F3: The Open-Source Data File Format for the Future

XINYU ZENG, Tsinghua University, China
RUIJUN MENG, Tsinghua University, China
MARTIN PRAMMER, Carnegie Mellon University, USA
WES MCKINNEY, Posit PBC, USA
JIGNESH M. PATEL, Carnegie Mellon University, USA
ANDREW PAVLO, Carnegie Mellon University, USA
HUANCHEN ZHANG*, Tsinghua University, China and Shanghai Qizhi Institute, China

Columnar storage formats are the foundation for modern data analytics systems. The proliferation of open-source file formats (i.e., Parquet, ORC) allows seamless data sharing across disparate platforms. However, these formats were created over a decade ago for hardware and workload environments that are much different from today. Although these formats have incorporated some updates to their specification to adapt to these changes, not all deployments support those modifications, and too often systems cannot overcome the formats' deficiencies and limitations without a rewrite.

In this paper, we present the **Future-proof File Format** (F3) project. It is a next-generation open-source file format with interoperability, extensibility, and efficiency as its core design principles. F3 obviates the need to create a new format every time a shift occurs in data processing and computing by providing a data organization structure and a general-purpose API to allow developers to add new encoding schemes easily. Each self-describing F3 file includes both the data and meta-data, as well as WebAssembly (Wasm) binaries to decode the data. Embedding the decoders in each file requires minimal storage (kilobytes) and ensures compatibility on any platform in case native decoders are unavailable. To evaluate F3, we compared it against legacy and state-of-the-art open-source file formats. Our evaluations demonstrate the efficacy of F3's storage layout and the benefits of Wasm-driven decoding.

CCS Concepts: • **Information systems → Data layout**; **Column based storage**.

Additional Key Words and Phrases: Columnar Storage, File Format, Compression, Extensibility

ACM Reference Format:
Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M. Patel, Andrew Pavlo, and Huanchen Zhang. 2025. F3: The Open-Source Data File Format for the Future. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 245 (September 2025), 27 pages. https://doi.org/10.1145/3749163

1 Introduction

Today's data ecosystem is predicated on open-source data file formats [18, 25, 26, 58]. The leading formats, Parquet [26] and ORC [25], originate from the early 2010s, when organizations developed new systems for data analytics [19], such as Hive [20, 79], Impala [23], and Spark [28, 82]. These

*Corresponding author.

Authors' Contact Information: Xinyu Zeng, Tsinghua University, China, zeng-xy21@mails.tsinghua.edu.cn; Ruijun Meng, Tsinghua University, China, mrj21@mails.tsinghua.edu.cn; Martin Prammer, Carnegie Mellon University, USA, mprammer@cs.cmu.edu; Wes McKinney, Posit PBC, USA, wes@posit.co; Jignesh M. Patel, Carnegie Mellon University, USA, jigneshp@cs.cmu.edu; Andrew Pavlo, Carnegie Mellon University, USA, pavlo@cs.cmu.edu; Huanchen Zhang, Tsinghua University, China and Shanghai Qizhi Institute, China, huanchen@tsinghua.edu.cn.

ACM 2836-6573/2025/9-ART245
https://doi.org/10.1145/3749163

formats enable organizations to share data across disparate data warehouses and data lakes [21, 22, 27, 28, 30, 35, 51]

A decade after these formats' inception, studies show they are insufficient for modern data analysis [65, 83]. This problem is because the formats' designers relied on outdated assumptions about (1) relative hardware performance and (2) workload access patterns. With the former, storage and network performance has improved by orders of magnitude in the last 10 years, whereas computational performance (CPUs) has not. The rise of data lakes also means that more data resides in high-bandwidth and high-latency cloud storage (e.g., Amazon S3), which often causes systems to be bottlenecked on compute and not I/O. Next, the "width" of data has also increased over the years: applications no longer only store tabular data with a small number of columns in data files. It is now common in machine learning (ML) workloads to store features in wide tables with thousands of columns, as well as high-dimensional vector embeddings or large blobs (e.g., images, videos) in the file. Applications also want to perform random access or even updates on these files. Existing formats are not suited for these usage patterns and, therefore, are woefully inefficient for them.

Recent advancements in compression [48, 52, 63, 86], indexing [75], and filtering [59, 85] methods address these defencencies. But existing file formats are not designed to be easily extensible to support such enhancements. Even if one did add a new feature to them, there are so many implementations of the Parquet and ORC libraries for a multitude of languages that it is not prudent to expect applications to have the latest version of the libraries to take advantage of them. A system may be unable to decipher a newer file's contents if it uses an older version of the format's library. Avoiding interoperability issues for future data causes most systems to only support the lowest common denominator of features.

To overcome this intractability, there are recent proposals to create completely new file formats, including Meta Nimble [39, 80], Lance [37], TSFile [87], Bullion [64], and BtrBlocks [62]. We contend these formats make the same mistake as their predecessors: they are based on contemporary hardware and workload assumptions, and they do not facilitate easy extensibility to support new features that will not break interoperability with existing deployments. Even if one of them was successful at supplanting Parquet or ORC as the dominant format, we will have the same problem 10 years from now of creating another format to address their deficiencies.

Given this, in this paper we present the open-source **FUTURE-PROOF FILE FORMAT** (F3) [33]. In addition to supporting state-of-the-art encoding methods for *efficient* data access, F3 is easily *extensible* while also guaranteeing *interoperability* no matter what library version a system uses. The crux of F3 is that it defines a specification for (1) the meta-data of a file's contents, (2) the physical grouping layout of encoded data in a file, and (3) the API to access data that is agnostic to a data's encoding scheme. F3's meta-data scheme minimizes the data needed to retrieve a subset of columns in a table. Similarly, F3 rectifies the layout problems in existing formats: it relegates Parquet's concept of an all-encompassing "row group" by separating I/O, encoding, and dictionary units. It also incorporates state-of-the-art methods for cascading compression [62] and vectorized decoding [44, 48].

The most critical design component of F3 is how it addresses interoperability. F3's exposes a public API that defines how implementations decode compressed data in the file. It allows F3 to treat encoding methods as plug-ins installed and upgraded separately from its core library. To ensure that any library version can read any file, F3 embeds the decoder implementation as a WebAssembly (Wasm) binary inside the file. That is, every F3 file contains both data and the code to read that data. F3's API does not require separate implementations for the native code and the Wasm versions; the same code compiles to both targets without any changes. The plug-in-based design, coupled with including the Wasm code in each file, (1) future proofs F3, (2) avoids the problems described above, and (3) enables faster evolution than existing solutions. For example,

	Parquet, ORC	Proprietary Format	F3
Interoperability	✓✗	✗	✓
Extensibility	✓✗	✗	✓
Efficiency	✗	✓	✓

Table 1. File Format Properties – Existing DBMSs either build proprietary formats to maximize efficiency or use Parquet and ORC for *sub-optimal* interoperability and extensibility. F3 pushes the boundary of interoperability and extensibility while maintaining good efficiency.

developers can deploy future encoding methods to production systems by including the Wasm code in the files without worrying about upgrading library versions across their entire fleet. As we will demonstrate, the storage overhead of Wasm binaries is negligible (i.e., kilobytes). The decoding performance slowdown of Wasm is minimal (10–30%) compared to a native implementation.

In summary, this paper makes the following three contributions: First, we introduce the F3 specification that addresses many inefficient file layout designs in previous formats. Second, we propose a plug-in-based decoding API that allows the embedding Wasm binaries of the decoding implementation in the file, facilitating interoperability and extensibility while maintaining efficiency at the same time. Lastly, we evaluate the performance of F3's layout design considerations and the Wasm-embedding mechanism on real-world data sets. F3's efficiency matches that of the state-of-the-art formats while benefiting from the Wasm-driven decoding.

2 Background and Motivation

We first provide an overview of existing columnar file formats. We then present the motivation behind building a file format designed for interoperability, extensibility, and efficiency.

2.1 Columnar Formats

Layout: Most columnar formats, such as Parquet and ORC, adopt the Partition Attributes Across (PAX) layout [50]. The file writer partitions the tabular data horizontally into row groups. Each row group is then stored column by column, wherein each column within the row group is called a column chunk. Such a hybrid columnar layout facilitates the use of vectorized query processing by the database management system (DBMS) and keeps the tuple reconstruction overhead within a row group small. The writer typically buffers the entire row group during the write operation before committing it to disk.

The files further segment each column chunk into smaller units, called pages or compression chunks. The page is the fundamental unit for encoding, compression, and data skipping. Column chunks are dictionary-encoded by default, where the first page contains the dictionary, and subsequent pages contain encoded data. Each file incorporates a footer that stores metadata, including schema, versioning, statistics, min-max zone maps, and filtering structures. The footer's contents are serialized into a binary format using protocols like Thrift and Protocol Buffers.

Encodings: Unlike general-purpose block compression (e.g., Snappy [34], LZ4 [7], and Zstd [47]), data type-specific encoding algorithms achieve both a good compression ratio and fast decoding speed. Today's formats use multiple encoding schemes, including *dictionary encoding*, *run-length encoding* (RLE), *bitpacking*, *delta encoding*, and *frame of reference* (FOR). The file formats often support the cascading application of multiple encoding schemes on the same data [39, 44, 62, 84]. For example, a file might first encode a string column using dictionary encoding. It then further compressed the resulting dictionary codes using RLE and bitpacking.

Nested Data: Existing formats use one of the following two approaches to represent nested data: (1) the Dremel model [67] used in Parquet and (2) the Length and Presence (L&P) model used in ORC. Zeng et al. [83] studies the trade-off between the two regarding performance and space:

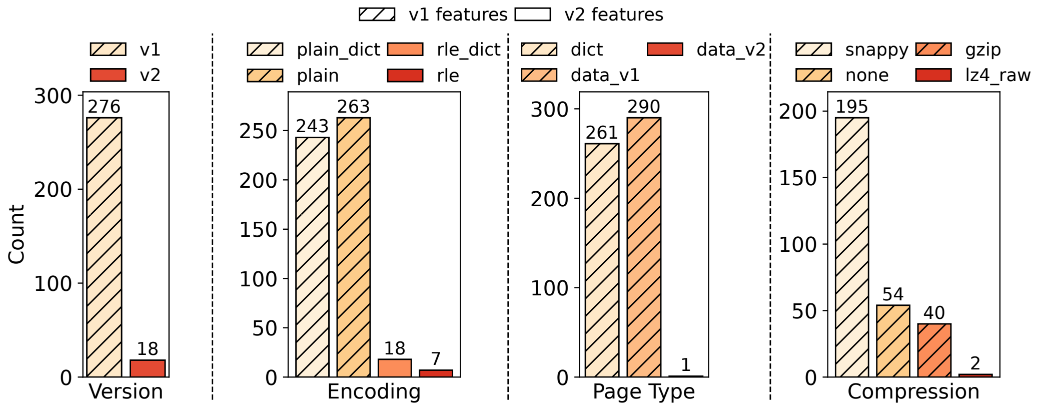


Fig. 1. Real-World Parquet Files – Features found in ~300 Parquet files collected from public object store buckets. According to the files' metadata, the software that created the files was released after 2020, but most of them only use the minimum features from Parquet v1 (2013, bars with ▨).

Dremel is faster when reading a leaf field for deeply nested data, whereas L&P has better space efficiency. Most emerging formats (e.g., Nimble [39], FastLanes [69]) adopt L&P mainly because of its easier implementation and more efficient mapping to in-memory formats like Arrow [2].

Indexing: Files can optionally include auxiliary data structures to improve query performance, such as indexes (e.g., zone maps), filters (e.g., Bloom filters), and statistical sketches (e.g., HyperLogLog [3]). Recent studies [62, 72] suggest that systems should decouple these data structures from the data files to enable independent tuning according to query patterns. As we describe in Section 3, F3 reserves space in each file for future indexing structures.

2.2 Towards Interoperability & Extensibility

We next discuss the importance of interoperability and extensibility in a file format, why these properties matter in real-world scenarios, and how they complement the need for efficient data access. Table 1 outlines the interplay between these three properties.

Interoperability: This property refers to the ability to read a file format across different programming languages, library versions, query engines, and hardware platforms. Proprietary formats optimized for specific use cases may achieve maximum efficiency for their target scenarios but often lack the broad interoperability that open file formats can provide. For example, reading data from a DBMS's proprietary internal format using another engine can only be done with an expensive ETL process, such as change data capture [54] or bulk export/import [43].

Existing open formats aspire to overcome these limitations by providing a specification that "anyone" can implement themselves. In practice, however, achieving consistent cross-language and cross-platform interoperability has proven challenging for the most popular formats (i.e., Parquet, ORC). We attribute many of these problems to the absence of rigorous day-one integration testing among the first Parquet implementations: the Impala and Parquet-Java Parquet implementations were developed simultaneously by separate teams but had several compatibility issues at the beginning. Another more recent example is how Parquet's specification added nanosecond timestamps in 2018, but they are still not supported by several major systems in 2025 (e.g., Spark, Iceberg) [6]. It requires a substantial and unsustainable development effort to maintain compatibility in diverse ecosystems as a format's specification evolves. As such, the inconsistencies between implementations hinder the adoption of new features. Addressing these challenges is critical for ensuring the long-term success of interoperable open formats.

Extensibility: The ability to easily introduce new features, particularly new encoding methods, to a file format is critical to future-proofing. As new applications emerge, there is a tendency to create a custom file format tailored for them [37, 39, 87]. Such balkanization impedes reuse across different application classes. For example, a file format designed to handle time series data will not be as effective for ML applications that require support for wide tables and encodings for high-dimensional vectors. Modern data systems are increasingly "multimodal" as applications become more complex and require holistic handling of different data in the same platform. Thus, there is a need for a file format to allow for extensibility by design to accommodate future application changes.

As summarized in Table 1, proprietary formats inherently limit user-defined extensions. Open formats support extensibility through mechanisms that allow the addition of new encodings, page types, and compression methods. However, such mechanisms of Parquet and ORC have seen limited adoption in practice.

To better understand how prevalent this problem is, we collected nearly 300 Parquet files from public object stores and examined their metadata to see what features they are using. The results from this analysis shown Figure 1 indicate that most real-world Parquet files do not use new features introduced in Parquet V2. Despite having a unified format specification, the open-source implementations of Parquet are fragmented across multiple ecosystems, including Impala [5], Parquet-Java [10], Parquet-C++ [8], Parquet-Rust [11], Parquet-Go [9], DuckDB [4], Trino [14], and numerous proprietary implementations. The community has struggled to centralize or document these efforts; for example, the official Parquet compatibility status page was only launched in 2024 and made complete recently in 2025 [40]. This decentralized ecosystem creates challenges in evolving and improving the format's features. Even when the Parquet specification introduces new features, file writers do not enable them due to fears of incompatibility with readers lacking support for these features. Consequently, developers often default to the most basic and widely supported implementations, limiting the format's practical extensibility [31].

F3 addresses the challenges of interoperability and extensibility via its decoding support (see Section 6), while still ensuring efficient data access by redesigning the file layout and utilizing the existing recent works as built-in encodings. We will present how F3 ensures efficiency first.

3 F3 Overview

We first present an overview of the F3 file format specification. Figure 2 illustrates the structure of F3, which is composed of two major parts: the Metadata Part (Section 4) and the Data Part (Section 5). The Metadata Part consists of Optional Data (OptData), Column Metadata (ColMetadata), Footer, and Postscript. The Data Part consists of Row Groups (RG), representing the actual data. We present an in-depth examination of each segment's design considerations and a comparative analysis of the unique designs that differentiate F3 from existing file formats throughout the rest of the paper.

At a high level, F3 adopts the same PAX layout as in Parquet and ORC, as shown in Figure 2. However, F3 distinguishes itself from conventional formats in several key ways. First, the Metadata in F3 eliminates the deserialization overhead from which Parquet and ORC suffer (Section 4) [80, 83]. Second, unlike Parquet, F3 decouples the physical **I/O Unit (which we call *IOUnit*)** from the logical row group so that the file writer can tune the size of the IOUnit independently according to different storage media (Section 5.1). Third, F3 decouples the dictionary-encoding scope from the logical row group. This design allows the file writer to determine the most effective scopes for

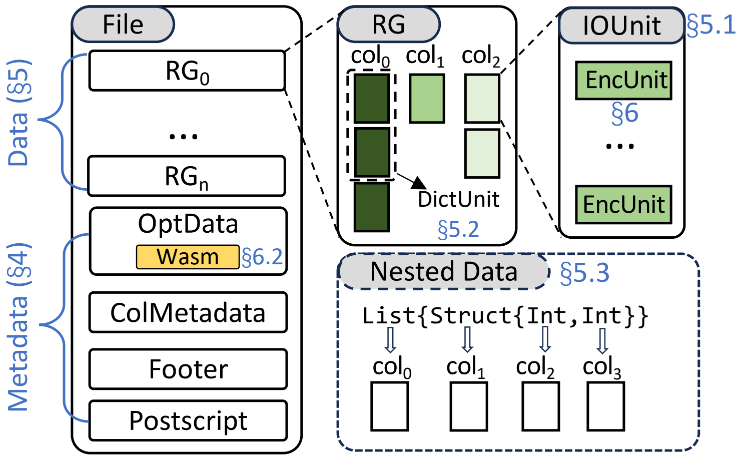


Fig. 2. F3 Format Overview – We highlight the mapping between each component of the file to the corresponding section in the paper in blue.

dictionary encoding for each individual column to achieve the best compression ratio (Section 5.2). Finally, each IOUnit consists of one or multiple **Encoding Units (EncUnit)**. Each EncUnit stores the data using the same encoding algorithm and serves as the minimal byte buffer during encoding and decoding. To address the interoperability and extensibility concerns in prior formats, we first design a general decoding API for the EncUnits (Section 6.1). We then allow file writers to include WebAssembly (Wasm) code implementing this API into the file Metadata so that any file reader can decode (arbitrary) encodings (Sections 6.2 to 6.4).

4 Metadata

The Metadata in F3 includes OptionalData, ColMetadata, Footer, and Postscript, as shown in Figure 3. OptionalData is a key-value metadata segment, currently used to store Wasm binaries (detailed in Section 6.2). It is also designed to incorporate indexes and filters. ColMetadata stores metadata for each column within each row group, including the offset and size of each IOUnit belonging to this column. ColMetadata also keeps the metadata for these IOUnits which includes their dictionary types (detailed in Section 5.2) and the metadata (e.g., size and encoding methods) for the associated EncUnits. Note that, unlike Parquet's "PageHeader" and ORC's "StripeFooter", IOUnits and EncUnits do not have headers or footers stored in place. Their metadata is centrally located in the ColMetadata segment, eliminating unnecessary I/Os when the query only requires metadata.

Prior research [64, 80, 83] has demonstrated that the datasets used in modern machine learning (ML) training workloads often lead to wide tables with files containing tens of thousands of columns, each representing a feature. A single training job, however, may require only a subset of these features. Such an access pattern incurs overhead during metadata parsing in Parquet and ORC because their serialization protocols (i.e., Thrift and Protocol Buffers) do not support random access. Consequently, the file reader must deserialize the entire metadata segment to access a specific column, a process whose cost can be comparable to reading the actual data columns [64].

To address this, F3 serializes each column's metadata into a FlatBuffer that enables zero-copy deserialization, highlighted in blue in Figure 3. The Footer is also a FlatBuffer that acts as a directory to the column-level metadata FlatBuffers. This ensures that when the reader only needs the metadata of some specific columns from some row groups, it can just read the Footer and the corresponding columns' metadata, without any deserialization.

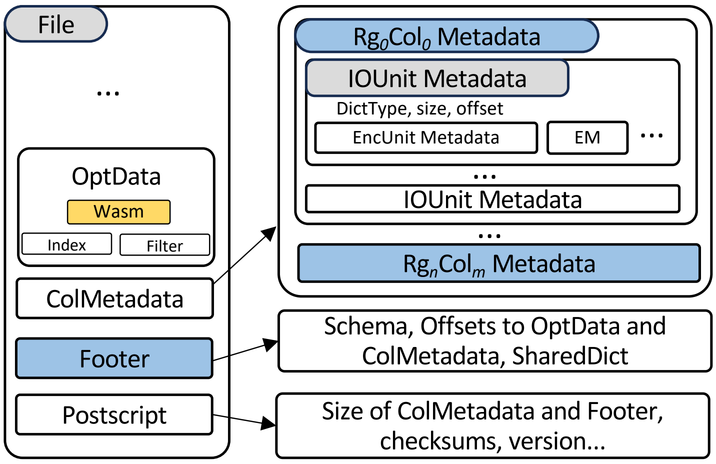


Fig. 3. Zoom-in of the File Metadata – Blocks in blue are separate FlatBuffers, allowing I/O skipping of even metadata.

The Postscript contains the sizes of ColMetadata and Footer. If the ColMetadata is small, we can read both Metadata and Footer in a single I/O request. If the ColMetadata is large, on the other hand, we can read the Footer first and then load the Metadata selectively to optimize I/O efficiency. The Postscript also stores the checksum of the data prior to the Postscript to ensure data integrity. Additionally, we allow users to store per IOUnit checksum in the IOUnits metadata, so that during reading the reader can verify the integrity of each IOUnit without the need to read the entire file.

5 Data Layout

A key design principle for the Data Layout is to allow higher flexibility in organizing the different units within a file. This principle differs from other formats that rigidly prescribe a fixed data organization tailored to specific applications.

5.1 Decoupled I/O Unit

An important design decision in F3 is to decouple the physical I/O granularity from the logical row group. The concept of a "row group" in Parquet is highly overloaded and is tied to the I/O granularity of the file. This is problematic because Parquet's file writer is not allowed to flush a partial column chunk to disk. Therefore, the writer must wait until the entire row group is buffered in memory before writing to disk. Additionally, the official documentation of Parquet and most implementations use column chunks as the unit of I/O during file reading. However, the size of a column chunk can vary drastically depending on the compressibility of the data (e.g., 50KB vs. 50MB). The file readers, thus, must implement another I/O coalescing layer to achieve optimal retrieval speed on cloud storage such as AWS S3.

F3 decouples the `IOUnit` from the row group size. A row group in F3 merely represents a logical horizontal partition to facilitate the semantic grouping of data and possibly another layer of data skipping. As shown in Figure 2, each row group consists of multiple IOUnits where the default size of each IOUnit is set to 8MB for S3— an optimal configuration for cloud object storage [55]. As demonstrated in Figure 4, F3 can flush an IOUnit whenever it reaches its capacity. For example, in step 1 column 0 buffered enough data for an IOUnit and flushes; then as data fills up in column 2 another IOUnit is flushed in step 2, and so on. This is different in Parquet because Parquet requires a column chunk to be contiguous (i.e., one column chunk per column) within a row group. With IOUnits, there is no need to buffer the entire row group before flushing to disk — a common

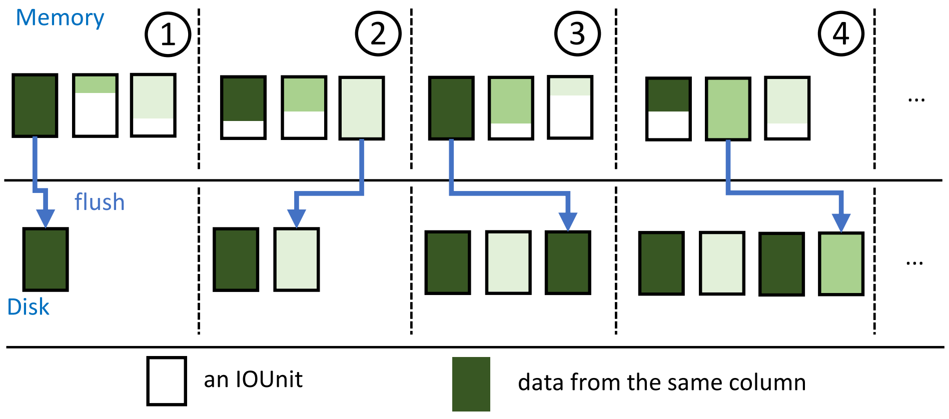


Fig. 4. F3's Writing Process – Different color means data from different columns. F3 flushes an IOUnit to disk as long as the column writer buffers enough data in memory. Parquet, on the other hand, must buffers the entire row group. The column chunk sizes thereby vary significantly, leading to non-optimal I/O sizes.

reason for OOM failures in Parquet writers. The size and offset of each IOUnit are recorded in the ColMetatdata segment, as illustrated in Figure 3.

An IOUnit comprises multiple `EncUnits`, the minimal units for encoding and decoding[1]. An EncUnit is essentially an opaque byte buffer that can be interpreted by the corresponding decoding implementation. Our prototype uses the encoding implementation in Vortex [44] as the built-in encodings, which includes more than 15 state-of-the-art encoding methods. F3 does not enable block compression by default because recent studies show that applying block compression on top of lightweight encodings may compromise the end-to-end file reading performance on modern hardware [48, 62, 65, 83]. We will discuss how to interplay with the EncUnit in detail in Section 6.

5.2 Flexible Dictionary Scope

Dictionary encoding is one of the most widely used and effective lightweight encodings in columnar formats [65, 83]. It maps values in a column to integers (i.e., dictionary codes) which are further encoded using RLE, bitpacking, etc. In Parquet and ORC, the scope of each dictionary is fixed to align with the corresponding row group. This means that each column chunk must have exactly one dictionary when dictionary encoding is enabled. In Parquet, for example, the first page of a column chunk is the dictionary page with a default size of 1MB.

Such a design is suboptimal because different column chunks could benefit from different dictionary scopes to achieve better compression. For columns with long values and a low number of distinct values (NDVs), a larger dictionary scope (i.e., fewer dictionaries) can significantly reduce the total size of the dictionaries without increasing the storage space for dictionary codes. On the other hand, a smaller dictionary scope can help columns with short values and high NDVs keep the dictionary codes short.

To motivate for flexible dictionary scopes, we present in Figure 5 the compressed sizes of three different columns from a table in the Public BI data set [81] with varying dictionary scopes from 2^{11} rows to 2^{23} rows. For example, if the column contains 1M rows, and the dictionary scope is 128K (i.e., 2^{17}), the column is divided into 8 partitions, each with its own dictionary. Both the dictionary and the codes are further compressed via BtrBlocks [62].

[1]In theory, the IOUnit can be multi-layered, e.g., 8MB IOUnit comprised of 4KB sub-IOUnits to facilitate reading on heterogeneous storage. We leave this as future work.

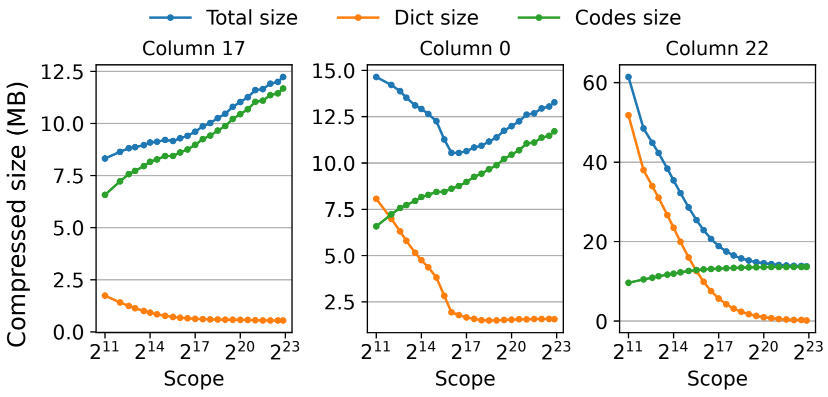


Fig. 5. Flexible Dictionary Scope – A motivating example of flexible dictionary scope: Varying dictionary scopes for three columns in the Eixo_1 table from Public BI. The dictionary and codes are separately compressed.

As shown in Figure 5, the total compressed size (i.e., encoded dictionaries + encoded codes) of Column 17 (left sub-figure) increases as the dictionary scope grows. This is because the dominating component in the total size is the encoded column (i.e., the codes) for Column 17, and a fine-grained dictionary scope helps keep the codes short. On the contrary, Column 22 (right sub-figure) exhibits the opposite trend. Because the dictionaries themselves account for a large portion of the total compressed size, Column 22 favors a more global dictionary scope to reduce the number of dictionaries. More interestingly, the optimal dictionary scope could be somewhere in the middle, as shown in the result for Column 0 (middle sub-figure). Our experiments show that of all the columns in the Public BI data set, 13% favor local dictionaries (i.e., 64K-row blocks), 35% favor a global dictionary, while the remaining 52% achieve the best compression ratio (CR) with a dictionary scope in-between. If we consider near-best CR (i.e., a CR within 5% difference of the CR from the optimal scope), global dictionaries achieve near-best CR for 67% of columns, local dictionaries for 48%, and the better of global and local scopes for 88%. This shows the opportunity to achieve better compression by adjusting the dictionary scope.

We, therefore, decouple the dictionary scope from the row group in F3. As illustrated in Figure 6, each IOUnit stores an `enum` as metadata to indicate whether it does not use Dictionary encoding (i.e., *noDict*), or uses its *local* dictionary, or a *shared* dictionary. If the IOUnit selects *local*, then both the dictionary and codes are stored in this IOUnit. If the IOUnit chooses *shared*, it will not include a local dictionary in the IOUnit. Instead, it will store an ID (`dict_id`) that maps to the dictionary in some other IOUnits. This `dict_id` to IOUnit mapping table is located in the file footer.

The shared dictionary design opens up additional opportunities: dictionaries can be shared across columns. and do not need to be identical. For example, Column 1's shared dictionary may map to IOUnits [0,1,2] while Column 2's shared dictionary maps to IOUnits [1,2,3]. The overlapping values between the two columns are stored only once in IOUnit 1 and 2.

Determining the optimal dictionary scope is a non-trivial task, as it requires estimating the compressed size of the column before encoding. In Section 7.6 we compare three simple strategies for scope selection. A detailed analysis of the scope selection process is beyond the scope of this paper. Notably, the crucial distinction between F3's design and that of Parquet is that F3 allows for a flexible dictionary scope, rather than tying it to the row group size. This flexibility makes F3 extensible for future use cases, such as when users wish to perform offline optimization on existing files.

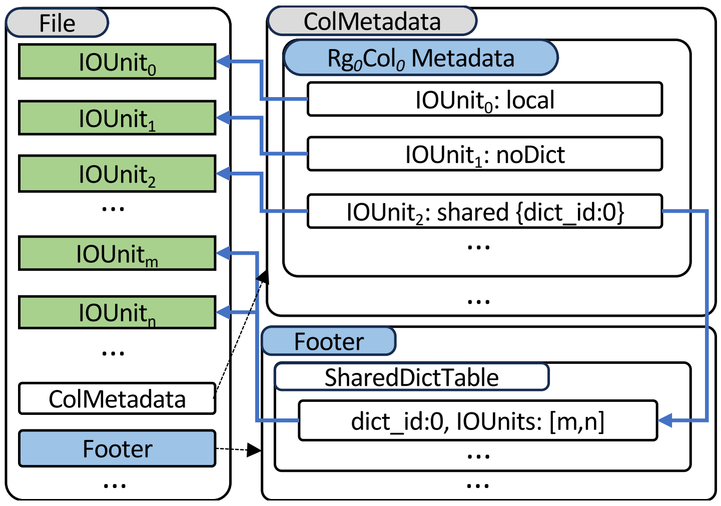


Fig. 6. Shared Dictionary Design – The file metadata records each IOUnit is using local dictionary, shared dictionary, or no dictionary. For shared dictionary, F3 stores the positions of the dictionary IOUnits in the SharedDictTable.

5.3 Nested Data Handling

The current two standard ways to handle nested data (i.e., List, Struct, and Map) are Parquet's **Dremel** model [67] and ORC/Arrow's Length and Presence (**L&P**) [68]. Prior work [68] has explored and quantitatively justified the trade-offs between these approaches: the Dremel model excels at random access to leaf fields in a schema, while the L&P model achieves smaller file sizes.

However, Zeng et al. [83] demonstrated that these differences have a negligible impact on compression ratio and file reading speed unless the schema is exceptionally deeply nested. Given that metadata for nested data occupies only a small portion of the file relative to the actual data, improvements in nested data handling are limited by Amdahl's law. Moreover, a recent study [69] revealed that most real-world nested schemas are shallow, with depths rarely exceeding four levels. On the other hand, the consistency between the in-memory and file formats is crucial for smooth adoption. Parquet's intricate Dremel model, for example, has imposed significant engineering challenges when round-tripping with Arrow, leading to limited or delayed support for nested data in many Parquet implementations [24, 36]. Recognizing Arrow's popularity in modern query engines, F3 adopts Arrow's variant of L&P model as its default mechanism for handling nested data.

6 Decoders

We now discuss F3's means for decoding data. As illustrated in Figure 2, an Encoding Unit (EncUnit) is a block of opaque bytes that stores encoded data and is the smallest granule of data reader/writer libraries interact with in an F3 file. To ensure effective interactions with EncUnits, F3 exposes two mechanisms: (1) a decoding API for accessing the EncUnit, and (2) support for embedding Wasm-based decoding kernels within the file.

6.1 Decoding API

The first consideration in designing a decoding API is the choice of output format for data. F3 uses Apache Arrow [2] due to its widespread adoption in modern data systems [13] and its support across multiple programming languages. Even other in-memory formats like Velox and DuckDB's vector

```
/// Check supported features. Examples: decode_ranges, batch_size
fn check(metadata: EncUnitMetadata) -> HashSet<Feature>;

/// Initializes the decoder.
fn init(encunit: Bytes, kwargs: HashMap<Feature, Word>) -> *mut Decoder;

/// Decodes the EncUnit into Arrow Arrays.
/// This function is called repeatedly to decode the encunit incrementally. It returns
/// `None` once all qualified values in the `encunit` have been consumed.
fn decode(decoder: *mut Decoder) -> Option<arrow::ArrayRef>;
```

Listing 1. F3 Decoding API – In Rust-style pseudocode.

format support (near) zero-copy to Arrow [1]. Such ubiquity aligns with F3's goal of interoperability. Arrow also supports random-accessible (i.e., point query) encodings like dictionary, run-end, and binary view [70] for compressed execution.

Listing 1 shows F3's decoding API for an EncUnit. It exposes three functions to (1) check what features the target EncUnit supports, (2) initialize the decoding process, and (3) decode the contents of the target EncUnit into an Arrow Array.

The *Check* function allows a library to determine which features the given target EncUnit supports. The input is the metadata of the EncUnit and the return value is a set of supported features.

The *Init* function initializes the decoder for the decoding process. The only mandatory argument is the byte sequence of the target EncUnit. The function also supports optional arguments depending on the encoding's features, as indicated by the *Check* function. For example, *decode_ranges* specifies the row ranges within the EncUnit to decode in order to facilitate row selection and skipping. Another feature (*batch_size*) defines the number of values to decode at a time; a smaller batch size (e.g., 2k) enables vectorized decoding and allows decoded data to stay in cache for further processing by the downstream query operators, as suggested by recent works [48].

This design abstracts the details of the input passed to the decoder behind the *kwargs*, making it extensible in the future. We leave the detailed specification of such *kwargs* as a future work. The API provides utility functions to check and utilize these extended capabilities. The key idea is that, even if the file reader cannot recognize the advanced functionality provided by the encoding, potentially due to compatibility issues of the *kwargs* spec, it can always fall back to the basic decoding.

Once the decoder is initialized, the file reader calls the *Decode()* function to decode the EncUnit into Arrow Arrays.

6.2 Embedded Wasm Decoders

F3's decoding API allows developers to introduce new encoding methods as "plug-ins" that can be installed or updated separately from the core library. However, requiring systems to download and install external dependencies to read a file is infeasible, especially in restricted or air-gapped environments. A better approach is to embed the decoding kernels within the file, ensuring that any system has the necessary logic to access the file's data. The challenge is enabling such embedding in a portable, safe, and efficient manner.

WebAssembly (Wasm) is a binary instruction format for a stack-based virtual machine [17, 56]. The initial goal of Wasm was to address the performance problems of JavaScript in client-side web applications: the execution performance of the leading JavaScript engines (e.g., Google's V8)

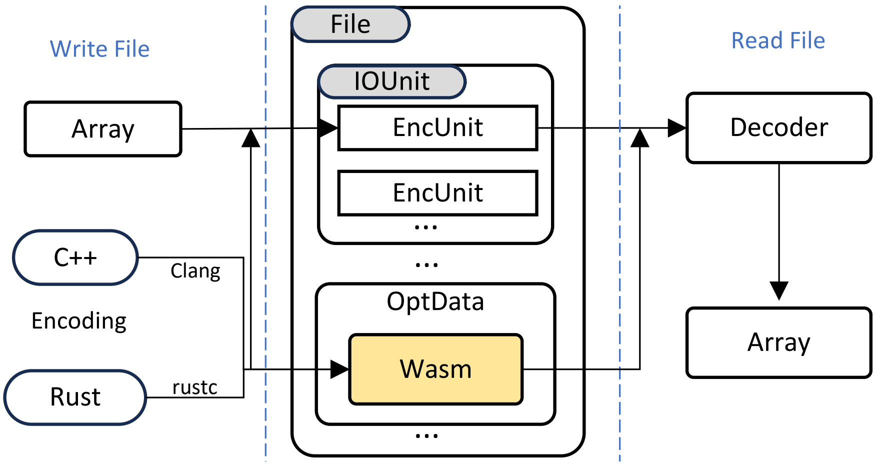


Fig. 7. Embedded Wasm Decoders – Overview of how F3 embeds Wasm decoding kernel in a file.

are orders of magnitude slower than native execution. Today, all major web browsers support Wasm, there are several standalone Wasm runtimes, and it serves as an efficient compilation target for system programming languages like C/C++, Zig, and Rust. Because web browsers operate on diverse hardware, the Wasm instruction set was designed to be portable to ensure it runs seamlessly across nearly all modern hardware and platforms while remaining language-agnostic.

The broad support and adoption of Wasm make it ideal for embedding a decoding kernel in the data file. With these embedded kernels, F3 files are completely self-contained: each file stores the encoded data and the code to interpret that data. Figure 7 shows an overview of the mechanism. When F3's writer library creates a file, it compiles each encoder kernel's corresponding decoder implementation (written in any programming language) into Wasm and stores it in the file. When reading the file, F3's core library can use either a native decoder implementation if it is available or fallback to the embedded Wasm binary to decode the data. This approach ensures that any reader can decode the data by running the Wasm code as long as the decoding API remains stable. Moreover, if the reader does have native support for a specific encoding but the native code is an outdated version with incompatible changes, the reader can still read the file using the Wasm. F3's embedded decoders eliminate the interoperability and extensibility issues seen in Parquet because any version of the reader can decode the data, and anyone can add new encodings using Wasm.

In addition to its portability, Wasm has features that are crucial for embedding binaries within a file format.

Safety: The Wasm specification is based on a memory-safe, sandboxed execution environment. One Wasm instance (i.e., a running Wasm program) can only access its isolated linear memory. It uses 32-bit addresses, thereby limiting each instance to 4 GB of memory (except for the ongoing Wasm64). If a Wasm program attempts to perform an invalid memory access outside this allocated space, the Wasm runtime immediately terminates the program. Since F3 directly embeds code in the file, it must ensure the code is not malicious, preventing harm to the system running the file reader. Wasm's safety guarantees ensure that, at worst, only the isolated 4 GB of linear memory is affected, keeping the rest of the process's memory secure.

Near-native Speed: Wasm supports modern hardware features important for efficient data decoding, including SIMD and lock-free atomic memory operators. This capability allows Wasm programs to achieve near-native speed for certain tasks. Furthermore, Wasm's binary format is optimized for streaming and parallel compilation, ensuring fast conversion from Wasm to native

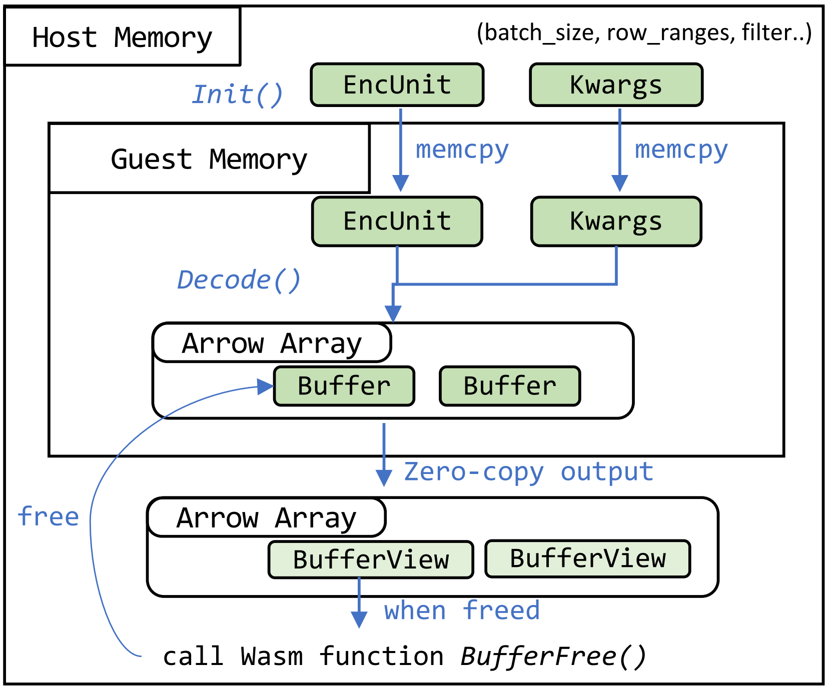


Fig. 8. Wasm Decoding Implementation – An overview of F3's decoding procedure for an EncUnit using Wasm. Buffers residing in guest memory will be freed once the reference count of the BufferView in the host gets to zero. We omit the call to *Check()* for brevity.

code. Given that columnar file reading is a performance-critical task, accounting for a significant portion of ETL pipeline time [71], this performance benefit helps F3 maintain its efficient operation even if a system only uses Wasm decoders.

Small Binary Size: Since Wasm was initially designed for web browsers, its binary format prioritizes size- and load-time efficiency to minimize startup latencies. Its instruction set uses a stack-based virtual machine, which studies show generates smaller binary sizes for programs compared to register-based implementations [56]. Smaller binary sizes are important for F3, as the size of the Wasm code impacts the overall file size after embedding.

6.3 Wasm Decoder Implementation

We next discuss how the Wasm-based decoding works in F3. We use Bytecode Alliance's Wasmtime [16], a standalone runtime for WebAssembly, for F3's Wasm-driven decoder. Figure 8 gives an illustration of the whole procedure.

Input. We first discuss the implementation considerations of the input to the Wasm-side *Init()* API call. The isolated linear memory space of Wasm instance is referred to as *guest*, while the program's address space running the Wasm instance is referred to as *host*. The input to a Wasm instance consists of the contiguous bytes of an EncUnit copied from the host's memory into the guest's memory, plus any additional runtime options.

Although research has shown the importance of minimizing the number of memory copies in analytical workloads [53, 78], we consider the memory copy while passing input to Wasm decoders hard to avoid for several reasons. First, the sandboxed linear memory restricts the guest to accessing only its own memory. Prior work has modified Wasm runtimes to allow access to host memory for reduced copying [57], but such changes compromise Wasm's security guarantees. Another approach is for the host to transfer single data items to the decoder on demand. However, such a fine-grained execution model incurs significant overhead because the runtime must convert parameters and the return value across the host's and the guest's representations [57]. A system could reduce these costs by inlining the function using link-time optimizations, but none of the current engines support this because it is non-trivial [41]. Lastly, the file reader can directly read an EncUnit into the guest memory, but this contradicts the design of decoupling I/O and decode. On the other hand, directly reading an IOUnit into the guest memory results in multiple EncUnits with potentially mixed native and (different) Wasm kernels stored in one guest memory. This requires shared memory across Wasm instances and is left as future work.

For optional arguments used in advanced features like selection and predicate pushdown, because Wasm only supports limited types as function arguments, the arguments are serialized into a bytes buffer and passed into Wasm.

Output. When the file reader retrieves output data from the Wasm decoder, a key objective is to ensure that consumers can use the output in the same way as the data produced by a native decoder, and in the meantime maintain efficiency. The first step towards this goal is that F3's output format is defined as Arrow Array in Section 6.1. However, there are challenges when passing Arrow data across the Wasm boundary. Wasm only supports limited types because it is a low-level assembly language, while Arrow's representation contains complex data structures. A straightforward approach is serializing the decoded Arrow Array into IPC stream [29], but it introduces the overhead of reassembling the Array buffers into a contiguous byte sequence for IPC output. To enable true zero-copy data transfer across the Wasm boundary, we pass the memory addresses of Arrow buffers of the output Arrow Array in *both the host and the guest* address space out from the Wasm. The file reader then reconstructs an Arrow Array using the host memory addresses of each buffer. Each buffer in this reconstructed Arrow Array is equipped with a custom deallocator, implemented as a Wasm function. This custom deallocator frees the corresponding buffer in the guest memory space when the host-side buffer is released, as shown in Figure 8. We refer to such Buffer with a custom deallocator as a "BufferView" as they reference the actual buffers residing in the guest's memory. This design ensures that consumers of the Wasm decoder's output can treat the data as identical to a native Arrow Array, enabling seamless integration with query engines. By combining zero-copy data transfer with efficient memory management, this implementation achieves both ergonomic usability and high performance.

Optimizations. The zero-copy design of output data introduces the following limitations. First, as long as the decoded Arrow Array is not freed, the associated Wasm instance cannot be freed because the "BufferView" holds a handle to it. If a separate Wasm instance is created for each decoding call, this approach could result in a proliferation of Wasm instances. This is problematic because Wasm instance creation is not cost-free; for example, the default configuration of Wasmtime on Linux requires an mmap operation to reserve linear memory in virtual memory for each instance. Excessive instance creation can significantly degrade file reading performance. Second, because the file reader uses the host address of the output buffers in the guest to reconstruct Arrow Array, the base address of the Wasm linear memory cannot be changed. Any alteration to this base address would result in invalid data access. This constraint necessitates static memory allocation for the Wasm instance, ensuring that the base address remains constant throughout its lifetime. To address those two limitations, we maintain an instance pool for each Wasm binary in the file. Each instance is created with static memory allocation, preventing changes to the base address during the instance's lifetime. Furthermore, existing instances in the pool are reused, significantly reducing the overhead associated with instance creation.

Another major extra cost of this Wasm decoding is the `memcpy` of EncUnit during the *Init()* phase. To mitigate the overhead, we reuse the guest memory that stores the bytes of EncUnit across each *Init()* call. And the overhead of Wasm function calls to allocate and deallocate this piece of memory is reduced.

6.4 Combining Native and Wasm Decoding

We now discuss how the file reader decides whether to use native or Wasm decoder. In F3, each built-in encoding has an ID and a semantic version [42]. For each encoding being used, its Wasm decoding binary along with a unique Wasm ID is written into the OptionalData section at the end of the file. The metadata of EncUnit contains the encoding ID and the Wasm ID, and the file metadata contains the semantic version of built-in encodings.

The decision process for using native or Wasm decoding is as follows. ① If the file reader recognizes the encoding ID, it first verifies whether the native code version in use is compatible with the encoding version recorded in the file metadata. If compatible, the file reader uses native code to decode the data. ② If the native code version is incompatible with the encoding version, the file reader retrieves the Wasm binary from the file using the Wasm ID and uses the Wasm API to decode the data. ③ If the encoding ID is not recognized by the file reader—indicating it is either a user-provided or a newly introduced built-in encoding—the reader directly uses the Wasm binary to decode the EncUnit. We maintain a global Wasm Module (a compiled Wasm binary) pool so that each Wasm binary is only compiled once.

6.5 Discussion

Users preferring native code performance. In case users prefer native decoding speed over Wasm, F3 plans to offer an option to associate a URL with each Wasm binary, pointing to source code or a precompiled library. When this option is invoked during file reading, the reader retrieves the code/library via the URL, optionally compiles it, dynamically links it, and employs it for decoding. This is possible thanks to the plug-in design of the decoding API. This approach, however, comes with certain limitations. ① It necessitates internet connectivity for retrieval, and the download-compile-link sequence introduces variable latency, potentially impacting initial read times. The overhead is more acceptable when amortized across multiple files sharing identical custom encodings. ② In contrast to Wasm's sandboxing, the dynamically loaded native code executes without isolation, presenting security risks if the code is untrusted or malicious. Due to these limitations, this option is included in the format specification primarily as a fallback mechanism for particular use cases rather than as the default decoding method.

Run the file reader in Wasm (e.g., browser). Our present implementation utilizes the Wasmtime runtime to execute the embedded Wasm decoding kernels within the file. However, this method is not directly applicable in environments where the file reader itself runs as a Wasm program, as conventional Wasm runtimes necessitate a native host. To enable the file reader to run in a Wasm environment, a separate code path must be introduced, allowing the file reader to be compiled for Wasm execution. Techniques similar to those used in DuckDB-Wasm [61] could be applied, where `dlopen` is used to link downloaded Wasm extensions to the main Wasm program. Another alternative is using an interpreter such as Wasm3 [46] to execute the embedded Wasm code within the file reader's Wasm program. This interpretation method, however, is likely to degrade performance. We consider this a good engineering feature and leave it as future work.

Options other than Wasm. There are other VM options to enable code plug-in, but they are not as suitable as Wasm for a file format. Java bytecode cannot match Wasm's performance due to its reliance on a heavyweight JVM with garbage collection. Native decoding libraries written in C++ and Rust can become fragile because of dependency issues. Although eBPF is designed for kernel-space execution, it lacks portability compared to Wasm, which is designed for user-space general-purpose computation. For example, eBPF's bytecode is endianness-dependent, and its bytecode size and ergonomics support are inferior to Wasm [60]. Similarly, LLVM's IR is unsuitable for long-term archival of program code as its specification is less stable than Wasm's.

Wasm version upgrade. Wasm ensures backward compatibility but does not guarantee forward compatibility: older clients may fail to interpret Wasm binaries with new features. Achieving unconditional interoperability is nearly impossible because no VM code format can fully anticipate all future advancements from day one. Despite this, Wasm stands out as the best candidate for a self-contained file format, thanks to its robust community support from web browsers. Upgrading the bundled Wasm runtime/client is much simpler than integrating an extension into every file reader implementation. Moreover, advanced Wasm features [15] often exceed the needs of decoding kernels, so the expected frequency for Wasm client version upgrade is low.

Security concerns. Despite the sandbox design of Wasm, it still has vulnerabilities, especially with the discovery of new attack techniques. For example, side-channel attacks (e.g., timing or cache attacks) can leak sensitive information across sandbox boundaries to reveal information about the hardware or memory access patterns. We believe there are opportunities for future work to improve Wasm security. One approach is for creators of Wasm decoding kernels to register their Wasm modules in a central repository to get the Wasm modules verified and tamper-resistant.

Parallelism in Wasm. In F3, the thread parallelism unit for compute and I/O tasks is designed to be EncUnit and IOUnit, respectively, which is another reason F3 hierarchically splits the file layout into these components. Therefore, although Wasm is developing its Threads feature [15], it is unlikely for our decoding kernels to adopt it. For data parallelism, Wasm already provides standardized SIMD support [15].

7 Evaluation

In this section, we analyze each of the F3 features presented. In addition to verifying the efficacy of the proposed features, this evaluation also examines whether the overheads of F3 components are within reasonable tolerances. Section 7.1 describes the experimental setup. Section 7.2 explores the performance of F3 and comparable formats as wide table projections. Section 7.3 evaluates compression-related behavior, as both the compression ratio and decompression speed, of comparable formats under various workloads. Section 7.5 evaluates the benefit of decoupled I/O units. What the flexible dictionary scope can achieve is evaluated in Section 7.6. Lastly, we systematically evaluate the idea of embedding Wasm in the file in Section 7.7.

7.1 Experimental Setup

We run the experiments on a server with 2 × Intel(R) Xeon(R) Platinum 8474C CPUs, 1TB DRAM, and an Intel P5620 NVMe SSD. The operating system is Debian 12. For all experiments, we use the following configurations of the formats (unless specified otherwise). Our prototype of F3 is implemented using nightly Rust. For the comparison with other formats, we use Parquet-rs (v53.2.0),

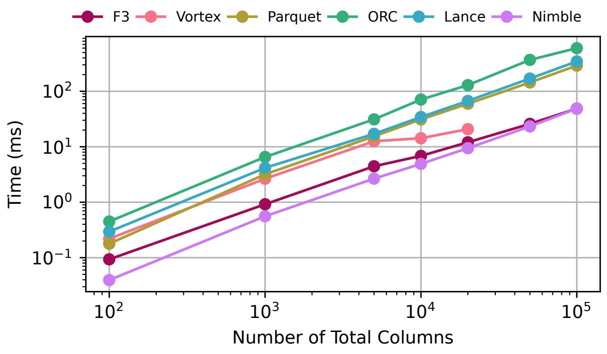


Fig. 9. Metadata Overhead – Time spent on metadata under a varying number of total columns when projecting 10 columns out. Note that the missing points for Vortex are fixed in the latest version of Vortex.

ORC C++ (v2.1.1), Lance v0.20.0, Vortex v0.21.1[2], and Nimble commit a135e2f. We also tested against BtrBlocks, but BtrBlocks' plain string format is fixed-size 8-byte offset and length, leading to zero-copy during string decoding, while other formats decode string into Arrow. BtrBlocks also only supports limited data types, resulting in several data sets being unable to be tested on it. It also lacks metadata support for testing projection efficiency. We therefore did not include it to ensure fairness. All the files are generated with the default configurations. We also explore different configurations of Lance, but they do not significantly impact the overall performance comparison. When measuring file scan time or decompression throughput, we clear the OS page cache and scan the file from the SSD. Each reported measurement is the average of five runs per experiment.

7.2 Metadata Overhead

We first test the metadata overhead under the case of wide-table projection in ML training workload. We generate a table of 64K rows with a varying number of float attributes. We store the table in different formats and randomly select 10 columns to project. Figure 9 shows the time spent on metadata.

As the number of columns in the table grows, the metadata parsing overhead increases linearly for all the formats. This is because, regardless of whether the metadata supports random access, there is always some overhead in the footer (e.g., schema) whose size is proportional to the number of columns. Nimble is the fastest due to its minimal use of FlatBuffers, which solely store row group offsets; metadata below this level employs a custom serialization format. Other formats using FlatBuffer require a verification [32] of the FlatBuffer by default, resulting in slower performance than Nimble. This verification is to prevent out-of-bounds read, undefined behavior, and security issues when random-accessing the FlatBuffer. When disabling this verification, F3 has lower metadata overhead than Nimble on all the data points. Using FlatBuffer makes the metadata specification transparent and easy to evolve compared to hand-written metadata serialization. F3 surpasses other FlatBuffer-based formats because F3 not only supports reading a column's metadata without deserialization, but also supports skipping I/O for the metadata of a specific column. For example, Lance currently has to read and deserialize all the metadata, despite using FlatBuffer, making it the slowest among the FlatBuffer-based ones [38]. Vortex allows skipping the deserialization step for irrelevant metadata sections, but still requires performing the I/O to read

[2]Vortex is fast evolving and has made many changes on both encodings and layout since this version. We recommend the readers to checkout the latest version of Vortex.

them. Even with thousands of columns, the metadata parsing overhead is still around 10ms in F3, an order of magnitude faster than Parquet.

7.3 Compression Ratio and Read Throughput

In this experiment, we verify that F3 has not sacrificed efficiency, as compared to other formats, for its extensibility benefits. The evaluation utilizes eight datasets: ClickBench, the TPC-H SF10 `lineitem` table, and six additional datasets derived from the benchmark suite presented in [83]. These six datasets represent diverse workload characteristics (core, bi, classic, geo, log, and ml) and incorporate data distributions sourced from public BI benchmarks, ClickHouse examples, Geonames, Edgar Logs, and UCI-ML, as detailed in [83].

Figure 10 presents the results. F3 achieves comparable or superior decompression throughput to Parquet, albeit with a slightly worse compression ratio. This is because F3 utilizes the open-sourced Rust implementation of BtrBlocks and FastLanes. The file-reading throughput is faster than Parquet and ORC thanks to vectorized decoding. The slightly higher compression ratio can be attributed to the absence of block compression in F3. Nimble, which also employs cascade encoding and vectorized decoding, optimizes more aggressively for compression ratio at the expense of read throughput compared to F3. F3's compression ratio is slightly higher than Vortex because F3 does not use the Vortex type system and must store extra data type information to bridge the two. The missing bar for ORC in ClickBench is because of unsupported data types in ORC.

7.4 Random Access

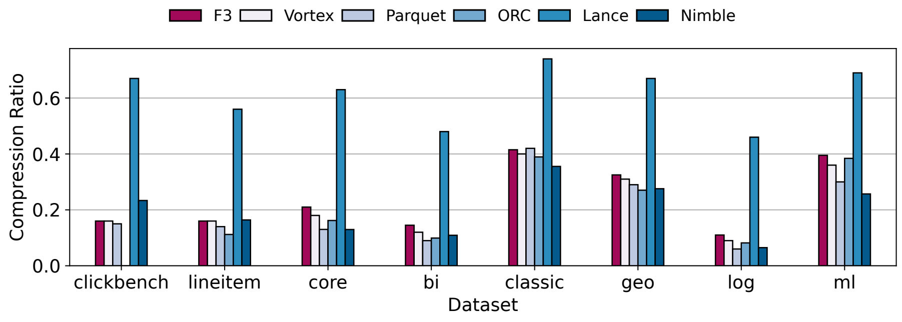
To measure the random access performance across different formats, we utilize the data sets in Section 7.3. For each dataset and format, 10 rows were randomly selected. The access latency for each row was measured independently, and the average latency across these 10 accesses is reported. The results are presented in Figure 10c. Lance is the fastest because it does not have cascading encoding or compression like the others, enabling direct offset calculation for certain types (e.g., integers) and minimizing read amplification. F3 achieved the second-best performance. The decoupled design of IOUnit can reduce read amplification without affecting compression ratio and read throughput, and the built-in encoding allows random access without full decoding. Vortex is slower than F3 because it has a relatively large prefetch of its footer despite accessing a single row[3]. Parquet incurs the highest latency as it requires decoding the entire row group to access a single row. ORC and Nimble employ row indexes to skip portions of a row group, but they still need to fully decode an EncUnit to random access the data.

This experiment and Sections 7.2 and 7.3 together give an overview of the performance trade-offs between all the formats, and show that F3 is on par with other formats on efficiency, despite their differing optimization objectives.

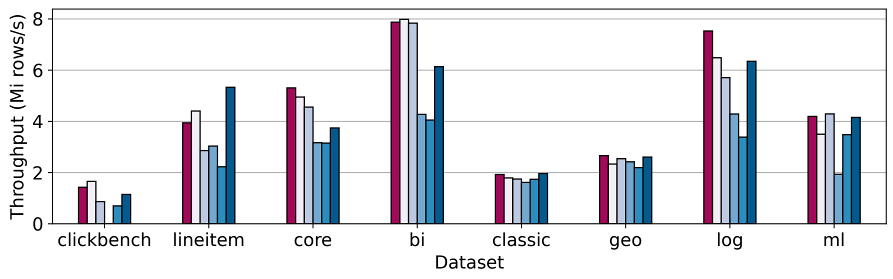
7.5 Decoupled IOUnit

This experiment evaluates the advantage of F3's separation of the I/O unit from the logical row group size. Two datasets were employed: a standard OLAP dataset "core" [83] and the Laion-5B dataset [73], which includes both tabular attributes and vector embeddings. The latter exemplifies data types prevalent in modern ML workloads [83]. We varied the row group size (in number of rows) for Parquet and F3, measuring the corresponding peak memory consumption during data ingestion. Lance does not have the concept of a row group. ORC and Nimble limit the size of row groups by uncompressed byte size (e.g., 256MB). Consequently, row group size variation was not

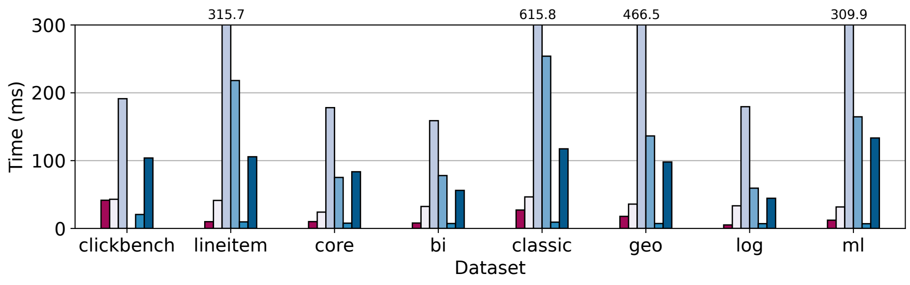
[3]This is fixed in the latest Vortex version.



(a) Compression Ratio: File size over in-memory Arrow size



(b) File-reading Throughput (from NVMe SSD)

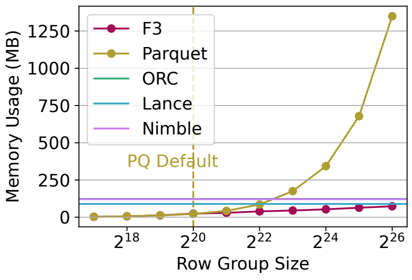


(c) Random Access: Time to read a random row out

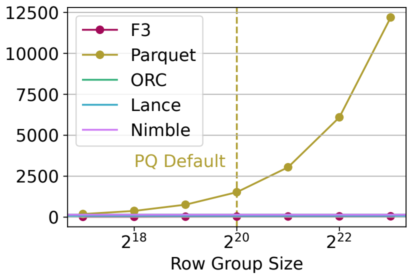
Fig. 10. Compression Ratio, Read Throughput, Random Access

performed for these formats; results are reported using their default configurations. Vortex was excluded as its write memory footprint is determined by the writer's input batch size [45].

Figure 11 shows the results of this experiment. Parquet's memory usage grows proportionally to the row group size because it must buffer the whole row group before writing to disk. On the default row group size (1M) in Figure 11b, Parquet's memory usage is as large as 2GB because the data set contains 768-dimension vector embeddings. This makes the value size of the embedding columns much larger than the other ordinary tabular columns. Aligning the row group size to the flushing buffer size in this case results in high memory usage. F3 and Lance, on the other hand, flush an IOUnit as long as the column buffers enough data that matches an IOUnit size, so they incur a low memory usage during writing. While Nimble and ORC manage write memory by limiting the physical size of a row group, this can yield row groups with fewer rows for wide tables or large values. This fragmentation negatively impacts read performance by creating smaller column chunks. As detailed in Table 2, ORC and Nimble often yield small, non-contiguous chunks whose size



(a) core



(b) Laion

Fig. 11. Decoupled IOUnit – The peak memory usage for the file formats during file creation.

Column	F3	Parquet	ORC	Lance	Nimble
url	8.8MB	75MB	1.0MB	11.2MB	1.0MB
similarity	8.2MB	6.0MB	81KB	4.1MB	92KB

Table 2. Average Column Chunk Size (Laion) – Showing two selected columns from the Laion data set.

depends on the compressed size of the value in the column, whereas F3 provides consistent chunk sizes (~8MB). Consequently, projecting the "similarity" column using Nimble is 40% slower than F3. This evaluation demonstrates that F3 effectively decouples the logical row group partitioning parameter from the physical I/O unit size, enabling independent tuning and resulting in consistent write memory consumption and predictable read chunk sizes across heterogeneous datasets.

7.6 Flexible Dictionary Scopes

To verify the benefit of the flexible dictionary scope, we select the first file of each table from the Public BI data sets and extract all columns, resulting in 2080 columns in total. For each column, we apply three different scope selection strategies: Local, Global, and GLBest. GLBest is a sampling-based strategy to decide whether to use Local or Global: it randomly samples a part of the column, uses Local and Global [4] to encode the sampled data respectively, and picks the strategy with a smaller encoded sample size to encode the column. The proportion of the sample to the total column size is referred to as Sample Rate (SR). We use an SR of 1% from BtrBlocks [62], and also try an SR of 100%, i.e., encode using both global and local scopes and preserve the smaller one.

We record the Compression Ratio (CR) and encoding time of each strategy on each column. To better visualize the results, we show the relative numbers of Global and GLBest compared with Local. Table 3 shows the average relative CR and relative encoding time, where a larger SR trades encoding speed for compression ratio. The CDF of relative CR is shown in Figure 12. Using 1% sampling, there are columns where the global dictionary is chosen even though the local dictionary performs better. However, in most cases, the best scope is selected. The above results demonstrate that F3's flexible dictionary scope offers a new opportunity to select the best dictionary scope based on the application's needs.

7.7 Wasm Decoder Evaluation

In the following four experiments, we seek to answer:

- Can the speed of the decoding algorithm shipped in Wasm match the native code performance?
- How large is the Wasm bytecode shipped in the file, and how long does it take to AOT-compile the Wasm binary to native code?

[4] Global requires building a global dictionary over all the data first.

	Avg. Relative CR	Relative Enc. Time
Global	92.2%	137%
GLBest (SR 1%)	90.4%	225%
GLBest (SR 100%)	89.7%	252%

Table 3. Flexible Dictionary Scope – Average relative CR and relative encoding time.

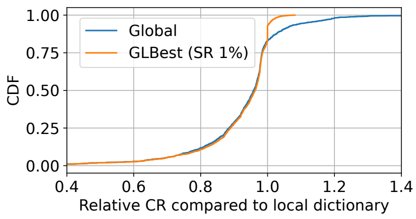


Fig. 12. CDF of Relative CR

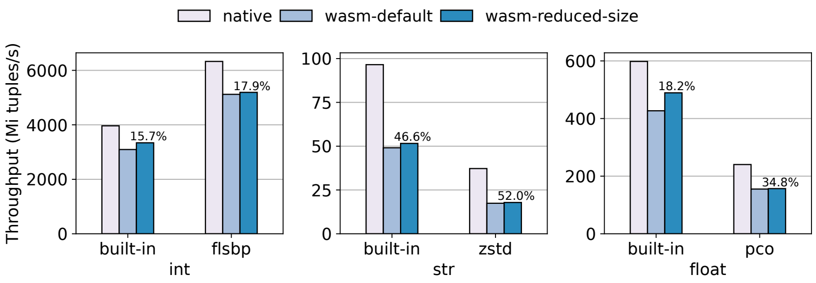


Fig. 13. Wasm Microbenchmark – The slowdown of wasm-reduced-size compared with native is annotated in the figures.

– What is the end-to-end file reading performance if we decode the entire file using the built-in (Vortex) encodings in Wasm?
– How does embedding Wasm enable extensibility for the future?

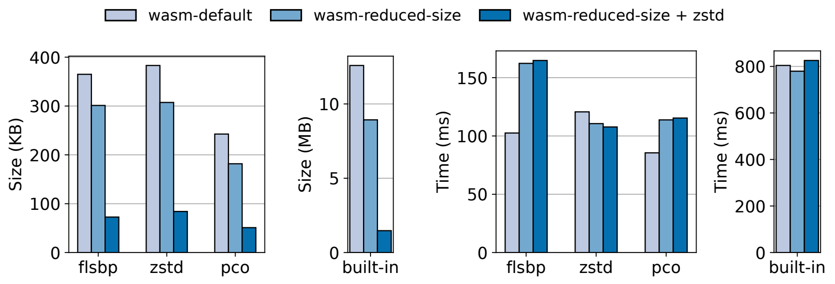
Wasm Microbenchmark. In this experiment, we analyze the performance gap between native and Wasm decoding kernels. Specifically, we evaluate three schemes—Pco [12], Zstd [47], and FastLanes-Bitpacking (flsbp) [48]—to measure the decoding performance for float, string, and integer data, respectively. For comparison, we also include the built-in encodings (Vortex) in F3.

Decoding performance is measured across three target profiles: native, wasm-default, and wasm-reduced-size. The *native* target is compiled in release mode with `target-cpu=native` enabled, allowing decoding algorithms to fully utilize hardware-specific optimizations (e.g., SSE, BMI, AVX). The *wasm-default* target is compiled with `target-feature=+simd128` to enable SIMD acceleration, while the *wasm-reduced-size* target applies additional compilation flags to minimize Wasm size.

As shown in Figure 13, the Wasm-based decoding kernels experience a slowdown of up to 46% when decoding strings using F3's built-in encoding. This higher overhead is attributed to the larger size of string columns, which leads to increased `memcpy` overhead when transferring input data from the host to the guest (Section 6.3). For integer and float decoding, the Wasm slowdown ranges from 15% to 35%. Interestingly, the wasm-reduced-size profile outperforms the wasm-default profile slightly, likely due to better instruction cache utilization.

Overall, while Wasm-driven decoding kernels exhibit a performance trade-off compared to native implementations, the slowdown is within an acceptable range given the significant interoperability and extensibility benefits offered by the Wasm mechanism. These advantages will be further demonstrated in subsequent sections.

Wasm Binary Size & Compilation Time. We further investigate the trade-off between Wasm binary size and Ahead-of-Time (AOT) compilation time for the same four schemes used in the prior experiment under the two Wasm targets. Additionally, for the wasm-reduced-size target, we apply Zstd compression to assess the potential for further reducing the Wasm binary size.



(a) Size of the Wasm binaries **(b)** Time to AOT compile the binaries

Fig. 14. Wasm Binary Size & Compilation Time – The size of the Wasm binaries of three decoding algorithms under varying Wasm compilation configurations, along with the time to Ahead-Of-Time compile the Wasm binaries to executable code.

Figure 14a demonstrates that the Wasm binaries of the three extension encodings are small and highly compressible. The findings indicate that introducing additional encoding Wasm binaries incurs minimal overhead in terms of file size. The current implementation of the built-in encoding includes over 15 cascade encodings from Vortex, contributing to a relatively large Wasm binary size. However, applying Zstd compression reduces the binary size to approximately 1 MB, highlighting the effectiveness of applying standard compression techniques to Wasm binaries.

Figure 14b presents the AOT compilation time, showing that the three extension encodings have low compilation overhead. The Wasm of built-in encodings, on the other hand, exhibits relatively high compilation overhead. Manual inspection reveals that the compiled library contains a substantial amount of unused code, suggesting opportunities to further reduce the Wasm binary size by pruning unnecessary components. Moreover, when processing a large dataset comprising multiple files, the checksum of the Wasm binary can be utilized to ensure that AOT compilation is performed only once, thereby amortizing the additional compilation cost.

Overall, the added size of Wasm binaries is unlikely to constitute a significant portion of the total size. Similarly, the observed compilation times remain within acceptable bounds. Lastly, given the demonstrated compressibility of Wasm binaries, we plan to explore compression techniques tailored to the Wasm binary format to further enhance performance and efficiency in future work.

End-to-end File Scanning. We evaluate the performance trade-off between the Wasm and native versions of decoding kernels in an end-to-end file scanning setting, using the wasm-reduced-size profile. In this experiment, the Wasm of the built-in encodings is AOT-compiled prior to loading the kernel into memory for file scanning. The results, shown in Figure 15, reveal that the use of Wasm introduces a 9.6% slowdown on the core dataset and up to a 25% slowdown on the classic dataset. Despite this performance overhead, we consider the slowdown to be reasonable in light of the significant interoperability benefits offered by Wasm. As the built-in encodings evolve, old file readers can still read newer files.

Wasm Extensibility. This experiment demonstrates how the Wasm embedding design of F3 enables unique opportunities for domain-specific datasets and ensures future-proof extensibility for incorporating new encodings. To illustrate, we use two synthetic datasets with distinct distribution patterns: the Linear pattern from LeCo [66], representing serially correlated time-series data, and the Sparse Feature pattern from Bullion [64], characterized by repeated window patterns common in machine learning training. These patterns are rarely addressed by mainstream encoding methods.

In addition to the built-in encodings, we incorporate variants of LeCo and Bullion's encodings as Wasm binaries (denoted as F3-custom-wasm) to encode the datasets. As shown in Figure 16a,

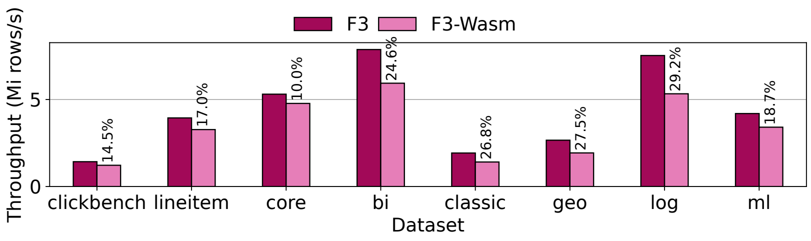
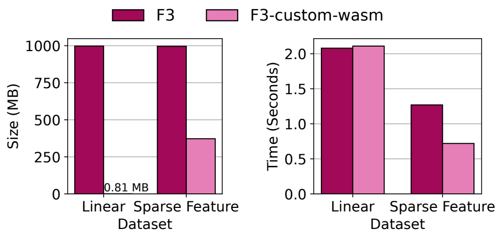
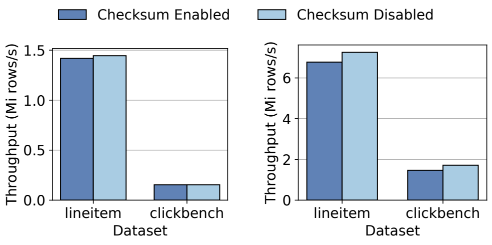


Fig. 15. End-to-end File Scanning – Comparison of scan performance between native and Wasm using the built-in encodings. Slowdown of Wasm over native is annotated on the F3-Wasm bars.



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

Fig. 16. Wasm Extensibility – Embedding domain-specific encoding methods in the file leads to significant file size reduction with negligible space overhead.



(a) Write Throughput **(b)** Read Throughput

Fig. 17. Data Integrity Overhead – The performance regression in F3 when calculating and verifying IOUnit checksums is small.

introducing these custom encodings significantly reduces the overall file size, while the additional Wasm binary size for the two custom encodings is approximately 150 KB—negligible compared to the achieved size reduction. Figure 16b shows that the file scan time is either faster or on par with the built-in encodings because the custom Wasm is domain-specific.

Furthermore, the general API design and Wasm mechanism facilitate seamless integration of custom encodings without requiring changes to the file format specification. This ensures that as encoding algorithms and data distribution patterns evolve, the F3 specification can remain stable, avoiding the compatibility challenges previously encountered with formats like Parquet.

7.8 Data Integrity Overhead

To mitigate the risk of silent data corruption propagating to column read operations, supplementary integrity validation mechanisms are required. F3 addresses this by enabling the computation of a checksum for each IOUnit during the write phase and verifying it upon reading. Figure 17 shows the performance regression on calculating and verifying the IOUnit checksum. Our analysis indicates that these integrity checks introduce negligible overhead.

8 Related Works

Existing Columnar File Formats: Among the existing file formats, Parquet and ORC are the popular ones, and have been thoroughly studied [65, 83] and demonstrated the need for them to be improved. Lance [37], Nimble [39] and Bullion [64] are newer formats that appear to address the specific use cases of columnar formats for machine learning. Lance's layout has a similarity to F3 as both the formats separate I/O unit from the row group. However, F3 still preserves the row group and allows the layout to be more flexible in dictionary scope and encodings.

Apache TsFile [87] is a format specifically designed for IoT use cases. It organizes data by series so data from the same device are stored in the same row group. However, since each device has a different schema, the schemas across different row groups are different. This is not easy to achieve in a general-purpose file format, but can be better achieved in an upper layer like [21, 22, 51].

Among the academic ones, BtrBlocks [62] mainly focuses on the strategy of cascading encoding, and FastLanes [48, 49] focuses on making integer decoding fully parallel via SIMD or SIMT. Frequency–Store [76] propose to decompose images into columns in frequency domain and store the images in columnar layout to serve various Image AI workloads.

Vortex [44] is a SOTA Rust implementation of the techniques in BtrBlocks and FastLanes. It includes both a compressed in-memory Array representation for compressed query execution, and a file format to serialize the compressed arrays. To not reinvent the wheel, we reuse the encodings in Vortex as the built-in encodings in F3. On the flip side, based on our discussions with Vortex, they have an extensible file layout and plan to integrate Wasm decoders into their design in the future.

File Format Extensibility. Extending the capabilities of many contemporary file formats often mandates modifications to their underlying specification or codebase. For instance, incorporating a new encoding in Nimble requires developers to implement the encoding in C++ as a derived class in the Nimble source repository, followed by the merge of the pull request. Additionally, every engine or library that uses Nimble must be upgraded to the latest version to support reading files with the new encoding. In contrast, F3's Wasm plug-in design allows users to seamlessly integrate encoding extensions—authored in any language that compiles to Wasm —specifically tailored to their data characteristics. Crucially, other F3 readers can process data with these custom encodings without requiring software updates or modifications to the core F3 repository.

Using Wasm in databases. Our work is not the first attempt to utilize the capabilities of Wasm to boost a DBMS (component). DuckDB has a Wasm version that outperforms other data analytics libraries on the Web [61]. Umbra has attempted to use Wasm to allow users to write User-Defined-Operators in any language [74]. Haffner and Dittrich explored using Wasm as an IR in a query compilation engine [57]. Sok et al. showed the overhead of Wasm-based serverless join [77] and analyzed the reasons. However, none of these works explored using Wasm as an extension and interoperability mechanism in a file format.

9 Conclusion

This paper presents the F3 project, a columnar file format that achieves interoperability, extensibility, and efficiency at the same time. F3 challenges the traditional columnar format layout design to maximize the efficiency a format can achieve, while at the same time allowing extensibility for the future. By defining a general decoding API and including the Wasm binaries of decoding algorithms in the file, F3 achieves interoperability and extensibility brought by the Wasm ecosystem. Evaluation of a prototype of the format shows that the layout design benefits efficiency and the Wasm mechanism allows the format to extend well with little overhead.

Acknowledgments

We thank Hannes Mühleisen for inspiring us to utilize Wasm for decoding. This work was supported (in part) by Shanghai Qi Zhi Institute, and Google DAPA Research Grants.

CMU-DB vs. MIT-DB: #1 ♙e4

References

[1] 2024. Aligning Velox and Apache Arrow: Towards composable data management. https://engineering.fb.com/2024/02/20/developer-tools/velox-apache-arrow-15-composable-data-management/.
[2] 2024. Apache Arrow. https://arrow.apache.org/.
[3] 2024. DuckDB Format. https://duckdb.org/docs/guides/performance/file_formats.html. Accessed: 2024-12-04.
[4] 2024. DuckDB's Parquet Implementation. https://github.com/duckdb/duckdb/tree/main/extension/parquet.
[5] 2024. Impala's Parquet Implementation. https://github.com/apache/impala/tree/master/be/src/exec/parquet.
[6] 2024. InfluxDB video: "Parquet is a standard like SQL is a standard". https://lists.apache.org/thread/tnxbykozo5owq2y36nw7lomr91hrdxhz.
[7] 2024. LZ4 Flex. https://github.com/PSeitz/lz4_flex.
[8] 2024. Parquet C++ Implementation. https://github.com/apache/arrow/tree/main/cpp/src/parquet.
[9] 2024. Parquet Go Implementation. https://github.com/apache/arrow-go/tree/main/parquet.
[10] 2024. Parquet Java Implementation. https://github.com/apache/parquet-java/.
[11] 2024. Parquet Rust Implementation. https://github.com/apache/arrow-rs/tree/main/parquet.
[12] 2024. Pcodec. https://github.com/mwlon/pcodec.
[13] 2024. Projects Powered By Apache Arrow. https://arrow.apache.org/powered_by/.
[14] 2024. Trino's Parquet Implementation. https://github.com/trinodb/trino/tree/master/lib/trino-parquet.
[15] 2024. Wasm Feature Extensions. https://webassembly.org/features/. Accessed: 2024-11-29.
[16] 2024. Wasmtime. https://wasmtime.dev/.
[17] 2024. WebAssembly. https://webassembly.org/.
[18] 2025. Apache Carbondata. https://carbondata.apache.org/.
[19] 2025. Apache Hadoop. https://hadoop.apache.org/.
[20] 2025. Apache Hive. https://hive.apache.org/.
[21] 2025. Apache Hudi. https://hudi.apache.org/.
[22] 2025. Apache Iceberg. https://iceberg.apache.org/.
[23] 2025. Apache Impala. https://impala.apache.org/.
[24] 2025. Apache mailing list-Coordinating / scheduling C++ Parquet-Arrow nested data work. https://lists.apache.org/thread/wyr53b94fjwfxgynn60bbprpxztqzdym.
[25] 2025. Apache ORC. https://orc.apache.org/.
[26] 2025. Apache Parquet. https://parquet.apache.org/.
[27] 2025. Apache Presto. https://prestodb.io/.
[28] 2025. Apache Spark. https://spark.apache.org/.
[29] 2025. Arrow IPC format. https://arrow.apache.org/docs/format/Columnar.html#serialization-and-interprocess-communication-ipc.
[30] 2025. Dremio. https://www.dremio.com//.
[31] 2025. DuckDB Blog: Query Engines: Gatekeepers of the Parquet File Format. https://duckdb.org/2025/01/22/parquet-encodings.html.
[32] 2025. Flatbuffers Verifier. https://github.com/google/flatbuffers/blob/master/rust/flatbuffers/src/verifier.rs.
[33] 2025. Future File Format (F3). https://github.com/future-file-format.
[34] 2025. Google snappy. http://google.github.io/snappy/.
[35] 2025. InfluxData. https://www.influxdata.com/.
[36] 2025. Jira-Read and write nested Parquet data with a mix of struct and list nesting levels. https://issues.apache.org/jira/browse/ARROW-1644.
[37] 2025. Lance. https://github.com/eto-ai/lance.
[38] 2025. Lance read all metadata code. https://github.com/lancedb/lance/blob/039c6c65c92f5e606fe1431060212a9b1f7becc5/rust/lance-file/src/v2/reader.rs#L447.
[39] 2025. Nimble. https://github.com/facebookincubator/nimble/.
[40] 2025. Parquet Implementation Status Page. https://parquet.apache.org/docs/file-format/implementationstatus/.
[41] 2025. Personal Discussion with Wasm maintainers.
[42] 2025. Semantic Versioniong. https://semver.org/.
[43] 2025. Snowflake Data Unloading. https://docs.snowflake.com/en/user-guide/data-unload-overview.
[44] 2025. Vortex. https://github.com/spiraldb/vortex/.
[45] 2025. Vortex Writer Flush Code. https://github.com/spiraldb/vortex/blob/6187ebb4ee130be56404342e21cddadcb69c7215/vortex-file/src/writer.rs#L90.
[46] 2025. Wasm3. https://github.com/wasm3/wasm3.
[47] 2025. Zstandard. https://github.com/facebook/zstd.

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

[49] Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. 1–11.

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

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

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

[53] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[54] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao, Jemiah Westerman, Phanindra Ganti, Boris Shkolnik, Sajid Topiwala, Alexander Pachev, Naveen Somasundaram, and Subbu Subramaniam. 2012. All aboard the Databus! Linkedin's scalable consistent change data capture platform. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. Article 18, 14 pages. doi:10.1145/2391229.2391247

[55] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2769–2782.

[56] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[57] Immanuel Haffner and Jens Dittrich. 2021. Fast compilation and execution of SQL queries with webassembly. *arXiv preprint arXiv:2104.15098* (2021).

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

[59] Brian Hentschel, Michael S Kester, and Stratos Idreos. 2018. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data*. 857–872.

[60] Wenjun Huang and Marcus Paradies. 2021. An evaluation of webassembly and ebpf as offloading mechanisms in the context of computational storage. *arXiv preprint arXiv:2111.01947* (2021).

[61] André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. 2022. DuckDB-wasm: fast analytical processing for the web. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3574–3577.

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

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

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

[65] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.

[66] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1, Article 65 (mar 2024), 28 pages. doi:10.1145/3639320

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

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

[69] Ziya Mukhtarov. 2024. *Nested Data-Type Encodings in FastLanes*. Master's thesis. TECHNICAL UNIVERSITY OF MUNICH.

[70] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[71] Pedro Pedreira, Deepak Majeti, and Orri Erling. 2024. Composable Data Management: An Execution Overview. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4249–4252.

[72] Martin Prammer, Xinyu Zeng, Ruijun Meng, Wes McKinney, Huanchen Zhang, Andrew Pavlo, and Jignesh M. Patel. 2025. Towards Functional Decomposition of Storage Formats. In *Conference on Innovative Data Systems Research (CIDR)*.

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

[74] Moritz-Felipe Sichert. 2024. *Efficient and Safe Integration of User-Defined Operators into Modern Database Systems.* Ph. D. Dissertation. Technische Universität München.

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

[76] Utku Sirin, Victoria Kauffman, Aadit Saluja, Florian Klein, Jeremy Hsu, and Stratos Idreos. 2025. Frequency-Store: Scaling Image AI by A Column-Store for Images. (2025).

[77] Chanattan Sok, Laurent d'Orazio, Reyyan Tekin, and Dimitri Tombroff. 2024. WebAssembly serverless join: A Study of its Application. In *Proceedings of the 36th International Conference on Scientific and Statistical Database Management*. 1–4.

[78] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Efficient property graph queries in an analytical RDBMS. In *Proceedings of the Conference on Innovative Data Systems Research. https://www. cidrdb. org/cidr2023/papers/p66-wolde. pdf*.

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

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

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

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

[83] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment* 17, 2 (2023), 148–161.

[84] Xinyu Zeng, Ruijun Meng, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2024. NULLS!: Revisiting Null Representation in Modern Columnar Formats. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, Carsten Binnig and Nesime Tatbul (Eds.). ACM, 10:1–10:10. doi:10.1145/3662010.3663452

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

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

[87] Xin Zhao, Jialin Qiao, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. 2024. Apache TsFile: An IoT-native Time Series File Format. *Proc. VLDB Endow.* 17, 12 (2024), 4064–4076. https://www.vldb.org/pvldb/vol17/p4064-song.pdf

Received January 2025; revised April 2025; accepted May 2025