` and does not immediately flush to disk. The resulting buffer plus the freshly produced `RowgroupDescriptorT` is then handed back to `FileWriter::FlushRowGroup`, which appends the bytes at the tracked offset, advances the cursor, and stores the descriptor in the table descriptor in memory. Finally, when all row groups are flushed, `FileWriter::Close` writes the table descriptor to disk and emits the footer.

5.3.2.2 ColumnWriteView

To stream arbitrary data into FastLanes we introduce a `ColumnWriteView` interface, which provides an abstract façade over a span of values that matches a single physical column. Two concrete implementations exist today. `PrimitiveWriteView<T>` wraps a `span<const T>` for fixed-width types and reports its length via `Size()`. The `StringWriteView` does the same for `str_pt` spans. Both defer to a `ColumnWriteViewVisitor` that knows how to ingest data into the corresponding `col_pt`: primitives call `Ingest::TypedIngest<T>` on the `TypedCol<T>` held inside the row group, while strings forward to `Ingest::FLSStringIngest`. `RowGroupWriter::WriteVector` accepts a `ColumnWriteView` plus a column index, fetches the `col_pt` from the active row group, invokes `Accept`, performing a double dispatch between the view and the destination column. This is necessary as `col_pt` is a `std::variant` and therefore the writer cannot know at compile time which physical type it will hold. Afterward, the tuple counter is incremented by `view.Size()`. As the `ColumnWriteView` works with spans, DuckDB's vectors can be wrapped without copying.

5.3.3 Extension

With the modifications to the FastLanes writer covered, we are now ready to describe the extension implementation.

5.3.3.1 Configuration

The writer can be invoked using an SQL statement equivalent to `COPY table TO 'file.fls';` with `(FORMAT fls)`. The extension supports three configurations: `row_group_size`, which denotes the maximum tuple count per row group, this value should be divisible by 1024. `row_groups_per_file`, which specifies the desired number of row groups per file. `inline_footer`, which determines if the FlatBuffer metadata will be inlined, or put in a separate file.

Execution Mode As the copy function supports three different execution modes we define a `GetExecutionMode` function acting as a mode selector. The extension supports all modes offered by DuckDB and therefore functions as described in Section 4.2.

5.3.3.2 Binding

The binding phase is relatively simple, as it is only responsible for parsing the aforementioned configuration options and passing the schema information into the `FastLanesWriteBindData` bind state.

5.3.3.3 Global Initialization

Global initialization takes the columns from the bind state and parses them into FastLanes compatible `DataType`s, constructing a `ColumnDescriptor` with name and type for each column. We note that `DataType::STR` contains a logic bug where `TypedStats` initializes a `nullptr`, and it should therefore not be used. String columns should be defined with `DataType::FLS_STR` instead. When the incoming DuckDB type is a decimal, we additionally construct a `DecimalTypeT` object, referenced by the `ColumnDescriptor`, containing the precision and scale.

We then instantiate a `FileWriter` using the builder pattern, passing the constructed `ColumnDescriptor`s and configuration options. Finally instantiating the initial state by calling `FileWriter::Open`.

5.3.3.4 Local Initialization

For the Sink and Combine path each worker thread owns a `FastLanesWriteLocalState` with an empty `ColumnDataCollection` and append state, initialized once to match the bound column types. Incoming `DataChunk`s are appended into this buffer until it reaches the configured `row_group_size`, at which point the Sink logic in Section 5.3.3.5 hands the entire collection to a `RowGroupWriter`. Batch mode skips the use of this local buffer entirely as `PrepareBatch` receives an already configured `ColumnDataCollection` as function parameter.

5.3.3.5 Copy Sink-Combine

The `Sink` accumulates `DataChunk`s streamed into the function, which can be run in parallel by either a single or multiple threads depending on the execution mode. Each worker appends its incoming chunk to the thread-local `ColumnDataCollection` until the total tuple count reaches the bound `row_group_size`. If the incoming chunk exactly fits the remaining

space, the thread creates a new `RowGroupWriter`, calls `PrepareRowGroup` wrapping the accumulated vectors in a `ColumnWriteView`, then invoking `RowGroupWriter::Flush` which commits the row group to disk. In case the incoming chunk does not fit in the remaining bound, it is split so that the current row group can be finished with the exact bound. Then it creates and populates a new thread-local buffer with the remainder.

In parallel execution, several workers may try to commit a completed row group at the same time. As such, `FileWriter::FlushRowGroup` has a mutex guard which ensures only one thread can flush a row group at a time through the same `FileWriter` instance. This mutex serializes flushes through the same `FileWriter` instance, meaning concurrent writes targeting the same output file. Row group preparation can still proceed in parallel, only the actual commits to disk are synchronized. A single `FileWriter` writes to one target file, while multi-file writing is achieved by creating multiple `FileWriter` instances, for example when file rotation is enabled.

Once the source is exhausted, workers call `Combine`, handling the case where `DataChunks` were accumulated but never committed to disk, because they did not reach the bound. As `Combine` synchronizes multiple threads it is guarded by a mutex so that shared buffers are not mutated concurrently. If the thread-local buffer contains tuples, we first check whether a thread-global `combine_buffer` already exists. If not we create this buffer and simply move the thread-local buffer into this shared buffer. If there is a shared buffer we distinguish three cases: (i) the local buffer fits in the shared buffer, so we move the local data into the shared buffer. (ii) the local buffer and the shared buffer total the bound, so we move the local data into the shared buffer and flush to disk. (iii) the local buffer does not fit in the shared buffer, then we try to fill the combine buffer as much as possible, flush it to disk, and then we create a new shared buffer which we move the local data into. The second case is the equality boundary between the first and third cases, but it is handled separately because it requires no splitting and makes the control flow more explicit.

After all threads have executed `Combine`, the pipeline cleans up by calling `Finalize`. This step checks if there is anything left in the `combine_buffer` and flushes this as a final, potentially incomplete, row group if so. Finally it calls `FileWriter::Close`, writing the final metadata.

5.3.3.6 Copy Batch

In batch execution mode multiple workers run `PrepareBatch` in parallel. The `PrepareBatch` function is passed a pre-allocated `ColumnDataCollection` containing approximately the bound amount of rows. Each worker then creates a row group writer, feeding the provided

collection into `PrepareRowGroup`. The `RowGroupWriter` is then returned so that it can later be picked up by `FlushBatch`. The `FlushBatch` function simply calls `RowGroupWriter::Flush`, committing the row group to disk. As batch mode enforces ordering, it achieves parallelism by encoding concurrently and out of order with `PrepareBatch`, then committing to disk in order with `FlushBatch`.

5.3.3.7 PrepareRowGroup

The `PrepareRowGroup` is a function that defines common operations that have to be performed every time a buffer is ready to be encoded as a row group. First, the function pre-allocates a `ViewWriterFactory` for each output column based on its physical type, which produces either a `PrimitiveWriteView`, `StringWriteView` or specialization for more complex types. We highlight one of the factories in Listing 6, where we can see how it wraps a DuckDB vector in a span that can be passed to the FastLanes writer. After constructing factories for each column, the function iterates over all chunks in the `ColumnDataCollection`, building the views on a per-vector basis, passing it to the `RowGroupWriter`. After all chunks have been passed on, the `RowGroupWriter` finalizes, finishing encoding of the respective row group.

```
template <typename PT>
struct PrimitiveViewWriterFactory final : ViewWriterFactoryBase {
        public:
        unique_ptr<ColumnWriteView> Build(Vector& src, idx_t count) override {
                const auto data_ptr = FlatVector::GetData<PT>(src);
                return make_uniq<PrimitiveWriteView<PT>>(
                std::span<const PT> {data_ptr, count}
                );
            }
    };
```

Listing 6: A factory for producing a `PrimitiveWriteView`, templated so it can be instantiated with any fixed width type.

5.3.3.8 File Rotation

When the `row_groups_per_file` option is set the function `RotateFiles` returns true, activating the `RotateNextFile` function. This function simply tracks how many row groups have been committed to disk by checking the shared variable `num_row_groups`, which is incremented after every flush. When this value is equal to or exceeds `row_groups_per_file` it returns true, prompting DuckDB to close the current file and then reopen the writer on a new path.

5.4 Patches

This section describes the patches applied to the FastLanes code. Some patches are essential for the correct functioning of the FastLanes extension, while others aim to improve performance. For each patch, we outline the underlying problem or challenge, discuss possible solutions, and present the implementation of the chosen approach.

5.4.1 Disabled Casting for Semantic Type

The FastLanes writer performs a per-row group check to determine whether a column's physical type can be cast to a smaller representation. While this reduces the on-disk size, the casting step is currently lossy with respect to the semantic information encoded in the column type.

Currently, FastLanes uses `DataType` in its schema metadata. This enumeration already contains semantic values such as `DATE`, `TIMESTAMP`, and `JPEG`. The problem is that a single `DataType` field serves two different purposes: it records the logical meaning of a value, and it also drives the interpretation of its physical representation during encoding and decoding. For example, `DATE` denotes a calendar date even though its underlying representation is an `int32_t`. In addition, for `FFOR` encoding the physical representation is always unsigned, so `DataType` is also used to recover the original signedness. As a result, when FastLanes changes the physical representation, it also rewrites the same `DataType` field. This can overwrite semantically rich types, such as `DATE`, with purely physical types, thereby losing the logical typing information required by downstream consumers.

TPC-H queries rely on the `DATE` type. To prevent FastLanes from optimizing such columns into an incompatible type, we disable the casting step when the incoming column is of type `DataType::Date`[1] (Listing 7).

```
if (column_descriptor.data_type == DataType::DATE) {
    should_be_cast = false;
    return;
}
```

Listing 7: Disabling type optimization for the `DataType::Date`.

This patch introduces only the minimal modifications required to execute the TPC-H benchmark. Other semantic types may still lose their logical meaning when a physical cast is applied. A more appropriate structural solution is to separate the current `DataType` into a `LogicalType` carrying semantic information and a `PhysicalType` describing the physical

[1] https://github.com/cwida/FastLanes/blob/dev/src/table/rowgroup.cpp#L370

encoding. This decoupling allows for physical optimizations without losing logical type information. Implementing such a change would require extending `ColumnDescriptor` with separate `logical_type` and `physical_type` fields, replacing the current `data_type` field. The `physical_type` would then be used for allocation, cast decisions, encoder and decoder dispatch, and the handling of signedness in `FFOR`. The `logical_type`, in turn, would be used by FastLanes consumers to interpret the meaning of the stored physical values.

5.4.2 Projection Support

To read a row group from a FastLanes file, one instantiates a `RowgroupReader`, which creates and populates a buffer based on the supplied row group metadata. In the original implementation, the entire row group blob is read into a single buffer, after which the reader instantiates the decoding expressions for every column in the row group. As a consequence, regardless of the column projection that DuckDB passes to the extension, all columns are read during initialization, and all columns are decoded up to the last operator in their expression chain on each `get_chunk` call.

To avoid unnecessary I/O and decoding work, we extend the `RowgroupReader` constructor with an additional parameter for the projected column identifiers. Instead of reading the full row group blob, the reader uses the column descriptors to allocate and populate separate buffers only for the requested columns. Likewise, only the decoding expressions for the projected columns are built.

A limitation of the current projected-read path is that MCC-encoded columns are not dependency-aware. The decoding expressions still assume that dependent columns are present at their absolute column indices, but the projection logic constructs a dense `m_expressions` vector containing only the projected columns and loads buffers only for those columns. As a result, a derived column may be projected without its source column, causing the decoder to reference an expression and buffer that were never loaded. The extension currently handles MCC-encoded columns by falling back to a full-row-group projection whenever MCC is detected. A more appropriate structural solution is to resolve MCC dependencies inside `RowgroupReader` in the FastLanes library itself, such that the additional required columns are loaded transparently while only the consumer-requested projection is exposed to the extension.

5.4.3 FlatBuffer API

Originally, FlatBuffers were implemented using the object API in FastLanes, of which an example is given in Listing 8. The benefit of using this API is that FlatBuffers are unpacked into native objects for the host language (in this case C++). This makes the resulting objects more natural to use, and allows for the modification of values in these objects. A downside is that this requires destination buffers to be allocated, and data to be copied.

Execution trace analysis revealed that object unpacking accounted for a substantial fraction of end-to-end query latency, leading to a recommendation to change the use of the object API into the zero-copy API. This API is less idiomatic because it exposes the FlatBuffer's internal offset-based layout directly through generated accessors, as can be seen in Listing 9. However, this API points into an existing buffer requiring no further allocations, copies or serialization. Instead, properties are accessed through offsets in the buffer, which boils down to pointer arithmetic. During this thesis, the recommendation has since been implemented by Azim Afroozeh [1].

```
const Table* root = flatbuffers::GetRoot<Table>(buf);
TableT native_obj;
root->UnPackTo(&native_obj);
```

Listing 8: Example usage of the FlatBuffer object API.

```
const Table* root = flatbuffers::GetRoot<Table>(buf);
auto id = root->id();
auto name = root->name()->c_str();
```

Listing 9: Example usage of the FlatBuffer zero-copy API.

5.4.4 File I/O

The FastLanes library uses an I/O abstraction which offers general operations such as `flush`, `append`, `range_read` and so forth. The benefit of such an abstraction is that it can be implemented by different providers, without having to change code that uses this abstraction. Each provider can then communicate using its own underlying protocol, where, for example, one provider communicates with the local file system, and another provider communicates over an internet protocol.

Currently, two providers are supported; one is the `File` provider. This provider internally uses `ifstream` for read operations and `m_of_stream` for write operations. When we look

[1] `https://github.com/cwida/FastLanes/pull/63`

at the cost of `ifstream` in the execution traces we see that it has a significant impact on end-to-end query latency. To improve performance we have modified the `File` provider to use `pread` instead of `ifstream`. In addition, we have replaced `fs::file_size` with `fstat` as well as caching its result to reduce the amount of system calls. The modified functions of the `File` provider are given in Listing 13 (Appendix). One drawback is that `pread` is a POSIX system call, which means it is not compatible with Windows. As a fallback one could use `#define` statements to conditionally compile `pread` for POSIX compatible systems, and `ifstream` for Windows machines.

An alternative approach, and one that would be more appropriate in the long term, would have been to implement FastLanes file access on top of DuckDB's `FileSystem` API. This would have provided additional benefits, such as support for remote files. However, at the time of this work the FastLanes I/O abstraction was not yet used consistently throughout the codebase. As a result, integrating DuckDB's `FileSystem` would first have required a broader refactoring of FastLanes to route all file access through a uniform backend interface. For this reason, the work in this thesis focused first on unifying and optimizing the existing `File` provider.

5.4.5 Additional Statistics

The FlatBuffer metadata for each column currently stores the *max* value and *null count* for that column within a row group. To enable a broader range of predicates for row-group pruning, we extend this metadata with a *min* entry and pass the computed values for numerical columns in the `gather_statistics_visitor` in the `Wizard`[1]. Numerical columns now correctly provide *min*, *max*, and *null count* statistics, as expected.

String columns, however, deviate from the semantics one would typically associate with these statistics. Instead of storing a lexicographically determined UTF-8 *min* and *max*, the existing implementation uses the *max* field to record the maximum number of bytes required to represent any string in the row group. There is no corresponding definition for a *min* value, and it is therefore left unset.

An alternative design would be to avoid embedding type-specific conventions directly in the `ColumnDescriptor`. Instead, the descriptor could use an opaque reference to a dedicated statistics object whose structure depends on the column's logical type. This would support arbitrary, type-specific statistics without polluting or overloading the schema shared by all column types. An example redesign is given in Listing 12 (Appendix) which uses a union to differentiate between column types.

[1] `https://github.com/cwida/FastLanes/blob/dev/src/wizard/wizard.cpp#L45`

6

Evaluation

This chapter evaluates the performance of the FastLanes extension under a variety of configurations. We begin with preliminaries necessary to interpret the results, followed by two main sections. The first examines the performance characteristics of the reader, while the second focuses on the writer. For both components we do a deep-dive into the single-threaded performance, followed by an investigation into scalability in a multi-threaded setting.

6.1 Preliminaries

6.1.1 Setup

All benchmarks are executed on an Apple MacBook Pro, unless explicitly stated otherwise. This platform was chosen as DuckDB targets end-user devices such as laptops, making it a representative environment. To further investigate the impact of differing SIMD instruction sets, we run benchmarks on a server equipped with Intel Xeon processors supporting the AVX-512 extension. The specifications of both machines are summarized in Table 6.1. During benchmark execution, background processes consumed at most 10% of available compute resources.

6.1.2 Benchmarks

We employ two benchmarks in our evaluation: TPC-H (70) and Public BI (71). Each benchmark consists of a set of queries and either a dataset or a specification to generate one. We use them in two complementary ways with regards to the reader: (i) to perform simple volumetric scans for assessing raw decoding throughput, and (ii) to run the complete benchmark for insights into real-world query performance. As the writer does not execute queries, we simply analyze the write performance with the datasets loaded into DuckDB.

	Apple MacBook Pro (M1)	Intel Xeon Server
Processor	Apple M1 Max (ARMv8.5-A)	Intel Xeon Platinum 8259CL (x86_64)
Cores / Threads	10 cores (8P + 2E) / 10 threads	4 cores / 8 threads
Base Frequency	3.2 GHz (P) / 2.1GHz (E)	2.5 GHz
Memory	32 GB unified LPDDR5	32 GB DDR4 ECC
Storage	1 TB NVMe SSD	400GB GP3 Volume (3000IOPS)
Operating System	macOS 15.6 (Sequoia)	Ubuntu 24.04.3 LTS
DuckDB Version	v1.4.0	v1.4.0

Table 6.1: Benchmarking hardware configurations

TPC-H is a widely recognized decision-support benchmark consisting of ad-hoc analytical queries. Its data is generated according to a scale factor, allowing performance analysis across dataset sizes. However, its representativeness has been questioned. Vogelsgesang et al. (72) note that TPC-H queries and schemas were handcrafted by experienced database administrators. As a consequence, the underlying design assumptions differ significantly from real-world scenarios. They highlight several insights that are relevant to our evaluation: (i) TPC-H data follows synthetic distributions, with limited skew compared to real-world workloads; (ii) TPC-H contains only limited correlations between columns, whereas real-world datasets often exhibit stronger inter-column dependencies; (iii) metadata queries occur far more frequently in practice, and although often cacheable, their share of total queries is significant; (iv) real-world systems frequently store values as strings (e.g., dates or booleans), whereas TPC-H uses types more in line with the value domain for easier processing; (v) fixed-precision decimals are extensively used in TPC-H, while platforms may not support fixed-point data types, opting for floating point representations instead; (vi) production datasets are often relatively small, with a long tail of very large datasets.

To address these limitations, we also opt for the inclusion of the Public-BI benchmark. Derived from the Tableau platform, this benchmark reflects workloads generated by real users interacting with data through a visual interface. The benchmark is comprised of the 46 largest workbooks present on the Tableau public registry, with queries extracted from the logs of the associated workbooks.

File Formats For the aforementioned benchmarks, we select a representative set of file formats, in addition to FastLanes, as encoding targets. To offer a representative relative performance comparison, we include the following: *Parquet*, the current industry standard and de facto baseline; *Vortex*, a next-generation format exploring alternative design choices; and *DuckDB*, which likewise represents a modern format, but also illustrates the

trade-offs of tight integration between the storage format and the query engine. Another important consideration is that all mentioned formats are either integrated into DuckDB as an extension, or part of DuckDB itself. Thereby easing a fair comparison between the formats.

6.2 Reader

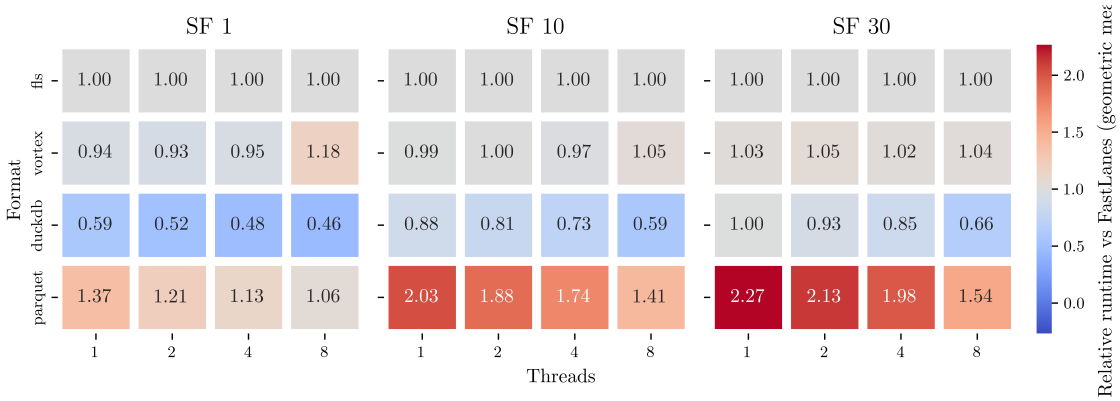


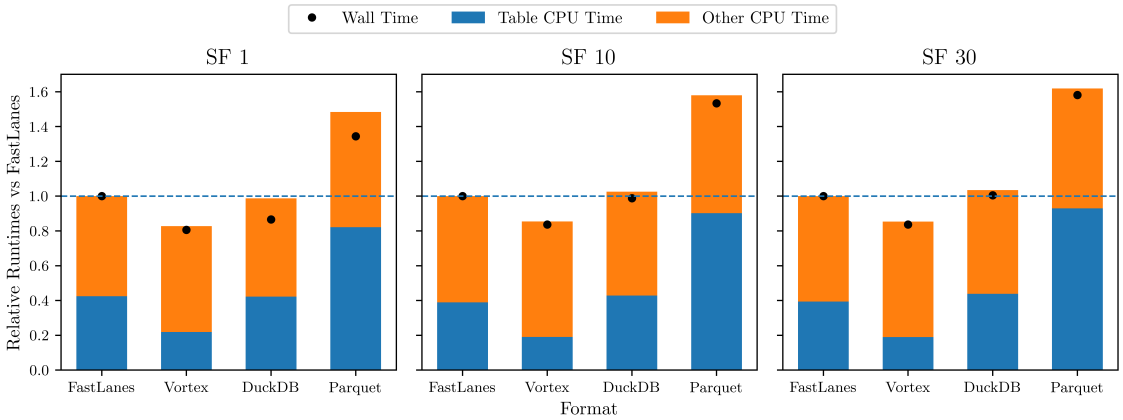
Figure 6.1: Relative geometric mean of the end-to-end runtime of file formats with respect to FastLanes across different thread counts and scale factors. Values below 1 indicate a speedup relative to FastLanes, while values above 1 indicate a slowdown.

6.2.1 Overview

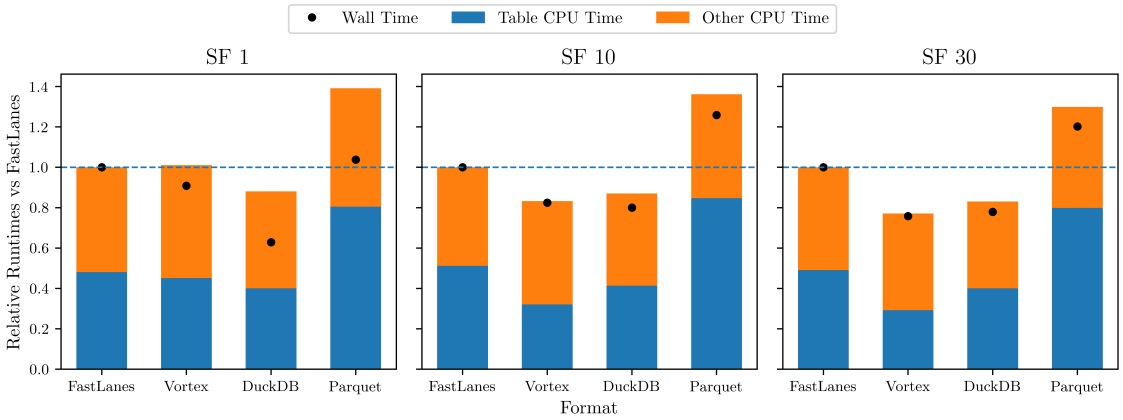
We begin by comparing the general performance characteristics across scale factors and thread counts. The row group size is fixed to `131072` tuples, for which the rationale is discussed in Section 6.2.4.

For this benchmark, we execute a set of single-column volumetric scans for every column in the TPC-H dataset. The set of scans depends on the operators supported by the respective column's data type. For instance, numerical columns perform `SUM`, `AVG`, `MIN`, `MAX` and `COUNT`, while string-typed columns support `MAX`, `COUNT` and `COUNT DISTINCT`. Since volumetric scans are not eligible for optimizations such as predicate pushdown or late materialization, they provide a direct view into the raw decoding performance of each file format.

Each scan is executed at least 5 times, up to a maximum of 60 iterations. To reduce redundant executions, a scan terminates early once the relative standard error falls below 5%. To eliminate the variability of I/O operations, all files required for a given query are



(a) Single-threaded



(b) Multi-threaded with 8 threads.

Figure 6.2: Relative end-to-end runtime of file formats with respect to FastLanes across different scale factors. The runtime is decomposed in time spent in the table function and other processes in the database.

copied to a RAM disk before starting execution. Moreover, DuckDB caches its tables in memory between queries executed on the same database instance. As this behavior does not apply to table functions defined through extensions, we start a fresh DuckDB instance for each query so that DuckDB reads files from the (RAM) disk for each query.

Figure 6.1 shows the geometric mean of relative end-to-end runtimes for all file formats with respect to FastLanes. We find that Vortex and FastLanes perform similarly, while DuckDB is relatively faster when working with smaller scale factors. Parquet has the worst relative end-to-end latency, and FastLanes has a better geometric mean in all instances.

6.2.2 Operator Timings

The total end-to-end runtime for a query can be decomposed into time spent in the table function and time spent in other database processes. This decomposition is given in Figure 6.2 for a row group size of `131072` with execution on both a single thread and eight threads. The values represent the mean runtime per volumetric scan, averaged across all queries sharing the same file format and thread count. In both figures, the bars represent relative CPU time, whereas the black dots represent the relative total wall time.

The *table* CPU time covers loading the data from a file system, decoding the values stored in the respective format, and loading these values into `DataChunk`s. The *other* CPU time covers operators which are not part of the scanning operator of the file format in question, such as projection and aggregation. By decomposing into these constituents we get a clearer picture on where time is being spent, and if certain operations are a target for optimization.

In single-threaded execution we gather that overall, Vortex spends the least amount of time in the table function over all scale factors, having half the running time compared to FastLanes. This is followed by FastLanes and DuckDB, where FastLanes performs better at higher scale factors. Parquet comes last, with the highest table CPU time over all scale factors. The latter is expected, as Parquet has to decode Snappy compressed pages, whereas all other formats use different combinations of LWCs. The performance of Vortex is surprising however, as its set of encodings has significant overlap with FastLanes. Furthermore, as FastLanes uses a data-parallel layout as well as fused encodings not present in Vortex, one would expect FastLanes to have an edge. For the other operators we find that timings are similar between the file formats in all scale factors, with DuckDB and FastLanes spending less CPU time compared to Vortex and Parquet.

When comparing the multi-threaded execution to single-threaded execution, we see that DuckDB and Parquet scale slightly better compared to FastLanes, whereas Vortex has similar scaling properties.

From both Figures we find that the relative wall time often has a different ratio compared to the relative total CPU time. This discrepancy can be attributed to the fact that CPU time does not measure every component in a query execution. For example, CPU timings exclude the optimizer and binding. From this we can derive that DuckDB and Vortex are overall faster in these areas compared to FastLanes and Parquet. For this there are a multitude of reasons; DuckDB's format stores metadata in its file format, as such it can resolve the required information by querying the catalog, which resides in memory. In

contrast, the external formats have to construct a reader, fetch data from the file system, allocate buffers, and parse metadata. Regarding Vortex, from profile traces we find that Vortex does not supply statistics during the query planning phase. This saves time spent on parsing, and therefore also limits optimization time as there is not as much data to work with. In addition, Vortex does not implement the `MultiFileFunction` interface, instead implementing the table function interface directly.

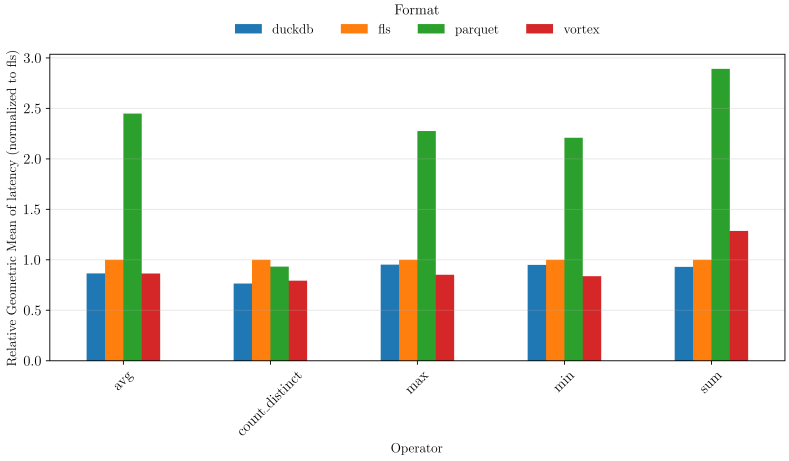
6.2.3 Operator Decomposition

We further decompose the timings per aggregation operator in Figure 6.3. The figures represent the geometric mean of each aggregation operator for each file format, normalized by FastLanes. The figures are based on a scale factor 10, single-threaded execution and row group size of `131072`.

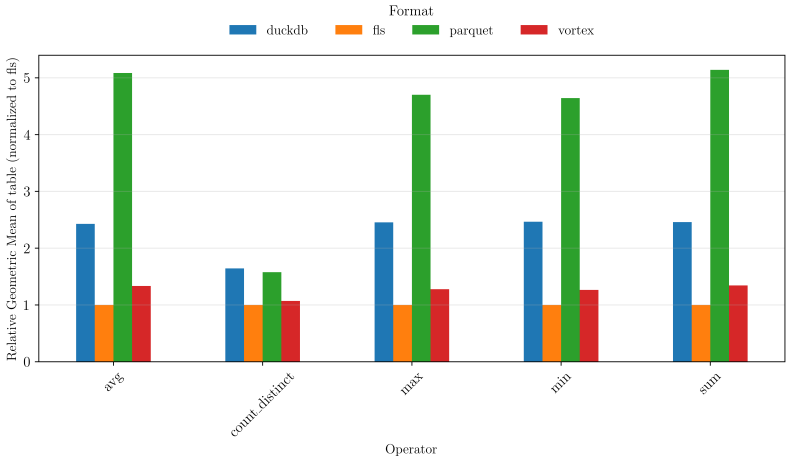
Immediately noticeable is the large discrepancy in `SUM` aggregation in Figure 6.3(c). When looking at the execution traces we find that because Vortex does not gather statistics during the bind phase, DuckDB falls back to a generic `sum` aggregation. Both DuckDB and FastLanes pass enough information upstream for the optimizer to use the `sum_no_overflow` path, which does not do any overflow checks as it can determine that the end result can never exceed the size of the primitive type. While Parquet passes information during the bind phase, it does not pass the necessary cardinality information required, and as such also falls back to the slower path.

We further find that FastLanes achieves the lowest table-level geometric mean across all aggregation operators (Figure 6.3(b)). However, when total query latency is considered this advantage is diminished by less efficient work in the other operators (Figure 6.3(c)) and the query planning step. Examining the traces, there are at least two areas that can be improved in the FastLanes binding phase reducing the impact on query latency. First, the schema can differ between row groups. This requires looping over all columns of all row groups and finding the largest sized type. The resulting type is then passed to DuckDB, instantiating vectors with the appropriate physical type. If we would simply take the type we encounter in the first row group to pass to DuckDB, it might result in overflows when decoding data into DuckDB vectors in later row groups, as DuckDB cannot intermediately change the physical types of vectors. Second, reading of the FlatBuffer metadata currently uses a redundant copy due to interface limitations.

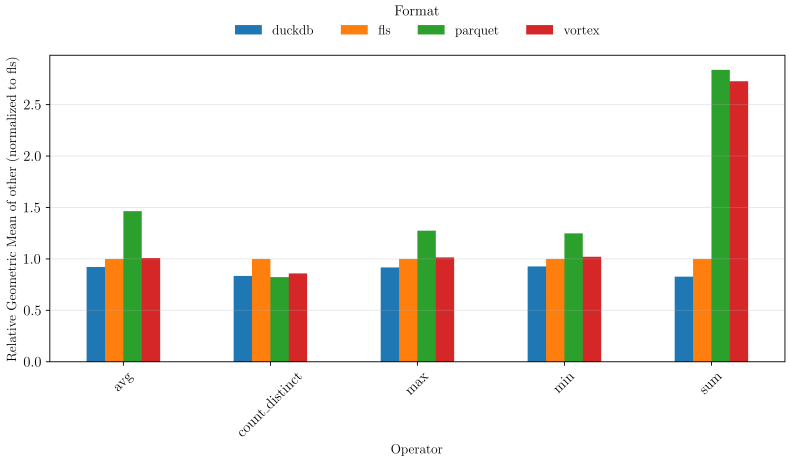
File-Level Schema As highlighted in the earlier section we currently loop over the metadata to find the correct types. This is necessary because row groups may currently



(a) Latency



(b) Table Function



(c) Other Operators

Figure 6.3: Relative geometric mean normalized to FastLanes across all file formats, grouped by aggregation operators.

store different physical types for the same logical column. However, this mechanism adds preventable bind-time overhead. A solution is to provide a file-level schema separate from the row-group local descriptors. This would allow DuckDB to bind against the per-file schema, while using row-group metadata for decoding.

However, even this mechanism is insufficient when working with multiple files. When a logical table is stored across multiple files, other files may contain wider types not present in the first inspected file. In this setting, the proper location for a common schema is table scoped metadata stored externally, rather than the footer of each individual file.

6.2.4 Row Group Size

The file formats in question are based on, or apply, a row group based layout. The row group size has impact on both compression and parallelization. Larger row groups may better exploit the properties of the data domain of each respective column, while also having less metadata overhead per tuple. However, as row groups are the unit of parallelization, careful consideration should go into the cost/benefit balance between compression and decoding speed. In addition, row group sizes should not be too small, as each parallelization unit inherently has startup overhead. In this section we aim to gain insight into the decoding performance across different row group sizes. We measure the performance for row groups sized with `65536`, `131072` and `262144` tuples for Parquet, FastLanes and DuckDB. While Vortex supports changing row group sizes, this is not exposed as configuration in the copy function, and as such will not be considered. The aforementioned sizes are derived from the default row group sizes of the respective file formats, as well as the optimal range for both Parquet and DuckDB (73). Furthermore, all sizes are both divisible by `1024` and `2048`. The former is a consequence of the vectorized execution size of FastLanes, whereas the latter is a requirement for DuckDB files.

In Figure 6.4 we highlight the geometric mean of the end-to-end latency over all volumetric scans performed for each file format (SF30), with separate plots per thread count. From this we gather that there is no significant difference in the overall wall time for FastLanes across different row group sizes. Parquet and DuckDB latency improves slightly overall when the row group size increases, but this difference diminishes with multi-threaded execution. As the largest improvement in performance is observed in going from `65536` to `131072`, and the DuckDB documentation (73) recommends a row group size around `122880` for both DuckDB and Parquet, we pin the row group size to be `131072` in the other benchmarks to limit the amount of dimensions that need to be reasoned over, while remaining representative for real-world scenarios.

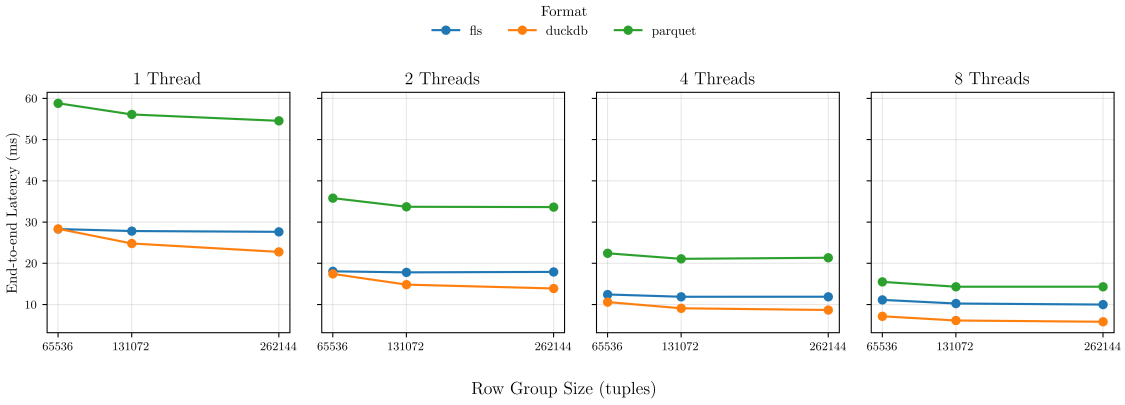


Figure 6.4: End-to-end latency of Parquet, FastLanes, and DuckDB for different row group sizes, with separate plots by thread count.

From Figure 6.4 we find that the FastLanes extension scales worse at higher thread counts than DuckDB's own format. While we have not examined the traces in detail to determine the exact cause, an informed hypothesis can be made based on the locking behavior of the reader. Before the extension enters `TryInitializeScan`, DuckDB's `MultiFileFunction` acquires a global scan lock. If `TryInitializeScan` performs expensive work, other threads may queue behind this lock. A likely improvement would therefore be to keep `TryInitializeScan` lightweight and defer part of the row group setup to the actual scan step, which does not block other threads.

6.2.5 TPC-H

In the previous sections we have focused on raw scanning and decoding performance. We now extend this analysis to include the impact of row group pruning, projection pushdown, predicate pushdown and late materialization. The results are given in Figure 6.5 for each query and file format combination. In the remainder of this section we analyze the results for queries where FastLanes performs relatively worse compared to the other formats. We describe each query in isolation and reason on the runtime behavior with execution traces.

For *Q1* we find that DuckDB has slightly better end-to-end latency while its time spent in the table function is worse. This can be attributed to DuckDB having access to a fast path `PhysicalPerfectHashAggregate`, whereas FastLanes falls back to `RadixPartitionedHashTable`. The fast path operates over numerical data instead of strings by transforming the intermediate chunks. Whether this transformation is possible is decided by the optimizer in `CompressAggregate` calling `GetStringCompress` on a

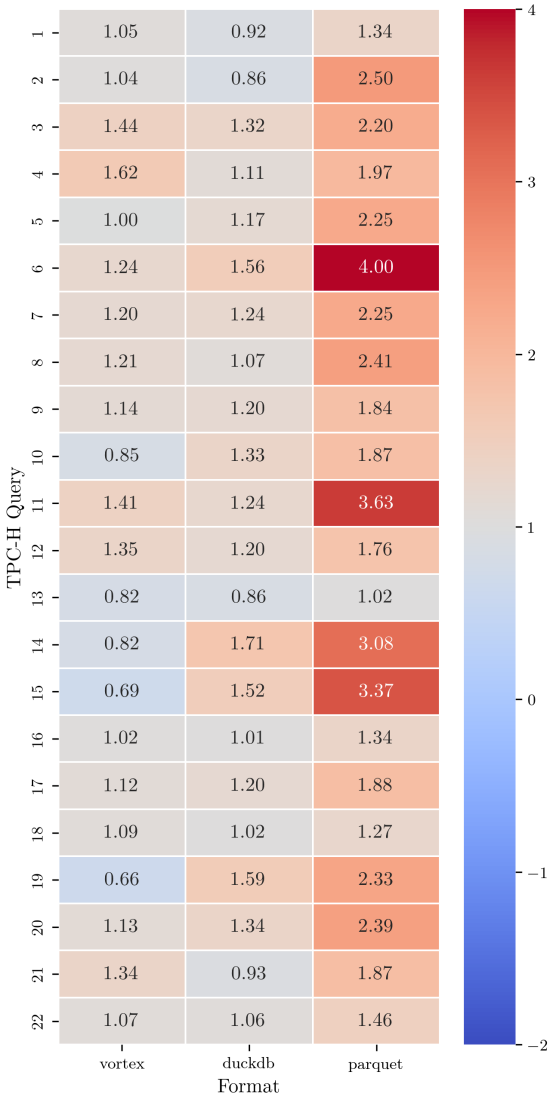


Figure 6.5: End-to-end relative latency of the geometric mean for each TPC-H query, per file format.

`VARCHAR` column. However, FastLanes does not have the required string statistics available in its column descriptors, such as the maximum string length, minimum string value, and maximum string value. As such, the column is not transformed and cannot use the perfect hash path.

For *Q2* we find that DuckDB spends less time in the table function compared to FastLanes, resulting in a comparatively better end-to-end latency. When we track how many rows are filtered out of each file scan we see that for the part table the fraction of filtered rows reaches 99.6%. As FastLanes does not support late materialization, it decodes all values for `p_mfgr`, which is a `VARCHAR` column. A similar case holds for the supplier table, where late materialization would save decoding work on relatively heavy columns such as `s_name`, `s_address`, `s_phone` and `s_comment`. We do not see FastLanes worsening compared to Vortex in this case, as Vortex does not support late materialization either.

For *Q5* Vortex spends less time in the table function compared to FastLanes. This can be attributed to the fact that Vortex supports predicate pushdown into the file format, whereas FastLanes makes use of the more general filter fallback supplied by DuckDB, which is applied in the extension code. The same holds for *Q10*.

For *Q13* both DuckDB and Vortex have better end-to-end latency compared to FastLanes. When analyzing we find an interesting artifact, where Vortex spends almost no time in the table function and double the time in other operators. The reason is that, in this instance, predicates are not pushed down into the Vortex extension, which leads to a logical projection being placed somewhere else in the pipeline by DuckDB. However, the latency differences are likely attributable to late materialization. The columns `o_custkey` and `o_orderkey` are all materialized in FastLanes, instead of using the remainder of the filter `o_comment NOT LIKE '\%special\%requests\%'`.

For *Q14* we find that Vortex has lower end-to-end latency due to less time being spent in the other operators. When we analyze the traces of both FastLanes and Vortex we find that FastLanes triggers caching of emitted chunks in `CachingPhysicalOperator`. The difference in behavior is due to a highly selective filter, and as the current FastLanes extension implementation decodes up to a fixed number of input tuples before the filter is applied, the emitted `DataChunk` falls below the threshold resulting in expensive flatten and copy operations. Vortex, by contrast, evaluates predicates inside the scan, avoiding the materialization of rejected tuples and thereby producing denser `DataChunks` of surviving rows. As a result, it avoids the sparse-chunk path that triggers caching overhead in DuckDB.

For *Q15* Vortex has lower end-to-end latency compared to FastLanes, due to less time being spent in both the table function and the other operators. With the trace we find

that FastLanes spends more time in the `ExpressionExecutor`, as well as other operators, which has similar reasoning to the smaller emitted `DataChunks` as with *Q14*. In addition, Vortex can short circuit decoding columns in a projection when a pushed down predicate evaluates false for the columns participating in a filter evaluation.

For performance differences between FastLanes and the other file formats for the other queries a combination of the earlier mentioned reasons also holds. We therefore find that supporting predicate pushdown into the file format, as well as late materialization, can significantly improve the performance of the FastLanes extension and should be high-priority targets for further optimization.

6.2.6 SIMD

To evaluate the impact of SIMD on the performance of the extension, we run benchmarks on an AWS-provided EC2 instance whose specifications are outlined in Table 6.1. For this experiment, we modified only the compile flags of the FastLanes library by editing its `CMakeLists.txt`; the DuckDB engine and the surrounding extension code were otherwise built unchanged. In its default configuration, FastLanes enables `-mavx512dq` when both the compiler and the host CPU report AVX-512DQ support. As a comparison point, we built FastLanes again with AVX-512 explicitly disabled and with `-march=haswell`, yielding a baseline representative of an AVX2-era target rather than generic x86/SSE2 code. Concretely, this modified configuration removes AVX-512 related flags and explicitly sets `-march=haswell`, `-mno-avx512f`, `-mno-avx512dq`, `-mno-avx512cd`, `-mno-avx512bw` and `-mno-avx512vl`.

We find that the AVX-512-enabled FastLanes build performs slightly worse than the Haswell-targeted baseline, as shown in Figure 6.6. Other configurations, such as different thread counts or decompositions into different query operators, show the same relation and are therefore omitted for brevity.

At first glance, these results are surprising, as AVX-512 uses wider registers than the baseline configuration. However, on some older Intel microarchitectures the use of AVX-512 instructions can lead to a lower sustained CPU frequency (74). To assess whether this effect is present on the tested machine (Listing 10), we execute two simple programs, one using AVX-512 instructions and one using AVX2 instructions (Listing 11). We gather statistics using turbostat, with the flags `--interval 1`, `--no-perf`, and `--show Avg_MHz,Bzy_MHz,TSC_MHz`. On this EC2 instance, we find that when the CPU is busy, the AVX-512 program runs at about 2.7 GHz, whereas the AVX2 program runs at about 3.1 GHz, a difference of roughly

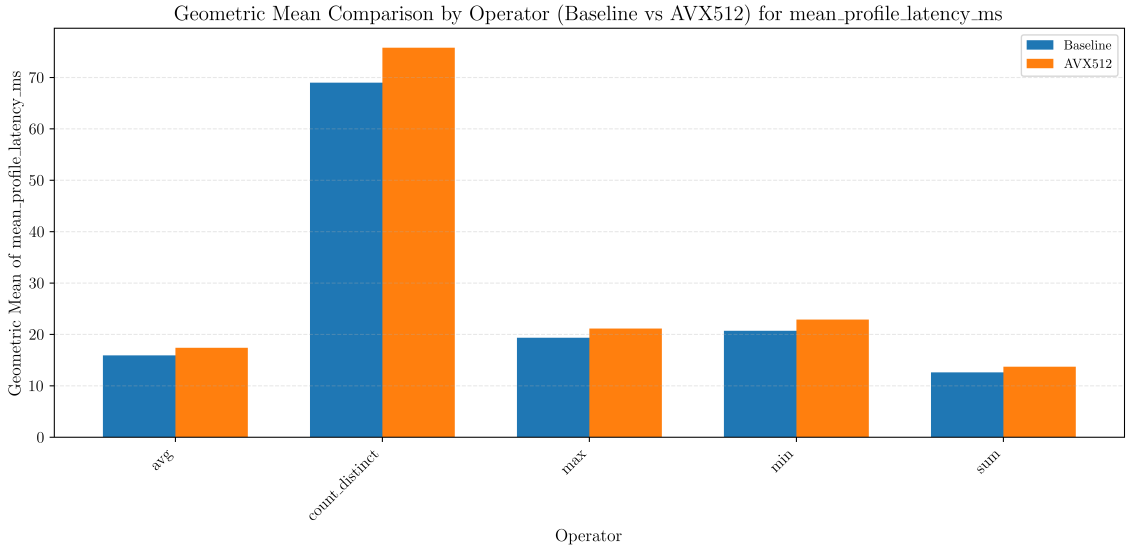


Figure 6.6: Geometric mean of end-to-end latency of volumetric scans for AVX-512 and baseline compiled. (threads=1, SF=10, row group size=131072)

12.9%. We follow up by performing the same analysis for a volumetric scan over a FastLanes view (`select * from lineitem;`), which exhibits the same CPU clock difference.

These measurements are not sufficient to disentangle the contribution of frequency binning from other effects. Nonetheless, on the tested EC2 system they indicate that the AVX-512-enabled FastLanes build runs at a lower CPU frequency than the Haswell-targeted baseline, which can negatively affect end-to-end query latency. It is important to note that the down-clocking behavior of AVX-512 differs across microarchitectures and product lines, and may be absent on newer Intel CPUs or on AMD systems. The cost-benefit trade-off of AVX-512 is therefore hardware dependent.

Additional Benchmarks In addition to the Amazon EC2 instance used in the main evaluation, we repeated the same query set on an Azure VM (Ev6 family, Intel Xeon Platinum 8573C). On this system, compiling the FastLanes library with AVX-512 enabled versus a baseline configuration with AVX-512 explicitly disabled resulted in nearly identical end-to-end performance. Due to restricted capabilities to inspect the state of the underlying hardware, we could not directly measure the impact of AVX-512 instructions on the CPU clock.

```
#include <immintrin.h>
#include <stdint.h>

int main() {
        __m512d v0 = _mm512_set1_pd(1.0);
        __m512d v1 = _mm512_set1_pd(2.0);
        __m512d v2 = _mm512_set1_pd(0.0);

        for (uint64_t i = 0; i < (1ULL<<34); i++) {
                v2 = _mm512_fmadd_pd(v0, v1, v2);
            }

        return (int)((double*)&v2)[0];
    }
```

Listing 10: AVX-512 program executing a fused multiply add (FMA) in a loop, compiled with the following command: `clang -O3 -mavx512f avx512.cpp -o avx512`.

```
#include <immintrin.h>
#include <stdint.h>

int main() {
        __m256d v0 = _mm256_set1_pd(1.0);
        __m256d v1 = _mm256_set1_pd(2.0);
        __m256d v2 = _mm256_set1_pd(0.0);

        for (uint64_t i = 0; i < (1ULL<<34); i++) {
                v2 = _mm256_fmadd_pd(v0, v1, v2);
            }

        return (int)v2[0];
    }
```

Listing 11: AVX2 program executing FMA in a loop, compiled with the following command: `clang -O3 -mavx2 avx.cpp -o avx2`.

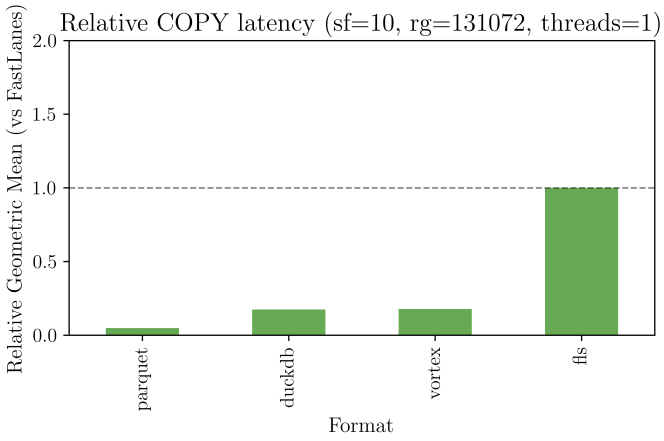


Figure 6.7: Geometric mean of end-to-end latency of `COPY TO` for writing TPC-H tables. (threads=1, SF=10, row group size=131072)

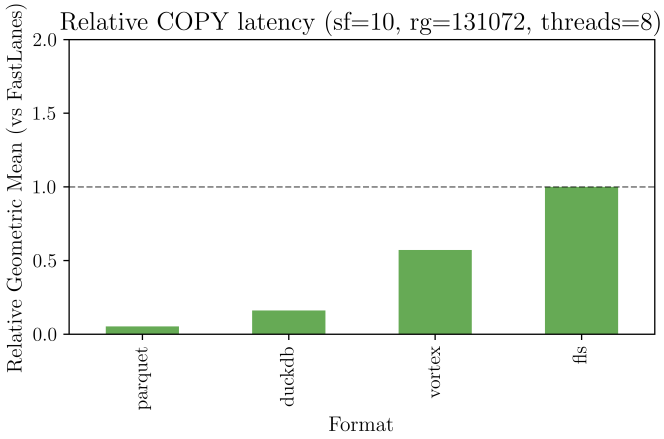


Figure 6.8: Geometric mean of end-to-end latency of `COPY TO` for writing TPC-H tables. (threads=8, SF=10, row group size=131072)

6.3 Writer

To evaluate the performance of the `COPY TO` implementation, we write the TPC-H tables (SF=10) for the columnar formats Parquet, DuckDB, Vortex, and FastLanes. Taking the end-to-end latencies from the profiling runs and computing the geometric mean for each format yields the results in Figure 6.7 (single-threaded) and Figure 6.8 (multi-threaded). FastLanes is consistently slower: in the single-threaded setting DuckDB runs 1.60–8.58× faster, Vortex 4.43–6.88× faster, and Parquet 16.59–25.18× faster.

The reason behind Parquet's advantage becomes clear when analyzing the traces: it uses `Snappy` to compress incoming data, having no expensive encoding selection algorithm in

addition. We further find that FSST compression makes a significant contribution to the end-to-end latency for Vortex and DuckDB.

FastLanes' latency is dominated by a single phase, with `RowGroupWriter::Finalize` accounting for ≈93% of the end-to-end time, delegating to the `Wizard`, which itself burns ≈83% of the wall-clock time. Within the wizard we find that `enc_analyze_opr` is the main contributor to the end-to-end latency, and might offer a target for future optimization.

In multi-threaded COPY, the relative ordering is similar except for Vortex; its speedup over FastLanes drops to 0.78–4.72×, i.e., some tables become slower than FastLanes. By looking at the implementation of the copy function of Vortex we find that only `REGULAR_COPY_TO_FILE` is supported, so it always executes single-threaded regardless of the requested worker count.

Finally, DuckDB's native format is also slower than Parquet and Vortex because it has to do extra work not related to the encoding and flushing of row groups. These include but are not limited to checkpointing, catalog initialization and Multi-Version Concurrency Control (MVCC) bookkeeping.

7

Discussion

In this thesis, we implemented FastLanes in an OLAP DBMS as an extension to DuckDB. We demonstrated that the FastLanes extension outperforms Parquet in all scenarios, while remaining competitive with DuckDB's native storage format and Vortex. However, before the extension is ready for end users, we note limitations of the current extension and of FastLanes, and we indicate directions for future work.

7.1 Overview

The results of this thesis suggest a broader trend in modern data analytics, where storage formats increasingly move toward designs that can better exploit recent hardware advances. Both Vortex and FastLanes follow this direction by moving away from GPC schemes, in favor of LWCs that are better suited to exploit SIMD execution and GPU-based processing. Our evaluation shows that this direction is promising, as both formats clearly outperform Parquet on raw scan workloads.

At the same time, the results show that raw decoding speed alone is not sufficient to guarantee the best end-to-end query performance. Query optimizations remain an important factor, by avoiding unnecessary decoding and materialization through row group pruning, column projection, predicate pushdown, late materialization and compressed execution. We further find that the capabilities of current DBMSs should be extended to fully exploit these next-generation file formats. In particular, database engines would benefit from more flexible DBMS-facing APIs, such as broader support for consuming compressed vectors and for handling dynamic physical representations across row groups.

7.2 Reader

Compressed Vectors We have shown that emitting compressed vectors yields a performance improvement for certain encodings such as dictionaries and constant vectors. This implies that FastLanes is most suitable for consumers that are able to utilize compressed execution, preventing premature decompression steps. DuckDB currently supports a subset of the compression schemes supported by FastLanes (`FSST_VECTOR`, `CONSTANT_VECTOR`, `DICTIONARY_VECTOR` and `SEQUENCE_VECTOR`). Other LWC encodings such as `FSST12` and `FFOR` still have to be fully decompressed into a flat vector. Therefore, extending support for additional LWC schemes may further improve the performance of FastLanes and other LWC-based columnar formats.

In addition, not all DuckDB vectors enjoy the same level of operator support. As we mentioned earlier, both `CONSTANT_VECTOR` and `DICTIONARY_VECTOR` have specialized paths for `ToUnifiedFormat` and `VectorOperations::Copy`. When a consumer hits the other compressed vectors, it will simply decompress them into a flat vector. Therefore, these encodings are mostly beneficial for the storage layer but offer little execution benefit.

Dynamic Physical Types FastLanes is able to change the underlying physical representation for a column across row groups. However, DuckDB currently configures the physical type of a column based on the logical type set during the bind phase. It is currently not possible to change the physical type at a later point in the scan, thus potentially requiring additional casting operations. This implies FastLanes' performance will improve with a more flexible consumer. For DuckDB, this would entail being able to redefine the physical type during scan initialization for each worker when starting a new row group.

FastLanes Schema The current FastLanes schema is based on the `DataType` enum, which mixes the physical representation and the semantic meaning of a column in a row group. When FastLanes casts an incoming schema to a more optimal physical representation to store on disk it erases the associated semantic information. Furthermore, as there is no file-scoped or shared table schema, extra upfront work is required to ensure that we emit a wide enough logical type during the bind phase in DuckDB. Future work should split the `DataType` into a `LogicalType` and `PhysicalType` to prevent the erasure of semantic information. In addition, there should be both a row-group schema and a file-scoped schema so that correct types can be emitted for less flexible consumers.

Metadata The columnar metadata of FastLanes currently comprises minimal statistics, and mixes the usage of metadata fields across different column types (e.g. `max` has a different semantic meaning for numerical and string columns). Future work should extend the available statistics in the metadata, and possibly redesign how metadata is referenced by a `ColumnDescriptor` to make it more resilient to future modifications.

Incomplete Type Support The FastLanes extension currently does not support NULL and complex types (STRUCT, LIST, MAP). This decision is made consciously as the FastLanes format does not provide a specification or implementation for NULL types. The implementation for complex types is experimental and subject to change, and has therefore not been included in the extension.

Predicate Pushdown The extension currently does not support predicate pushdown into a FastLanes file, as the underlying implementation is incomplete. The extension does have infrastructure present through `FastLanesScanFilter`, which can be used to do extension scoped predicate evaluation.

Column Projection FastLanes currently does not support column projection. To enable usage in the extension, we patched the FastLanes code (Section 5.4.2) to support taking a vector of column IDs. This patch is limited in functionality however, as it does not support MCC schemes. Future work should extend on the patch as described in the respective section.

Extension Limitations There are currently two major limitations in the extension's implementation.

First, the extension uses batches to load multiple FastLanes vectors into a single DuckDB vector. However, the extension only loads a maximum of 2 FastLanes vectors per column in a `DataChunk` per scan call, a highly selective filter may therefore emit low-density `DataChunk`s. The scan loop should be modified to allow for an arbitrary amount of FastLanes vectors to be loaded, depending on the space left in the DuckDB vector associated with the `DataChunk` (min 1024 values). This would mean introducing something akin to a `filled` variable which tracks the amount of valid rows present in the vector, instead of using the current templated approach which uses `Pass:First` and `Pass:Second`. In addition, instead of filtering after loading in all FastLanes vectors, the filter should be applied on every load iteration.

Second, the extension does not support late materialization. As shown in the results, this has a direct impact on end-to-end latency for TPC-H queries. Future work should therefore extend the FastLanes API to support late materialization, and integrate this into the extension.

7.3 Writer

Implementation The current implementation of the copy to function extension depends on a heavily modified version of the original FastLanes writer. While modifications relate mainly to ingestion, careful consideration is required to decide whether the newly introduced streaming API is desirable and should be fully integrated into the FastLanes library code.

Performance The performance of the FastLanes writer is currently not competitive with respect to the other evaluated formats. As end-to-end latency is mainly impacted by processes deeper in the FastLanes format, and considering time constraints, we chose not to optimize the related code further. Future work should focus on either reducing time spent in `enc_analyze_opr`, or reducing the amount of times the associated `Analyze` function is called.

8

Conclusion

In this thesis, we studied the columnar formats Parquet, Vortex, and FastLanes, and we described their binary layouts and encoding approaches. We also described the DuckDB internals needed to understand the behavior of table and COPY functions. Finally, we applied this knowledge by presenting and evaluating a FastLanes extension that integrates into the OLAP DBMS DuckDB. The remainder of this chapter is dedicated to answering the research questions stated in the introduction.

RQ1: What is an appropriate extension architecture for integrating FastLanes into DuckDB with minimal overhead? For the reader, we described an implementation that uses decode kernels with a prepare step and a decode step, so the extension prepares row-group-scoped state and transient buffers once and then reuses them in a per-vector decode step. In addition, the use of compressed vectors, such as dictionaries, prevents repetitive or redundant copies. Finally, when designing an extension, one should take into account possible auxiliary structures like `MultiFileInfo`, which abstracts away common logic and allows the extension to be integrated into other extensions that build on top of these structures, such as data lake extensions.

For the writer, we described an implementation that supports all available execution modes. In this implementation, we wrap source DuckDB vectors into spans to transform data into a representation ingestible by FastLanes with minimal copies.

RQ1.1: Is the current FastLanes library API sufficient for efficient integration? For the reader, the existing FastLanes API cannot handle column projection, late materialization, and predicate pushdown. In addition, there is no file-scoped schema present which can lead to errors when decoding into DuckDB vectors when scanning multiple files. Finally, `ColumnDescriptor`s miss important statistical information that can be passed to

the optimizer, such as min values for numerical columns or min and max statistics for string columns.

For the writer, the existing FastLanes API does not expose a streaming interface capable of row group at-a-time encoding and flushing, and it only supports `csv` and `json` input files. In addition, there are no mechanics to seed `ColumnDescriptor`s with DuckDB-sourced schema information.

RQ1.2: What extensions or modifications are required or recommended for the FastLanes library? For the reader, we have introduced several patches as outlined in Section 5.4. Based on these patches, we made the following modifications: we extended the `RowGroupReader` interface to support projection, added an additional `min` statistic for numerical columns, and used `pread` instead of `ifstream` to reduce system call overhead. We further derive the following recommendations: separation of the `DataType` into a `LogicalType` and `PhysicalType` to disentangle responsibilities of binary interpretation and semantic meaning, the addition of a global schema alongside row-group-scoped schemas to support less flexible consumers, additional statistics for string-based columns as well as a future-proof metadata structure for these statistics, and an extension to the external API to support late materialization.

For the writer, modifications were required to the ingestion interface, introducing a row-group-based streaming API with support for wrapping arbitrary data into spans ingestable by the FastLanes library. We also modified existing encoding selection and encoding functions to support row-group-granularity execution.

RQ2: What is the read and write performance of FastLanes on OLAP workloads in DuckDB compared to Parquet, Vortex, and DuckDB's internal format? For the reader, FastLanes has a relative speedup, based on the geometric mean of single-threaded execution for all scale factors, for the end-to-end latency of 0.589–0.999× compared to DuckDB, 1.370–2.267× for Parquet and 0.945–1.026× for Vortex when considering volumetric scans. For the table scan time FastLanes has a relative speedup of 2.290–2.502× compared to DuckDB, 4.063–4.153× compared to Parquet, and 1.289–2.385× compared to Vortex. For multi-threaded execution, the end-to-end latency and table scan time improve for DuckDB and Parquet relative to FastLanes, while they worsen for Vortex.

For TPC-H, FastLanes has a relative speedup, based on the geometric mean of single-threaded execution for SF30, for the end-to-end latency of 0.86–1.71× compared to DuckDB, 1.02–4.00× compared to Parquet, and 0.66–1.62× compared to Vortex. For the table scan

time FastLanes has a relative speedup of 0.79–2.44× compared to DuckDB, 1.03–6.65× compared to Parquet, and 0.13–3.11× compared to Vortex. For multi-threaded execution, the relative end-to-end latency becomes 0.72–1.34× for DuckDB, 0.37–1.58× for Vortex and 0.95–2.43× for Parquet. The relative table times are 0.64–1.81× for DuckDB, 0.16–2.68× for Vortex, and 0.89–4.57× for Parquet.

For the writer, we find that the relative end-to-end latency compared to FastLanes, in single-threaded execution and SF10, is 0.117–0.623× for DuckDB, 0.145–0.226× for Vortex, and 0.040–0.060× for Parquet. Considering multi-threaded execution, the relative end-to-end latencies become 0.106–0.272× for DuckDB, 0.212–1.286× for Vortex, and 0.037–0.076× for Parquet.

Closing Remarks In conclusion, the results of this work validate the broader design direction of next-generation file formats such as Vortex and FastLanes. Lightweight, data-parallel encodings can provide substantial scan-performance benefits in an OLAP setting. At the same time, this thesis shows that fully realizing these benefits requires additional work outside the file formats themselves. The consuming DBMS must be able to preserve compact representations, so as to avoid unnecessary decoding and materialization during query execution, and support more flexible physical typing to reduce casting operations. Likewise, file formats should expose APIs and metadata that enable query optimizations such as projection pushdown, predicate pushdown, row-group pruning, and late materialization. Future progress in this area therefore depends not only on improving file formats, but also on corresponding advances in database execution engines.

References

[1] Alex Petrov. *Database Internals: A Deep Dive into How Distributed Data Systems Work*. O'Reilly Media, Inc., Sebastopol, CA, November5 2019. vii, 11, 13

[2] Peter Boncz, Thomas Neumann, and Viktor Leis. **FSST: Fast Random Access String Compression**. **13**(12):2649–2661. viii, 18, 19

[3] Thomas Neumann and Michael Freitag. **Umbra: A Disk-Based System with In-Memory Performance**. viii, 21

[4] Apache Parquet Project. **File Format** [online]. July 2024. Overview of the Apache Parquet file format, including layout and structural specifications; last modified July 7 2024. [cited 2025-08-29]. viii, 26

[5] **File Format · Vortex Documentation** [online]. September 2025. Specification of the Vortex file format, including its magic header, postscript structure, schema location, footer, and design goals such as compatibility, encryption, and efficient layout querying [cited 2025-09-01]. viii, 30

[6] Martin Loncaric, Niels Jeppesen, and Ben Zinberg. **Pcodec: Better Compression for Numerical Sequences**. viii, 34, 35

[7] Azim Afroozeh and Peter Boncz. **The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code**. **16**(9):2132–2144. viii, 38, 41

[8] Azim Afroozeh and Peter Boncz. **The FastLanes File Format**. **18**(11):4629–4643. xi, 1, 42

[9] Apache Parquet Project. **Apache Parquet: Open-Source Column-Oriented Data File Format** [online]. Official project homepage describing Parquet's design and purpose: an open-source, column-oriented data file format supporting efficient data storage and retrieval [cited 2025-08-27]. 1, 25

[10] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. **BtrBlocks: Efficient Columnar Compression for Data Lakes**. *Proceedings of the ACM on Management of Data*, **1**(2):1–26, June 2023. 1, 35

[11] Jeff Dean, Sanjay Ghemawat Ghemawat, and Steinar H. Gunderson. **Snappy: A fast data compression and decompression library**. Online; posted on GitHub and Google open-source repository, 03 2011. 1, 14

[12] Coder Spirit. **Exploring the LZ77 Algorithm**. Accessed via Wayback Machine archive. 1, 15

[13] **Facebook Incubator - The Nimble File Format** [online]. November 2025. GitHub repository for The Nimble File Format [cited 2025-11-17]. 1

[14] **vortex-data/vortex: An extensible, state-of-the-art columnar file format (GitHub repository)** [online]. September 2025. GitHub repository for Vortex — a next-generation columnar file format and toolkit, formerly hosted at SpiralDB and now a Linux Foundation project [cited 2025-09-01]. 1, 29

[15] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an Embeddable Analytical Database**. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD/PODS '19. ACM, June 2019. 1

[16] Alexander van Renen and Viktor Leis. **Cloud Analytics Benchmark**. *Proc. VLDB Endow.*, **16**(6):1413–1425, February 2023. 2

[17] Surajit Chaudhuri and Umeshwar Dayal. **An overview of data warehousing and OLAP technology**. *ACM SIGMOD Record*, **26**(1):65–74, March 1997. 5

[18] James Serra. *Deciphering data architectures: choosing between a modern data warehouse, data fabric, data lakehouse, and data mesh.* O'Reilly Media, Incorporated, Beijing Boston Farnham Sebastopol Tokyo, first edition edition, 2024. 5

[19] Lennart Schmidt, Johannes Pietrzyk, Juliana Hildebrandt, Alexander Krause, Dirk Habich, and Wolfgang Lehner. **Rethinking MIMD-SIMD Interplay for Analytical Query Processing in In-Memory Database Engines**. 5, 7

[20] Stefan Manegold. **Memory Hierarchy**. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1–8. Springer New York. 5

[21] Peter Boncz, Stefan Manegold, and Martin L Kersten. **Database Architecture Optimized for the New Bottleneck: Memory Access**. 6

[22] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. **DBMSs On A Modern Processor: Where Does Time Go?** 6

[23] Johns Paul, Shengliang Lu, and Bingsheng He. **Database Systems on GPUs**. *Found. Trends Databases*, **11**(1):1–108, July 2021. 7

[24] Andrew Crotty. **Lecture Notes Query Processing I**. *Carnegie Mellon University*, 2021. 8, 9, 10

[25] **Advanced Database Systems - Query Evaluation: Processing Models**, 2024. 9, 10

[26] Joy Arulraj. **Database System Implementation - Query Execution (Part 1)**. 9, 10

[27] Jaba Giceva. **Data Processing on Modern Hardware - Cache awareness for query execution models**. 9, 10

[28] Mark Raasveldt. **DuckDB Internals (CMU Advanced Databases / Spring 2023)**, April 2023. 10

[29] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. **Weaving Relations for Cache Performance**. 11, 12, 13

[30] The PostgreSQL Global Development Group. **65.6 Database Page Layout**, 2025. 11

[31] George P. Copeland and Setrag N. Khoshafian. **A decomposition storage model**. *SIGMOD Rec.*, **14**(4):268–279, May 1985. 11, 12

[32] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. **An Empirical Evaluation of Columnar Storage Formats**. 13, 15

[33] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. **RCFile: A Fast and Space-Efficient Data Placement Structure in MapReduce-based Warehouse Systems**. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1199–1208. IEEE. 13

[34] Apache ORC Project. **Apache ORC – High-Performance Columnar Storage for Hadoop**. Official homepage of the Apache ORC project; self-describing, type-aware columnar file format optimized for large-scale Hadoop workloads. 13

[35] Apache ORC Project. **ORC Specification v1**. Online documentation — Optimized Row Columnar (ORC) file format version 1; accessed via official Apache ORC site. 13

[36] Daniel Abadi, Samuel Madden, and Miguel Ferreira. **Integrating Compression and Execution in Column-Oriented Database Systems**. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 671–682. ACM. 14, 15, 16

[37] Yann Collet and Murray Kucherawy. **Zstandard Compression and the 'application/zstd' Media Type**. RFC 8878, February 2021. 14

[38] J. Ziv and A. Lempel. **A Universal Algorithm for Sequential Data Compression**. **23**(3):337–343. 14

[39] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. **Column-Stores vs. Row-Stores: How Different Are They Really?** In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 967–980. ACM. 15

[40] Mark Raasveldt. **Lightweight Compression in DuckDB**. 15, 16

[41] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. **Column-Oriented Database Systems**. **2**(2):1664–1665. 16, 17

[42] Joshua Lockerman and Ajay Kulkarni. **Time Series Compression Algorithms Explained**, March 2024. Archived from the original on 11 August 2025 via Wayback Machine. 17

[43] Julia Spindler, Philipp Fent, Adrian Riedl, and Thomas Neumann. **Can Delta Compete with Frame-of-Reference for Lightweight Integer Compression?** 17, 18

[44] **IEEE Standard for Floating-Point Arithmetic**. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. 18

[45] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. **ALP: Adaptive Lossless Floating-Point Compression**. **1**(4):1–26. 20

[46] **Why German Strings are Everywhere** [online]. July 2024. Blog post on "German style strings" from the CedarDB Blog by Lukas Vogel [cited 2025-10-20]. 21

[47] Julien Le Dêm. **Announcing Parquet 1.0: Columnar Storage for Hadoop** [online]. July 2013. Engineering blog post announcing the 1.0 release of Parquet, a columnar storage format developed collaboratively by Twitter and Cloudera. [cited 2025-08-27]. 25

[48] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. **Dremel: Interactive Analysis of Web-Scale Datasets**. 25, 28

[49] Apache Parquet Project. **Implementation Status** [online]. May 2025. Documentation page summarizing feature support across Parquet implementations (last modified May 20 2025). [cited 2025-08-27]. 25

[50] Apache Thrift Project. **Apache Thrift — Scalable Cross-Language Services Framework** [online]. Official project homepage for Apache Thrift, a software stack and code generation engine for building cross-language services [cited 2025-08-28]. 26

[51] Apache Thrift Project. **THRIFT-110: A More Compact Format** [online]. February 2009. Apache JIRA issue regarding enhancements for a more compact Thrift protocol; marked "Closed" and "Fixed." [cited 2025-08-28]. 27

[52] Erik van Oosten. **Thrift Specification – Comparing Binary and Compact Protocol** [online]. 2016. Section within "Thrift specification – Remote Procedure Call," documenting a comparison between the binary and compact Thrift protocols. [cited 2025-08-28]. 27

[53] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. **Thrift: Scalable Cross-Language Services Implementation**. 27

[54] Apache Parquet Project. **Page Index** [online]. January 2024. Documentation page describing the format and purpose of Parquet's column and offset index pages; last modified January 14 2024. [cited 2025-08-29]. 28

[55] Apache Parquet Project. **Bloom Filter** [online]. March 2024. Documentation page describing Parquet's support for split block Bloom filters—compact data structure for predicate pushdown in high-cardinality columns; last modified March 11 2024. [cited 2025-08-29]. 28

[56] Apache Parquet Project. **Compression** [online]. March 2024. Documentation page describing supported compression codecs for Parquet data and dictionary pages; last modified March 11 2024. [cited 2025-08-29]. 28

[57] Apache Parquet Project. **Encodings** [online]. March 2024. Documentation page detailing Parquet's supported encodings for data pages—plain, dictionary, RLE/bit-packing hybrid, delta encodings, byte-stream split; last modified March 11 2024. [cited 2025-08-29]. 28

[58] **SpiralDB: A Database for Any Data, Any Size** [online]. September 2025. Official homepage of SpiralDB, a multimodal database built atop scalable open-source file formats for flexible, large-scale data storage and analytics [cited 2025-09-01]. 29

[59] Nicholas Gates. **Life in the Fastlanes: Decoding "greater-than 100 billion" Integers per Second with Scalar Rust** [online]. June 2024. Archived performance blog post on scalar Rust decoding [cited 2025-09-01]. 29

[60] Nicholas Gates. **Vortex – A Linux Foundation Project** [online]. August 2025. Archived page announcing the Vortex project as contributed to the Linux Foundation [cited 2025-09-01]. 29

[61] **FlatBuffers: Efficient Cross-Platform Serialization Library** [online]. September 2025. Official documentation site for FlatBuffers — an efficient, zero-copy cross-platform serialization library originally developed by Google [cited 2025-09-01]. 32

[62] Nicholas Gates. **Towards Vortex 1.0** [online]. April 2025. Archived blog post discussing the upcoming Vortex 1.0 release [cited 2025-09-01]. 32

[63] **cwida / FastLanes: Towards a New File Format** [online]. October 2025. GitHub repository for FastLanes — Next-Gen Big Data File Format [cited 2025-10-06]. 36

[64] Centrum Wiskunde & Informatica (CWI). **CWI — Centrum Wiskunde & Informatica** [online]. 2025 [cited 2025-10-06]. 36

[65] Sven Hielke Hepkema. **G-ALP: Rethinking GPU Decompression of LightWeight Encodings**. 2025. 42

[66] **dlopen(3) — Linux manual page** [online]. September 2025. Linux man page describing the `dlopen()` function for loading dynamic shared objects at runtime [cited 2025-09-05]. 47

[67] **dlsym(3) — Linux manual page** [online]. September 2025. Linux man page describing the `dlsym()` function for symbol lookup in dynamically loaded objects [cited 2025-09-05]. 47

[68] DuckDB Labs. **extension-template: A template for DuckDB extensions** [online]. September 2025. GitHub repository containing a ready-made template to help you develop, test, and distribute DuckDB extensions [cited 2025-09-05]. 47

[69] Raufs Dunamalijevs. **Predicate Pushdown in FastLanes**. 65

[70] Transaction Processing Performance Council (TPC). **TPC-H Benchmark Description**. 77

[71] CWI Database Architectures. **public_bi_benchmark**. GitHub repository. 77

[72] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. **Get Real: How Benchmarks Fail to Represent the Real World**. In *Proceedings of the Workshop on Testing Database Systems*, pages 1–6. ACM. 78

[73] DuckDB Foundation. **Documentation - File Formats**. 84

[74] Franek Korta. **Avx throttling (Part I)**, april 2020. 88

Appendix

```
// file: flatbuffers_schemas/column_descriptor.fbs
...

table ColumnStatisticsNumeric {
  min: BinaryValue;
  max: BinaryValue;
  null_count: ulong;
}

table ColumnStatisticsString {
  min: BinaryValue;
  max: BinaryValue;
  null_count: ulong;
  max_length: ulong;
}

union ColumnStatistics {
  ColumnStatisticsNumeric,
  ColumnStatisticsString
}

table ColumnDescriptor {
  data_type:           DataType;
  encoding_rpn:        RPN;
  idx:                 ulong;
  name:                string;
  children:            [ColumnDescriptor];
  statistics:          ColumnStatistics;
  column_offset:       ulong;
  total_size:          ulong;
  expr_space:          [ExpressionResult];
  segment_descriptors: [SegmentDescriptor];
}

root_type ColumnDescriptor;
```

Listing 12: Example redesign in which `ColumnDescriptor` references a type-specific statistics object rather than definining the entries directly.

```
File::~File() {
        if (m_of_stream != nullptr) { FileSystem::close(*m_of_stream); }
        if (m_fd >= 0) { FileSystem::close(m_fd); }
}
void File::Read(const Buf& buf) {
        if (m_fd == -1) {
                m_fd = FileSystem::open_r_binary(m_path);
        }
        if (m_file_size == -1) {
                m_file_size = FileSystem::read_file_size(m_fd);
        }
        FLS_ASSERT_LE(m_file_size, buf.Capacity());
        ReadInternal(buf.mutable_data(), static_cast<n_t>(m_file_size));
}
void File::ReadRange(const Buf& buf, const n_t offset, const n_t size) {
        if (m_fd == -1) {
                m_fd = FileSystem::open_r_binary(m_path);
        }
        if (m_file_size == -1) {
                m_file_size = FileSystem::read_file_size(m_fd);
        }
        FLS_ASSERT_LE(offset + size, m_file_size);
        FLS_ASSERT_LE(size, buf.Capacity());
        ReadInternal(buf.mutable_data(), size, static_cast<off_t>(offset));
}
n_t File::Size() {
        if (m_file_size == -1) {
                m_file_size = FileSystem::read_file_size(m_fd);
        }
        return static_cast<n_t>(m_file_size);
}
void File::ReadInternal(uint8_t* dst, const n_t size, const off_t offset) const {
        n_t done = 0;
        while (done < size) {
                const ssize_t n_bytes = ::pread(m_fd, dst + done, size - done,
                                        offset + static_cast<off_t>(done));
                if (n_bytes < 0 && errno == EINTR) {
                        continue;
                }
                if (n_bytes <= 0) {
                        FLS_ABORT("Could not read from file in read-only mode")
                }
                done += static_cast<n_t>(n_bytes);
        }
}
```

Listing 13: Modified `File` provider functions, using `pread` and `fstat` instead of `ifstream` and `fs::file_size`.