AnyBlox: Let Your Data Read Itself

Mateusz Gienieczko
giem@in.tum.de

Maximilian Kuschewski
maximilian.kuschewski@tum.de

Thomas Neumann
neumann@in.tum.de

Viktor Leis
leis@in.tum.de

Jana Giceva
jana.giceva@in.tum.de

Technical University of Munich

ABSTRACT

Research advancements in storage formats continuously produce more efficient encodings and better compression rates. Despite this, new formats are not adopted due to high implementation cost, and existing formats cannot evolve because they need to maintain compatibility across systems. Can this problem be solved by introducing a new abstraction? We answer affirmatively with AnyBlox, a framework for reading arbitrary datasets using lightweight WebAssembly decoders bundled with the data. By decoupling decoders from both systems and file format specifications, AnyBlox allows transparent format evolution, instance-optimized encodings, and enables mainstream adoption of research advancements. It integrates seamlessly with modern systems like DuckDB, Spark, and Umbra, while delivering solid performance and security guarantees.

1. INTRODUCTION

While researchers continuously develop faster and more space-efficient encodings [40, 12, 3, 1], as well as novel techniques like correlation-driven compression [27, 31, 51, 69, 52], these find **no adoption** in practice. New storage formats fail to gain traction, while existing formats undergo **ossification**, as datasets continue to be produced using outdated specifications to maintain compatibility with existing readers [39]. In this paper we investigate a future-proof abstraction layer between data encodings and data systems.

Monolithic database systems of the past fully controlled their storage representation, hiding problems inherent to format evolution from the users. However, the era of traditional data silos is over. The database community turns towards modular systems based on open formats [41], the industry moves towards open table formats in data lakes [78], and data scientists want to freely move between platforms when analyzing their existing data in exotic formats (e.g., High Energy Physics [29], bioinformatics [11, 20, 37, 21]).

These developments caused OLAP systems to break away from physical data independence advocated by Codd [23] and instead directly interact with data in its storage format (see Figure 1). Issues of format interoperability are now painfully apparent, as adoption becomes too expensive for maintainers, preventing evolution and limiting users.

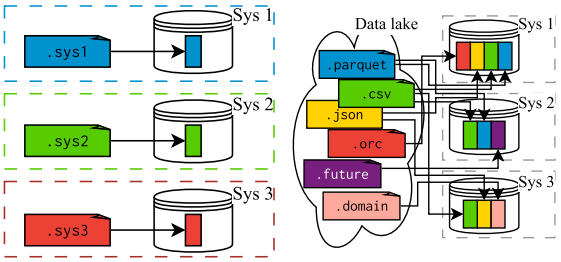


Figure 1: The $\mathbf{N} \times \mathbf{M}$ problem: In the past (left) every system controlled its data and internal format. Today (right) data is outside of systems' control. No system supports all formats and each system needs glue code for each format.

We identify the underlying issue as an instance of the $\mathbf{N} \times \mathbf{M}$ **problem**, where N systems having to support M formats leads to $N \times M$ implementation and maintenance effort. Any novel approach falls into a vicious cycle – an encoding needs to be popular enough to justify the cost of support, but it will not gain popularity without being widely supported. Similar problems were faced and solved before in different domains by introducing an abstraction layer, such as LLVM for compilers [44] and the Language Server Protocol for language tooling [54]. In this paper we ask:

What is the correct abstraction between modern data-processing systems and storage formats?

We analyze this question across four key dimensions that are desirable in a future-proof storage format abstraction: **(1) Portability:** how easy is it to integrate the solution across different systems and hardware architectures? **(2) Security:** what isolation guarantees does the solution provide? Can an error compromise the host system? **(3) Performance:** how does the solution fare in terms of query latency and throughput? **(4) Extensibility:** how easy is it to support a new data format with the solution?

In existing systems, supporting a new storage format usually means either adding a native decoder implementation, or running user-provided code through an extension mechanism. Native integrations can be fast, but they do not scale: each format must be implemented, tested, and maintained separately inside each host system, directly causing the $N \times M$ problem. In-process extensions improve development overhead, but are the worst case for security, since they amount to loading arbitrary code into the database process.

A natural alternative is to isolate extensions (containers/VMs, remote services, serverless UDFs) [61, 66, 9]. This improves isolation but shifts the bottleneck to data movement: decoder workloads are data-intensive, so the overhead of sending large intermediate results across process or network boundaries dominates in current approaches. Finally, approaches based on static verification are promising, but are currently unfeasible for complex decoders.

As Table 1 shows, state-of-the-art approaches tend to offer at most *two* of the four desired properties above at a time. This motivates our main contribution: **AnyBlox, a framework for self-decoding datasets**, which aims to simultaneously provide portability, security, performance, and extensibility. AnyBlox does this by bundling data together with WebAssembly bytecode decoders, allowing any consumer to read any data encoding without knowledge of its details.

Section 2 presents our **Bring-your-own-Decoder architecture** based around lightweight WebAssembly decoders, allowing arbitrary formats to be supported while providing solid performance and security guarantees.

Section 3 shows the integration of AnyBlox into a **wide range of database systems**, which employ vastly different data processing paradigms.

Section 4 showcases **multiple complex encodings** packaged into AnyBlox and evaluates their performance.

With these contributions we argue for a new paradigm in data ingestion – instead of systems decoding datasets based on format specification, **data should decode itself**, eliminating the unsustainable tight coupling between formats and systems. We outline the potential impact and future research directions in Section 5.

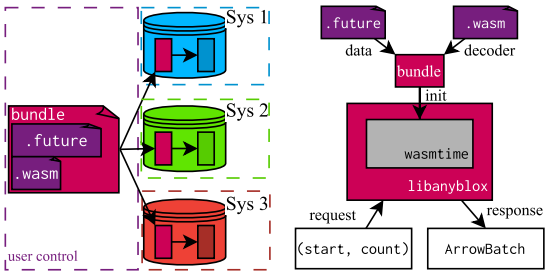
2. ANYBLOX

There are two main conceptual properties a future-proof data access solution must have: First, in order to solve the $N \times M$ problem and enable arbitrary data format evolution, it must abstract both data format internals away from systems, and system internals away from data decoders. Second, this abstraction layer must allow direct file access and data sharing use cases. These goals are at odds with each other, since direct access seemingly precludes any opportunity for introducing abstraction layers – where to put the format decoder that implements the abstraction? We argue that there is only one possible place for the decoder: alongside the data itself.

Self-decoding data solves the conceptual goals of abstraction with direct file access, but comes with its own set of challenges: How to ensure decoder portability across platforms? What are the security implications of running foreign code, and to make it performant? In the following, we describe our implementation of self-decoding data sets, *AnyBlox*, and how it solves these challenges. AnyBlox works on bundles of encoded data alongside **WebAssembly bytecode** defining a Decoder. The decoder can be packaged with the data in a single file, but a data lake might store decoder and data separately in an object store and tie them together via its metadata layer. Hosts can use the **AnyBlox Library (`libanyblox`)** to read any data following the API, which consists of the Decoder format (Section 2.1), the format of data returned to the Host (Section 2.2), and the metadata contained in the dataset (Section 2.3), as shown in Figure 2.

Table 1: Evaluation of different approaches according to our design dimensions. We analyze AnyBlox in Section 2.

Approach	Portability	Security	Performance	Extensibility
native	− − −	−	+ + +	− − −
extensions	−	− − −	+ + +	+ / −
isolated ext.	+ / −	+ + +	−	+
static verification	+	+ / −	+ / −	−
AnyBlox	+ + +	+	+	+ + +



(a) Supporting a `.future` format with AnyBlox using a bundled Decoder. Each Host runs arbitrary Decoders via `libanyblox`.

(b) Host maps the data and the Wasm Decoder in an isolated wasm runtime. It then continually requests batches of tuples.

Figure 2: Integration of AnyBlox into Hosts.

2.1 WebAssembly Decoders

WebAssembly (Wasm) is a specification of a virtual machine and its portable bytecode [30]. Its strength lies in a machine-verifiable guarantee on memory-safety and isolation from the host system [75]. It has been successfully utilized for isolation in different data processing applications, e.g. for ML workloads in Big Query [47], stateful serverless FaaS [63], and UDFs in databases [64], which makes WebAssembly a prime candidate for defining Decoders.

Portability. Wasm code is portable and can be compiled and run on a wide range of architectures, including browsers, mobile devices, embedded, and server workstations; as we show in Section 3, AnyBlox easily integrates into a variety of existing Hosts.

Security. WebAssembly's specification is verified to uphold isolation guarantees [75]. Naturally, the guarantees depend on the *implementation* of the specification being correct. AnyBlox utilizes the Wasmtime implementation of Wasm and the Cranelift compiler to lower the Wasm bytecode to the native instruction set [16, 15], although AnyBlox does not rely on specific implementation details: both the compiler and the runtime are easily replaceable. Recent work suggests that the main source of vulnerabilities in WebAssembly is the compiler, namely the instruction selection and lowering process [73]. Formal verification of WebAssembly software isolation is an area of active research, in particular it was demonstrated that it is possible to implement a provably-safe Wasm sandbox with low performance overheads [13].

AnyBlox is built on top of open-source libraries and within the Rust ecosystem. The majority of the code is in safe Rust,

meaning it guarantees memory and thread safety. All the unsafe code is related to the Wasm memory maps and encapsulated in a few hundred lines of code in a single module, making it easily auditable.

Performance. In theory, Wasm allows for near-native performance, since it is JIT-compiled into the Host's native instruction set. Moreover, AnyBlox avoids expensive data copying due to our memory manager design (see Section 2.6). While we devote Section 4 to performance evaluation, let us note here that research into closing the gap between Wasm and native code is both extensive and ongoing [67, 33]. In 2019 Jangda et al. identified instruction selection as the main source of performance degradation [32], and Yan et al. argue that common LLVM optimizations are ineffective when applied to WebAssembly [76]. Since then, major strides have been made in development of the novel Cranelift compiler. Crucially, AnyBlox can easily benefit from future improvements in Wasm compiler technology, as all the JIT and sandboxing details are encapsulated in `libanyblox` and not exposed in the public API.

Extensibility. Complex decoding schemes are easily implementable, as most general-purpose programming languages have a WebAssembly toolchain and compiler, including popular backends like LLVM [44], JVM [56], and CLI [25]. The main obstacle we have identified when porting decoding schemes to Wasm are SIMD instructions, which are a crucial component for some high performance compression schemes like PFOR [46], DELTA, dictionary, RLE [24], as well as data encodings like JSON [43]. WebAssembly exposes SIMD intrinsics, but developer effort is required to convert from x86/ARM SIMD to the different Wasm instructions. However, since WebAssembly defines SIMD intrinsics on the level of the bytecode, Wasm modules are fully portable: the JIT compiler selects the appropriate instructions for the Host machine.

2.2 Output Format

Since we want everyone to be able to write a single program that will decode their format, we need a clear definition of the decoding output. After considering natural requirements arising from our analysis of existing data processing systems and decoding schemes – columnar storage, support of most standard SQL types, low-overhead conversion to internal representations, portability and extensibility – we have chosen Apache Arrow [7].

Arrow is a robust standard, already supported as a data format by a number of database systems (e.g. DataFusion [41], LingoDB [**?**]), and fulfills all the above criteria. Arrow standardizes the most used data types for integers, floating-point numbers, decimals, date, time, strings, etc. Apache maintains high-quality libraries for efficient processing of Arrow in most commonly used programming languages. Arrow data is columnar, but complex multifield structures can be expressed as a logical type. Moreover, Arrow is extensible and allows custom data types.

2.3 Metadata

AnyBlox requires a thin metadata layer to tie the Decoder and data together. The Host needs to know the schema of the data that it loads from an AnyBlox file, usually during the query planning stage, ergo before decoding begins. We also require the **number of rows** in the compressed file, (an estimate of) **the size of decoded data**, and the **minimum recommended batch size**. These metrics aid the Host in query planning.

To make AnyBlox truly future-proof we also allow providing the **Decoder URI** and **Decoder cryptographic checksum**. This provides two distinct capabilities. First, the file may not contain the decoder and instead provide an external URL as the URI, allowing the Host to download it from a remote location. To maintain security, the downloaded payload should always be verified against the checksum. Second, once an encoding scheme becomes proliferated, a Host may decide to provide a more integrated native implementation of the Decoder. When opening a file it can compare its Decoder URI against a list of well-known URIs and instead decode the payload using its native Decoder.

2.4 Host Communication

AnyBlox is a modular system that needs to span the distance between an arbitrary data processing system and an execution environment for arbitrary WebAssembly code. We carve out two interfaces: between Host and `libanyblox`, and between `libanyblox` and an the WebAssembly Decoder. We present a top-level diagram of these API boundaries in Figure 2b.

To read a dataset the Host first initializes a **Decoder Job** ("open"), passing metadata, the dataset file, and the **WebAssembly Decoder**, which is a Wasm module exposing a `decode_batch` function (c.f., Section 2.5). Then, the Host can repeatedly request arbitrary tuple ranges, which are decoded into **Apache Arrow Record Batches**.

The architecture is illustrated in Figure 2b. A Host opens an AnyBlox bundle (data plus a Wasm Decoder) and then repeatedly requests tuple ranges. Each request invokes the Decoder's `decode_batch` function and returns an Apache Arrow RecordBatch. This design allows Hosts to remain independent of the file format specification, while the decoder remains independent of any single Host.

To make repeated decoding calls efficient, AnyBlox supports a small *state* region that the Decoder can use for caching and for reusing allocations across calls. The state is an optimization only: decoders are required to be correct without it, which keeps the API simple and robust.

2.5 Decoder API

The Decoder needs to expose only one function with the following formal parameters:

- `i32 data`: the pointer to the place in Wasm linear memory where the encoded data starts;
- `i32 data_length`: the size of the encoded data;
- `i32 start_tuple`: the ID of the first tuple to decode;
- `i32 tuple_count`: the number of tuples to decode;
- `i32 state`: the pointer to the place in Wasm's linear memory where the job state is stored;
- `i64 proj_mask`: a bitmask of columns to decode.

The Decoder, constrained by the WebAssembly standard, has no access outside of the sandbox aside from the `memory.grow` call. It is prohibited from modifying the pages containing the encoded data. Wasmtime catches any runtime errors or out-of-bound accesses, terminates the Decoder, and returns to the runtime.

The state page is zero-initialized at the start. State is only an optimization, allowing Decoders to cache metadata applicable to the entire dataset or reuse dynamic memory allocations across calls. In particular, since the Decoder must

perform correctly at the first call, and the runtime may clear the state page before each call, Decoders are guaranteed to be *idempotent*: their behavior can only vary based on the input parameters to `decode_batch` and the data – they have no access to system calls, thus no sources of randomness, clocks, etc. Therefore, because AnyBlox zeroes the state page at the start of every job, it guarantees that decoding the same file twice using the same parameters will return equivalent results.

2.6 Runtime

During initialization the Decoder is JIT compiled and the resulting module cached, so the costly compiler optimization step is incurred only once per Decoder. The linear memory object for the Decoder instance is created. Finally, an object representing the thread-local job is created as a handle to this instance and handed back to the Host. The compilation and isolated execution of Wasm code is done by Wasmtime [16]. Our core technical contribution is custom linear memory management for Decoder instances that ensures data integrity and low overhead access to both the input and output data.

The WebAssembly standard imposes that memory accessible to the Wasm instance is linear and starts at `0x0`. The guest code can allocate memory by invoking a `grow(x)` function, where x is the request size in 64 KiB pages, while the return value is the pointer to the start of the newly allocated region of x pages. The implementation of `grow` is provided by the Host, and the standard does not impose a specific mechanism for memory management on the Host's side.

Wasmtime implements the specification using a well-known Software Fault Isolation technique for 32-bit pointers [26]: It maps 8 GiB of protected virtual memory, while allocation requests from the Wasm instance are served by modifying the protection to allow read-and-write access for the required number of pages. This ensures that any memory access expressible in Wasm code falls within the virtual allocation, as the addressable pointer and length are both limited by $2^{32} - 1$, and thus requires no runtime bounds-checks in the compiled code. The physical memory is handled by the underlying OS's paging mechanism.

The challenge here is providing the entire encoded dataset to the Wasm instance through this linear memory abstraction. A naive solution would reserve the required chunk and copy the dataset into it, which would incur a heavy initialization cost as a potentially large amount of data would be copied. One could also envision a paging abstraction, where the Host would expose a function to read a given page of the dataset while ensuring copying happens only once per page; this approach suffers from excessive complexity, as we would be essentially reimplementing a page mapping and page-miss handling mechanism, and it could not be made transparent for the Decoder – it would have to work on the page-sized chunk abstraction of the dataset, which, among other things, would make vectorized algorithms more complicated and less efficient.

Instead, we propose transparent *data hooks* and thread-local memory pools. The dataset is available as a file descriptor, which we memory map into the virtual linear memory. This takes few system calls and is fast even for large datasets. The Decoder transparently accesses the data the same as any other area of its linear memory. This approach is similar to how Faasm handles shared memory regions by swapping memory maps [63], but AnyBlox can provide a simpler and more cache-friendly mechanism since threads share only read access to the data.

AnyBlox makes no difference between file descriptors pointing to physical files, shared memory regions, or in-memory buffers like those maintained by `memfd_create` [50]. The initial memory required for the module's code, stack, and static data is allocated at the beginning of the memory. This is usually a couple megabytes. The hook is then mapped into the first available page after the initial pages, with the state page allocated immediately after. The Decoder's allocation requests are served from the following pages, growing dynamically as required.

The key property of this design is that linear memory regions can be cached in the thread-local pool and reused. While the Host is likely to decode a dataset with multiple threads, it is unlikely it will have two jobs running on the same thread at the same time, since context switching is detrimental to performance; even less likely to have two jobs on the same thread on the same dataset at the same time. Assume we allocate a linear memory region for a Decoder, it finishes its job, and we want to decide if another instance can reuse the memory. It is possible if the following constraints are upheld: a) the initial memory pages are large enough; b) the already hooked dataset is the same; c) all additionally allocated memory is zeroed (this is required by the WebAssembly standard). In practice, the initial regions are small (around a megabyte). When creating an instance we know the initial requirement as well as the dataset used, so the cache policy is straightforward: If a fitting map exists, use it, clear out the allocated memory with `madvise`, and protect all pages after the state page with `PROT_NONE`; if no fitting map exists and we are below memory instance limit, create a new virtual allocation; otherwise, pick the least recently used existing map and reuse it by unhooking the existing dataset and reinitializing.

The system can only infer that the dataset is the same if it comes from exactly the same opened bundle object. Sharing a single bundle between threads is important for performance, and simple, since the data is read-only. Moreover, if a query plan contains the same bundle as a source multiple times, the Host would benefit from merging them as one.

If the file is not directly available, e.g. it resides remotely and cannot be mapped to a file descriptor, it must first be downloaded. An alternative solution would be to enable lazy loading by custom handling of a page fault, e.g. via signal handling or specific operating system extension points [49]. In this scenario, the Host would provide code that populates a given page when it is first accessed by the Decoder. The Decoder would access memory as usual with full transparency, but only data it actually needs to access would be populated by the Host dynamically. This would not interfere with our memory caching mechanisms, allowing the materialized data to be reused later. AnyBlox does not yet implement this technique, but it is of interest for future work.

2.7 Discussion

AnyBlox is built to be future-proof. Our design allows for evolution without breaking existing implementations and datasets. In particular, (1) **Improvements to Wasm compilers** can be directly harnessed with an update to `libanyblox`; since the Wasm *bytecode* remains the same, this is a seamless upgrade that preserves backwards-and-forwards

compatibility. (2) A **new decoder dialect** could also be introduced in a compatible manner as long as there exists a Wasm transpiler. For example, if a statically-verified DSL for Decoders was designed, a Decoder in the DSL could be distributed with an automatic equivalent Wasm version being generated alongside. Systems running an older version of AnyBlox would use the Wasm code as before, while new versions could opt into running the DSL. (3) AnyBlox supports **Optional metadata**, allowing system-specific metadata that serves as a hint for one system but not another. An example of this is *sizeInBytes*, describing the estimated size of a fully decoded dataset in memory, which can aid distributed systems (e.g. Spark) in their network traffic estimations.

We believe these properties make AnyBlox a suitable solution for the problem of data format ossification. However, the use of WebAssembly also introduces some drawbacks. First, WebAssembly by default has a 32-bit address space into which AnyBlox maps datasets, meaning each mapped file has a maximum size of 4 GiB. Wasm recently gained support for using 64-bit addresses [17], but the performance impact of enabling this option is not yet well studied. Most systems using e.g. Parquet, keep file sizes well below 4 GiB [70, 6, 65] to facilitate cheap updates, so we expect this restriction to have little impact in practice. Second, Wasm code is currently not as fast as native code and does not support wide (> 128bit) vector instructions. Section 4 studies the performance impact of this limitation.

3. INTEGRATION

To demonstrate the portability of our solution we have integrated them into four fundamentally different systems: **DataFusion** [41], **DuckDB** [59], **Umbra** [55], and **Spark** [77]. We posit that these four Hosts cover a wide-enough variety of paradigms to strongly suggest AnyBlox poses no significant hurdles when integrating into an arbitrary data-processing system. Differences include:

Core paradigms. Umbra is a compiling system, and DuckDB and DataFusion are vectorized systems – two very different paradigms [35]. Spark focuses on distributed workloads, and only generates code for expressions.

Parallelism. Umbra and DuckDB are both morsel-driven with a central executor [45], while Spark and DataFusion use Volcano-style parallelism with exchange operators [28].

Programming languages. Umbra and DuckDB are written in C++, DataFusion in Rust, while Spark is written in Java and Scala, running in a managed JVM environment.

Maturity. Spark is a mature platform for diverse workloads and provides a plugin system; DuckDB is a widely used in-process analytical database with a robust extension API; Umbra is a closed-source research database that focuses on perfromance and provides no dedicated extension points; DataFusion was published only recently in 2024.

In the following we highlight integration details of interest for developers seeking to bring AnyBlox into their system. In case of DuckDB, its flexible system allows us to define a custom AnyBlox operator and distribute it as a plug-and-play module, requiring no invasive changes to the main DuckDB codebase. DataFusion provides a similar extension point for a custom table source, and the Spark integration uses a `.jar` plugin. We expect any system supporting external extension modules to allow easy integration in this manner. Umbra is not open-source, so we integrated AnyBlox directly into the codebase and only provide a precompiled binary.

3.1 Internal Data Representation

While the Arrow format is optimized for zero-copy data transfer between systems using it as its internal data format like DataFusion, not all hosts do. SQL-based systems rarely have standardized logical types, and each host may choose to perform the SQL-to-Arrow type mapping differently. Arrow's representation of primitive types is natural, so types such as `Int32` or `Float64` require no conversion. `Date` types need to be adjusted to the system's internal representation, e.g., Umbra requires translation into the Julian calendar. `Bool` types are bit-packed in Arrow and might require conversion into bytes. This results in a zero-copy, in-place type conversion in most cases, with booleans being an exception.

Beyond the representation of primitive types, systems use different internal data layouts. DataFusion uses Arrow as its native format [41], allowing truly zero-copy data sharing with AnyBlox. DuckDB and Spark, like many modern systems [71, 62], support reading data in the Arrow format, converting it to their internal data layout on the fly. Thus, there already is mainline code that maps Arrow types and translates arrays to the internal representations. Umbra requires emitting code to load the values from Arrow buffers into registers based on the underlying physical type.

String types require the most care. Arrow supports two layouts: `Utf8` and `Utf8View`. `Utf8` contains a length and a pointer to the character data stored in a separate buffer. This can be easily translated to the Spark string format (regular JVM `String` type). However, both DuckDB and Umbra store strings using the small-string optimization [55], requiring more complex logic. `Utf8View` is actually based on this representation [8] as values may or may not contain offsets in separate buffers, which need to be translated to native pointers. Crucially, while the `Utf8`-to-`Utf8View` translation cannot be performed entirely in-place, the actual string data buffers do not need copying. The transfer is truly zero-copy in cases where the host and the decoder use the `Utf8View` layout for all strings.

3.2 Language Interoperability

Umbra and DuckDB are written in C++, while the AnyBlox Runtime is written in Rust. This makes the integration simple, as `libanyblox` can easily provide C++ bindings into Rust code. Integration into any system using a similar systems-level programming language with support for a C FFI should be similar. With a Rust-native system like DataFusion the integration is straightforward. Spark is written in Scala and Java, and thus requires a Java Native Interface bridge to communicate with `libanyblox`. Calls to AnyBlox return FFI-safe arrays – Arrow specifies a C data interface that provides uniform FFI across platforms, and thus the Arrow implementation on the JVM allows loading `ArrowArray` payloads into JVM objects.

3.3 Concurrent Scans

All integrations follow a similar pattern to implement parallel scanning of data, where an AnyBlox operator represents a full scan job that then gets divided into tasks for parallel processing. AnyBlox knows the exact size of the table from its metadata, and can skip columns based on the projection mask provided by the Host.

Spark and DataFusion decide the parallelism during query

Table 2: Approximate complexity of integrating AnyBlox into each of the Hosts by a single programmer. Spark also required writing a JNI bridge (433 LoC), which is reusable for any other JVM Host.

System	Paradigm	Parallelism	LoC	Days
Umbra	Compiled	Morsel-driven	1305	10
DuckDB	Vectorized	Morsel-driven	586	3
Spark	Hybrid	Volcano-style	756	15
DataFusion	Vectorized	Volcano-style	461	2

planning, letting the AnyBlox scan operator statically partition the workload without shared global state. This partitioning logic is independent of the underlying format itself and could be extended to multiple nodes (e.g., with Hadoop). The morsel-driven parallelism used by DuckDB and Umbra involves a global executor, with threads dynamically fetching work items to process in their local AnyBlox job. Threads keep local state, and, as an optimization, they may request large batches with `decode_batch` and process them in smaller chunks, e.g., the DuckDB global vector size.

3.4 AnyBlox Is Not A File Format

In recent years there has been a proliferation of file formats designed to meet the needs of current and future data management systems [2, 42]. Of note is the F3 format, which incorporates a similar idea of portable Wasm decoders for its Encoding Units [79]. AnyBlox, in contrast, is explicitly *not* a file format – it is a framework that can be used when building file formats, database systems, and applications. In particular, unlike F3, AnyBlox imposes minimal requirements on the layout of the underlying data and can be used to decode existing datasets with no changes. This is exemplified by our ROOT integration, where we use completely unmodified `.root` files as input.

AnyBlox is also decoupled from statistics and metadata. It does not enforce any metadata granularity, allowing the Host system to keep statistics on a column level, encoding page level, or with some other system-specific index. In this way AnyBlox functionally decomposes the storage encoding from the search acceleration layer [58].

The key characteristic of AnyBlox is its transparent, zero-copy data sharing with the sandbox regardless of the data layout. This flexibility allows AnyBlox integration to be a path towards future-proofing existing data formats. As an example, we implemented an experimental integration of AnyBlox decoders into the Parquet file structure, with Decoder definitions given in place of dictionary pages. A similar approach would work for the Vortex format [68].

4. EVALUATION

While the security guarantees of AnyBlox follow from the WebAssembly sandbox and portability is proven by our system integration in Section 3, what remains is substantiating our extensibility claims and showing the effectiveness of Wasm and the employed memory architecture for performance-intensive workloads. For that purpose, we evaluate the AnyBlox integrations using varied decoders, OLAP workloads, and microbenchmarks.

Experiments are performed on a 32-core Intel Xeon Gold 6430 machine with 256 GB of DDR5 RAM. Unless stated otherwise, samples are collected after two warm-up runs, which isolates the overhead of disk reads, Decoder compilation, and thread initialization. We present median results, but unaggregated results are available in our repository.

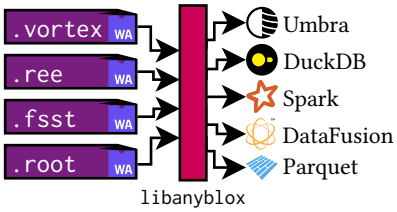


Figure 3: AnyBlox bridges the gap between a variety of storage encodings and arbitrary data processing systems.

4.1 Did we solve the $N \times M$ problem?

Yes, AnyBlox solves the $N \times M$ problem, as we demonstrate in the following. We cover a wide range of formats by implementing (1) the encoding schemes FSST [12] and Run-End Encoding (REE), (2) Vortex [68], a novel columnar file format using state-of-the-art compression research [12, 3, 40, 1], and (3) CERN ROOT, a specialized binary format storing exabytes of particle physics data [18]. We complement this with an integration of AnyBlox into a variety of systems, done by a programmer with no prior experience with any Host codebase. No integration took more than two weeks, and all integrations were done in less than 2000 lines of code (LoC), as Table 2 shows. While neither the integration time nor the required lines of code are rigorous measurements of software complexity, they do indicate that AnyBlox can be straightforwardly ported to different systems. Figure 3 shows an overview. All integrations were straightforward to implement and required no further changes to any Host system. This enables great interoperability with minimal effort, allowing data transformations such as the following (native DuckDB syntax):

```
COPY (SELECT * FROM anyblox('./ParticleDecay.root')
      JOIN anyblox('./metrics.vortex') USING pType)
TO './ParticleDecayJoined.parquet'
```

4.1.1 *Vortex*

Vortex [68] is a novel format based on BtrBlocks [40] and other recent encoding research [3, 1]. Vortex outperforms existing alternatives using modern and multi-layered encoding schemes to achieve high compression rates, but suffers from the adoption problem outlined in Section 1. We built an AnyBlox decoder with little effort by compiling the Vortex codebase, specifying `wasm32` as the target for the Rust compiler, and excluding non-essential, OS-specific logic (such as file I/O), which cannot compile to Wasm. AnyBlox can bring this cutting-edge research codec into multiple database systems without any changes on the Host side, leading to immediate compression and performance improvements.

4.1.2 *CERN ROOT*

ROOT is a complex format developed at CERN since 1995 and is widely used across the High Energy Physics (HEP) community, with more than 2 EB of data stored in ROOT files [18]. Though much of this data is tabular, ROOT itself is not a relational format and allows encoding arbitrary C++

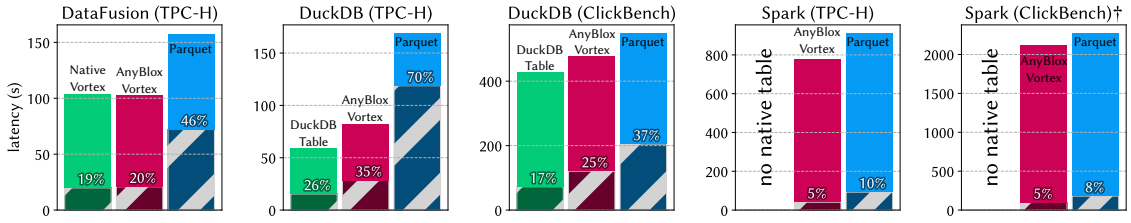


Figure 4: TPC-H and ClickBench evaluated on a single thread with DuckDB, Spark† , and DataFusion. Time spent in the scan of the main table is shaded.

objects. Since ROOT is not supported in any conventional data systems, recent work on analyzing ROOT data with SQL had to convert all datasets to Parquet [29]. Our AnyBlox decoder (based on a Rust port [14] for a large subset of ROOT) instead makes ROOT files immediately readable by AnyBlox-enabled systems – no conversion required. Umbra surpasses the throughput of the native ROOT framework when performing an aggregation on particle decay data [48]. More importantly, Umbra's parallel processing automatically works for ROOT files, yielding 20× better throughput on 32 threads, while the native framework offers no easy mechanism for parallelism.

4.1.3 FSST & Run-end encoding

FSST [12] and REE are lightweight encoding schemes that can be easily implemented in any language that compiles to WebAssembly. Both are also implemented in Vortex [68]. We use standalone Rust implementations in additional decompression microbenchmarks (omitted in this Research Highlight version).

4.2 Vortex Native vs. Vortex in AnyBlox

We show that (1) AnyBlox can be used to provide both compression *and* performance improvements by introducing novel encoding schemes; and (2) AnyBlox's sandboxing overheads are negligible for complex decoding schemes as compared to a native solution. To that end, we evaluate the TPC-H benchmark with scale factor 20 in DataFusion and varying the representation of the largest table (`lineitem`). DataFusion is written in Rust, allowing direct integration of AnyBlox and Vortex. We use the native DataFusion Parquet reader on a file with the default compression scheme based on Snappy, which provides the best decompression throughput; the Vortex file format decoder inside AnyBlox; and the exact same decoder but compiled natively without any sandboxing. Predicate pushdown has been disabled.

Across all queries both Vortex implementations outperform Parquet with over 2× lower scan latency (c.f., Figure 4, left), and it provides a better compression rate (c.f., Figure 5). For Vortex, the bulk of processing time is spent executing relational operators (around 80%), whereas the Parquet reader spends almost half of its time decoding the file. The difference in performance between native and AnyBlox-based Vortex decoding is marginal, with a 5% increase in scan latency. However, this translates to a mere 1% increase in overall latency, which is within measurement noise.

†Spark only successfully executed 18 out of 24 selected ClickBench queries due to issues with the native Parquet reader unrelated to AnyBlox.

4.3 DuckDB Performance

Next, we test AnyBlox in a full database system – DuckDB. We compare AnyBlox Vortex to the native Parquet reader, but also include comparison against the native, non-portable table format of DuckDB. The performance results are shown in Figures 5, 4, and 6.

4.3.1 TPC-H

Across all queries the more efficient Vortex encoding outperforms Parquet despite the Wasm overhead, and AnyBlox is often close to the native baseline. In the single-threaded `lineitem` scan in Figure 4, both the native format and AnyBlox spend most of the time executing relational operators (74% and 65%, respectively), compared to Parquet, which has to spend most of the time (70%) decoding data. The biggest differences between in-memory table scans and AnyBlox occur in scan-dominated queries that select many columns from `lineitem` (Q1), high selectivity `lineitem` filtered scans (Q10, Q12 at 25% and 0.5% selectivity, respectively), and in Q21, where the triple self-join on `lineitem` exacerbates the cost, as Figure 6 shows.

4.3.2 ClickBench

TPC-H is an important standard benchmark, but fails to represent the real world, especially when it comes to string data [40, 27, 74]. We therefore turn to the ClickBench benchmark [22], which contains real-world web-traffic data in a single `hits` table. To focus on string compression, the following experiment extracts the five string columns that are heavily used in the majority of queries from this table, and compresses them using Parquet, Vortex, and DuckDB, which uses FSST [12] string compression in its native table format. Compared to DuckDB, Vortex achieves a 1.85× better compression rate due to its multistage encoding without relying on heavy-weight Snappy compression like Parquet, as Figure 5 shows.

Since AnyBlox can only support files at most 4 GB in length and compressed ClickBench exceeds that, we partition the data into 4 files with equal number of tuples in each. We do this for the native representation (split into 4 tables), Parquet, and Vortex alike. We restrict the workload to a subset of ClickBench queries focusing on string processing. In most queries, total processing time vastly outweighs scan times, Q29 being a prime example due to a complicated regular expression. In contrast, Q24 spends more time in the scan because DuckDB cannot push down the substring predicate into the non-native formats. Overall, AnyBlox spends only 25% of the total runtime in the scan.

In the multithreaded scenario, AnyBlox still outperforms native Parquet readers, as the following table shows (in mil-

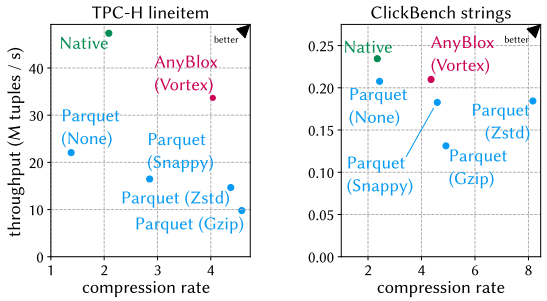


Figure 5: Compression ratio vs workload throughput on DuckDB for native tables, AnyBlox, and all Parquet compression codecs supported by DuckDB.

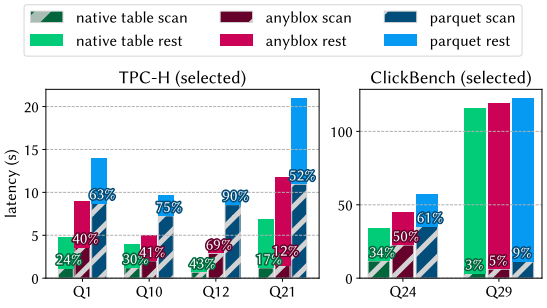


Figure 6: Selected queries from DuckDB TPC-H and ClickBench evaluation. Other systems exhibit similar behavior. Time spent in the scan of the main table is shaded.

lions of tuples per second, TPC-H scale factor 20 workload):

	Native table	**AnyBlox (Vortex)**	**Parquet (Snappy)**
1 thread	47.36 MT/s	33.67 MT/s	16.47 MT/s
32 threads	978.17 MT/s	593.18 MT/s	396.33 MT/s
ratio	20.65 ×	17.62 ×	24.06 ×

Parquet seems to scale marginally better, but at a large cost in terms of absolute performance [53].

4.4 Spark Performance

Finally, we evaluate TPC-H and ClickBench using Spark. Spark has no native data format, but has a native Parquet reader. Spark on a single thread is less efficient than DuckDB – most time ($\geq 90\%$) is spent executing operators. This means that, in Spark, scan performance has little impact on real-life OLAP workloads. For example, a 40% less efficient decoder would only make the Spark workflow 4% slower. Figure 4 shows this: Though Vortex in AnyBlox is more performant than Parquet, the overall impact is small. These characteristics are exacerbated in ClickBench, where the analytical operations dominate the scan even further.

4.5 Decoder microbenchmarks

To understand the performance of AnyBlox in detail, we perform microbenchmarks studying the impact of batch-size, the scaling behavior of AnyBlox, and perform a breakdown of the per-thread and per-batch initialization cost. These microbenchmarks show that WebAssembly overheads depend on the decoder's usage of SIMD, that AnyBlox scales well across threads, and that per-batch initialization cost is small. While lack of wide SIMD instructions in WebAssembly negatively impacts performance in microbenchmarks, our end-to-end results show that the overall impact is usually small because scans are only one component of query latency in real systems. The detailed results of these microbenchmarks are available in the full version of the paper.

5. SUMMARY AND OUTLOOK

We presented AnyBlox, a framework for self-decoding data that is easily **portable** across arbitrary data processing systems and **extensible** for arbitrary decoding schemes. Any system that implements an AnyBlox scan is able to **securely** and **performantly** read any format providing a WebAssembly decoder, solving the $N \times M$ problem and format ossification. Hosts can use bundled decoders directly, securely reference remote decoders using their cryptographic hashes, and transparently apply native decoders for popular encoding schemes. One can envision a future "Parquet v3" format that integrates AnyBlox, allowing arbitrary format evolution and instance-optimized encoding schemes.

Instance-optimized encoding schemes such as correlation-based compression [31, 52, 51], learned white-box compression [27], and entropy compression [60] can be very powerful, but their specialized nature makes their adoption even more difficult than more general-purpose schemes. AnyBlox paves the way for exploring a large space of clever, instance-optimized data representations – even pushing towards Kolmogorov complexity [38] – while still being future-proof and readable by any system.

Building future-proof data formats is but one of the problems that need to be solved to facilitate truly modular, future-proof, open database systems. The community has been calling for moving away from large, monolithic, overly complex systems for at least 25 years [19, 36, 57]. LLVM showed that an open framework can revolutionize the design of complex modern systems like compilers [44]. Recently, Lamb et al. presented an open and extensible query engine based around composable system design principles [41], proving such an approach can be successful. Execution plan frameworks have been made more extensible using sub-operators [10, 34], portable query optimizers [5, 4], and system-independent persistence mechanisms [72]. Our work joins this effort with an extensible and modular solution for arbitrary storage formats.

Acknowledgements

We thank Andrew Lamb for fruitful discussion and his feedback, and Maurice Scholtes for contributing the Parquet integration.

Funded/Co-funded by the European Union (ERC, FDS, 101164556; ERC, CODAC, 101041375). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

6. REFERENCES

[1] A. Afroozeh and P. Boncz. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.*, 16(9):2132–2144, May 2023. Publisher: VLDB Endowment.

[2] A. Afroozeh and P. Boncz. The FastLanes File Format. *Proc. VLDB Endow.*, 18(11):4629–4643, July 2025.

[3] A. Afroozeh, L. X. Kuffo, and P. A. Boncz. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data*, 1(4):230:1–230:26, 2023.

[4] R. Alotaibi, Y. Tian, S. Grafberger, J. Camacho-Rodríguez, N. Bruno, B. Kroth, S. Matusevych, A. Agrawal, M. Behera, A. Gosalia, C. A. Galindo-Legaria, M. Joshi, M. Potocnik, B. Sezgin, X. Li, and C. Curino. Towards Query Optimizer as a Service (QOaaS) in a Unified LakeHouse Ecosystem: Can One QO Rule Them All? In *Conference on Innovative Data Systems Research*, Amsterdam, The Netherlands, 2024. www.cidrdb.org. arXiv: 2411.13704.

[5] C. Anneser, N. Tatbul, D. Cohen, Z. Xu, P. Pandian, N. Laptev, and R. Marcus. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.*, 16(12):3515–3527, Aug. 2023. Publisher: VLDB Endowment.

[6] Apache Parquet. Recommended parquet file size on hdfs, 2025.

[7] Apache Software Foundation. Apache Arrow, 2023.

[8] Apache Software Foundation. Apache Arrow Columnar Format - Variable-size Binary View Layout, 2023.

[9] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, E. Hotinger, Y. Leshinksy, J. Liang, M. McCreedy, F. Nagel, I. Pandis, P. Parchas, R. Pathak, O. Polychroniou, F. Rahman, G. Saxena, G. Soundararajan, S. Subramanian, and D. Terry. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 2205–2217, New York, NY, USA, 2022. Association for Computing Machinery. event-place: Philadelphia, PA, USA.

[10] M. Bandle and J. Giceva. Database technology for the masses: sub-operators as first-class entities. *Proc. VLDB Endow.*, 14(11):2483–2490, July 2021. Publisher: VLDB Endowment.

[11] A. Black, D. R. MacCannell, T. R. Sibley, and T. Bedford. Ten recommendations for supporting open pathogen genomic analysis in public health. *Nature Medicine*, 26(6):832–841, June 2020.

[12] P. Boncz, T. Neumann, and V. Leis. FSST: fast random access string compression. *Proc. VLDB Endow.*, 13(12):2649–2661, July 2020. Publisher: VLDB Endowment.

[13] J. Bosamiya, W. S. Lim, and B. Parno. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1975–1992, Boston, MA, Aug. 2022. USENIX Association.

[14] C. Bourjau. mALICE: An open source framework for analyzing ALICE's Open Data, 2018.

[15] Bytecode Alliance. Cranelift, 2016.

[16] Bytecode Alliance. wasmtime, 2019.

[17] Bytecode Alliance. Memory64 proposal for webassembly, 2025.

[18] CERN. CERN ROOT Format. CERN, 1995.

[19] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 1–10, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[20] C. Chen, S. Nadeau, I. Topolsky, N. Beerenwinkel, and T. Stadler. Advancing genomic epidemiology by addressing the bioinformatics bottleneck: Challenges, design principles, and a Swiss example. *Epidemics*, 39:100576, June 2022. Place: Netherlands.

[21] C. Chen, A. Taepper, F. Engelniederhammer, J. Kellerer, C. Roemer, and T. Stadler. LAPIS is a fast web API for massive open virus sequencing data. *BMC Bioinformatics*, 24(1):232, June 2023.

[22] ClickHouse. ClickBench, 2022.

[23] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[24] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *International Conference on Extending Database Technology*, pages 72–83, Venice, Italy, 2017. OpenProceedings.

[25] ECMA International. Common language infrastructure (cli), 2012.

[26] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 293–306, USA, 2008. USENIX Association. event-place: Boston, Massachusetts.

[27] B. V. Ghita, D. G. Tomé, and P. A. Boncz. White-box Compression: Learning and Exploiting Compact Table Representations. In *Conference on Innovative Data Systems Research*, 2020.

[28] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 102–111, New York, NY, USA, 1990. Association for Computing Machinery. event-place: Atlantic City, New Jersey, USA.

[29] D. Graur, I. Müller, M. Proffitt, G. Fourny, G. T. Watts, and G. Alonso. Evaluating query languages and systems for high-energy physics data. *Proc. VLDB Endow.*, 15(2):154–168, Oct. 2021. Publisher: VLDB Endowment.

[30] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. Association for Computing Machinery. event-place: Barcelona, Spain.

[31] A. Ilkhechi, A. Crotty, A. Galakatos, Y. Mao, G. Fan, X. Shi, and U. Cetintemel. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1733–1746, New York, NY, USA, 2020. Association for Computing Machinery. event-place: Portland, OR, USA.

[32] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

[33] S. Jiang, R. Zeng, Z. Rao, J. Gu, Y. Zhou, and M. R. Lyu. Revealing Performance Issues in Server-Side WebAssembly Runtimes Via Differential Testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 661–672, 2023.

[34] M. Jungmair and J. Giceva. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.*, 16(11):3461–3474, July 2023. Publisher: VLDB Endowment.

[35] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, Sept. 2018. Publisher: VLDB Endowment.

[36] A. Khurana and J. L. Dem. The Modern Data Architecture: The Deconstructed Database. *login Usenix Mag.*, 43:36–40, 2018.

[37] S. Knyazev, K. Chhugani, V. Sarwal, R. Ayyala, H. Singh, S. Karthikeyan, D. Deshpande, P. I. Baykal, Z. Comarova, A. Lu, Y. Porozov, T. I. Vasylyeva, J. O. Wertheim, B. T. Tierney, C. Y. Chiu, R. Sun, A. Wu, M. S. Abedalthagafi, V. M. Pak, S. H. Nagaraj, A. L. Smith, P. Skums, B. Pasaniuc, A. Komissarov, C. E. Mason, E. Bortz, P. Lemey, F. Kondrashov, N. Beerenwinkel, T. T.-Y. Lam, N. C. Wu, A. Zelikovsky, R. Knight, K. A. Crandall, and S. Mangul. Unlocking capacities of genomics for the COVID-19 response and future pandemics. *Nature Methods*, 19(4):374–380, Apr. 2022.

[38] A. N. Kolmogorov. On tables of random numbers. *Sankhy: The Indian Journal of Statistics, Series A*, pages 369–376, 1963. Publisher: JSTOR.

[39] L. Kuiper. Query Engines: Gatekeepers of the Parquet File Format, Jan. 2025.

[40] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data*, 1(2):118:1–118:26, June 2023. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[41] A. Lamb, Y. Shen, D. Heres, J. Chakraborty, M. O. Kabak, L.-C. Hsieh, and C. Sun. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, pages 5–17, New York, NY, USA, 2024. Association for Computing Machinery. event-place: Santiago AA, Chile.

[42] Lance contributors. Lance, 2025.

[43] G. Langdale and D. Lemire. Parsing gigabytes of JSON per second. *VLDB J.*, 28(6):941–960, 2019.

[44] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, San Jose, CA, USA, 2004. IEEE Computer Society.

[45] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, New York, NY, USA, 2014. Association for Computing Machinery. event-place: Snowbird, Utah, USA.

[46] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2203.

[47] J. Levandoski, G. Casto, M. Deng, R. Desai, P. Edara, T. Hottelier, A. Hormati, A. Johnson, J. Johnson, D. Kurzyniec, S. McVeety, P. Ramanathan, G. Saxena, V. Shanmugan, and Y. Volobuev. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, pages 334–346, New York, NY, USA, 2024. Association for Computing Machinery. event-place: <conf-loc>, <city>Santiago AA</city>, <country>Chile</country>, </conf-loc>.

[48] LHCb collaboration (2017). Matter Antimatter Differences (B meson decays to three hadrons) - Data Files, 2011.

[49] Linux Kernel Developers. userfaultfd(2) Linux User's Manual, 2024.

[50] Linux Kernel Developers. memfd_create(2) Linux User's Manual, 2025.

[51] H. Liu, M. Stoian, A. v. Renen, and A. Kipf. Corra: Correlation-Aware Column Compression. In *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases, VLDB 2024, Guangzhou, China, August 26-30, 2024*. VLDB.org, 2024.

[52] X. Lyu, A. Kipf, P. Pfeil, D. Horn, J. Giceva, and T. Kraska. CorBit: Leveraging Correlations for Compressing Bitmap Indexes. In *VLDB Workshops*, 2023.

[53] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 14, USA, 2015. USENIX Association. event-place: Switzerland.

[54] Microsoft. Microsoft LSP, 2016.

[55] T. Neumann and M. J. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research*, Amsterdam, The Netherlands, 2020. www.cidrdb.org.

[56] Oracle. The java virtual machine specification, 2013.

[57] P. Pedreira, O. Erling, K. Karanasos, S. Schneider, W. McKinney, S. R. Valluri, M. Zait, and J. Nadeau. The Composable Data Management System Manifesto. *Proc. VLDB Endow.*, 16(10):2679–2685, June 2023. Publisher: VLDB Endowment.

[58] M. Prammer, X. Zeng, R. Meng, W. McKinney, H. Zhang, A. Pavlo, and J. M. Patel. Towards Functional Decomposition of Storage Formats. In *Conference on Innovative Data Systems Research*, Amsterdam, The Netherlands, 2025. www.cidrdb.org.

[59] M. Raasveldt and H. Mühleisen. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery. event-place: Amsterdam, Netherlands.

[60] V. Raman and G. Swart. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 858–869. VLDB Endowment, 2006. Place: Seoul, Korea.

[61] K. Saur, T. Mirmira, K. Karanasos, and J. Camacho-Rodríguez. Containerized execution of UDFs: an experimental evaluation. *Proc. VLDB Endow.*, 15(11):3158–3171, July 2022. Publisher: VLDB Endowment.

[62] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.*, 17(12):3731–3744, Aug. 2024. Publisher: VLDB Endowment.

[63] S. Shillaker and P. Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[64] M.-F. Sichert. *Efficient and Safe Integration of User-Defined Operators into Modern Database Systems*. PhD Thesis, Technische Universität München, 2024.

[65] Snowflake. Introduction to extern tables in snowflake, 2025.

[66] Snowflake, Inc. Introduction to external functions | Snowflake Documentation, 2025.

[67] B. Spies and M. Mock. An Evaluation of WebAssembly in Non-Web Environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10, 2021.

[68] Spiral. vortex, 2024.

[69] M. Stoian, A. v. Renen, J. Kobiolka, P.-L. Kuo, J. Grabocka, and A. Kipf. Lightweight Correlation-Aware Table Compression. *ArXiv*, abs/2410.14066, 2024.

[70] The Apache Software Foundation. Parquet file size in apache impala, 2025.

[71] TileDB, Inc. TileDB, Apr. 2024.

[72] E. Tsalapatis, R. Hancock, R. Hossain, and A. J. Mashtizadeh. MemSnap ţ Checkpoints: A Data Single Level Store for Fearless Persistence. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pages 622–638, New York, NY, USA, 2024. Association for Computing Machinery. event-place: <conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>.

[73] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown. Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, pages 231–248, New York, NY, USA, 2024. Association for Computing Machinery. event-place: <conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>.

[74] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*, pages 1:1–1:6. ACM, 2018.

[75] C. Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, New York, NY, USA, 2018. Association for Computing Machinery. event-place: Los Angeles, CA, USA.

[76] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, pages 533–549, New York, NY, USA, 2021. Association for Computing Machinery. event-place: Virtual Event.

[77] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association. event-place: San Jose, CA.

[78] M. A. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research*, 2021.

[79] X. Zeng, R. Meng, M. Prammer, W. McKinney, J. M. Patel, A. Pavlo, and H. Zhang. F3: The Open-Source Data File Format for the Future. *Proc. ACM Manag. Data*, 3(4):1–27, Sept. 2025.