

AnyBlox: A Framework for Self-Decoding Datasets

Mateusz Gienieccko
Technical University of Munich
Munich, Germany
giem@in.tum.de

Maximilian Kuschewski
Technical University of Munich
Munich, Germany
maximilian.kuschewski@tum.de

Thomas Neumann
Technical University of Munich
Munich, Germany
neumann@in.tum.de

Viktor Leis
Technical University of Munich
Munich, Germany
leis@in.tum.de

Jana Giceva
Technical University of Munich
Munich, Germany
jana.giceva@in.tum.de

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.

PVLDB Reference Format:

Mateusz Gienieccko, Maximilian Kuschewski, Thomas Neumann, Viktor Leis, and Jana Giceva. AnyBlox: A Framework for Self-Decoding Datasets. PVLDB, 18(11): 4017 - 4031, 2025.
doi:10.14778/3749646.3749672

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/AnyBlox/vldb-2025>.

1 INTRODUCTION

This paper investigates a future-proof abstraction layer between data encodings and data systems. While researchers continuously develop faster and more space-efficient encodings [1, 2, 13, 49], as well as novel techniques like correlation-driven compression [31, 39, 58, 59, 82], 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fail to gain traction, while existing formats undergo **ossification**, as datasets continue to be produced using outdated specifications to maintain compatibility with existing readers [48].

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

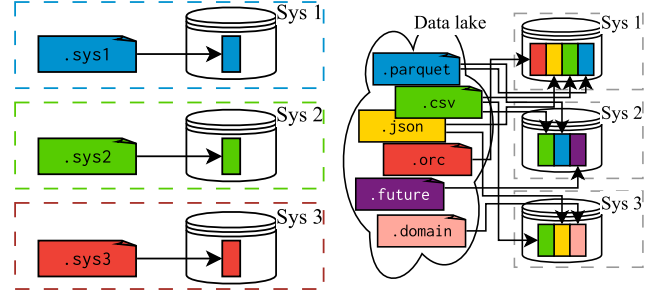


Figure 1: The $N \times M$ problem: In the past (left) every system controlled its data and internal format. Today (right) data is outside of systems' control. No system supports all formats and each system has specialized glue code for each format.

open formats [50], the industry moves towards open table formats in data lakes [100], and data scientists want to freely move between platforms when analyzing their existing data in exotic formats (e.g., High Energy Physics [33], bioinformatics [12, 22, 23, 45]).

These developments force OLAP systems to break away from physical data independence advocated by Codd [25] and instead directly interact with data in its storage format, as illustrated in Figure 1. This makes issues of format interoperability painfully apparent, as format adoption becomes too expensive for maintainers, preventing evolution and locking away potential users.

We identify the underlying issue as an instance of the $N \times M$ **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 that prevents adoption – 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's language-independent intermediate representation for compilers [52] and the Language Server Protocol for language tooling [62]. In this paper we ask:

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

In Section 2 we analyze the strengths and weaknesses of state of the art and identify key properties such an abstraction must provide. As our main contribution we introduce **AnyBlox**, a **framework for self-decoding datasets**, in which data is bundled together

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749672

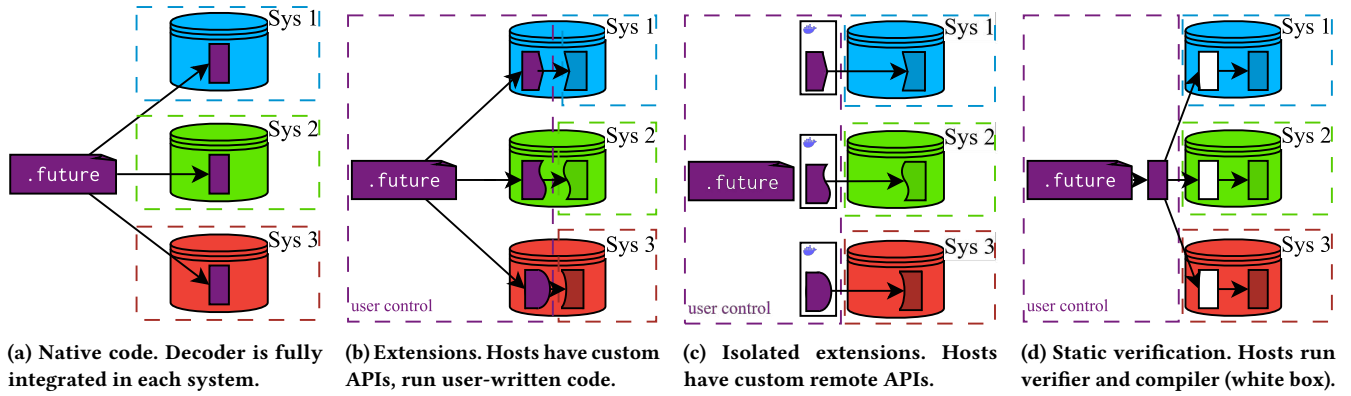


Figure 2: Supporting a .future format in three different Hosts according to each of the analyzed approaches.

with WebAssembly bytecode decoders, allowing any consumer to read any data encoding without knowledge of its details:

- Section 3 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 3.6 describes our novel **memory management scheme**, which allows zero-copy transfer of data between decoder and database system.
- Section 4 shows the integration of AnyBlox into a **wide range of database systems**, which employ vastly different data processing paradigms.
- Section 5 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 6.

2 PROBLEM SPACE & RELATED WORK

We identify four key properties that are desirable in a future-proof storage format abstraction:

- **Portability** – how easy is it to integrate the solution across different systems and hardware architectures?
- **Security** – what isolation guarantees does the solution provide? Can an error compromise the host system?
- **Performance** – how does the solution fare in terms of query latency and throughput?
- **Extensibility** – how easy is it to support a new data format with the solution?

State-of-the-art systems integrate data formats using a variety of approaches, which Figure 2 illustrates. All of these approaches have strengths, but also shortcomings that cause the problems identified in Section 1. The following section analyzes this in detail, and Table 1 outlines key findings. This analysis informs the design of AnyBlox, which we introduce in Section 3.

In the following, we will refer to a system ingesting data as the **Host**, and the algorithm required to read encoded data as a **Decoder**. The Host is an arbitrary data-processing system; it can be a simple tool printing decoded data to a terminal, an interactive notebook, or even a complete database system. The Decoder is an arbitrary algorithm that takes data in some specific encoding as input and produces data in a specified output format; the input encoding is unrestricted – data can be row- or column-encoded, nested, unstructured, or even virtual, e.g., an algorithm which produces TPC-H table data for a given scale factor.

2.1 Approach: Native Code

Today, most Hosts integrate new formats by adding a native implementation directly into their codebase. That is how databases support new compression schemes [36], how the Apache ORC file format [86] proliferates [101], and how JSON (Javascript Object Notation) [29] made its way into the database world [35, 38, 64].

Using native integrations, authors of an encoding scheme need to put the Decoder code into the existing codebase of every Host they wish to support (c.f., Figure 2a). This directly leads to unsustainable $N \times M$ development effort for N systems and M formats.

This approach suffers from obvious **Portability** issues, as a Decoder in one Host cannot be reused in another. Similarly, it provides weak **Extensibility**, i.e., it is difficult to support a new format in an existing Host. Its main advantage is **Performance**, since the tight integration allows the implementation utilize all Host internals. A well-written native integration thus provides an *upper bound* for the performance of other approaches.

Table 1: Evaluation of different approaches according to our design dimensions. We analyze AnyBlox in Section 3.

Approach	Portability	Security	Performance	Extensibility
native	---	-	+++	---
extensions	-	---	+++	+/-
isolated extensions	+/-	+++	-	+
static verification	+	+/-	+/-	-
AnyBlox	+++	+	+	+++

While native Decoders don't inherently expose new attack vectors, the integration of a dedicated Decoder for each format into the system core increases the likelihood of exploitable bugs. The effects of a bug in the Host's core are unrestricted and can range from crashing the process to data corruption, which is especially dangerous in case of systems written in memory-insecure languages like C/C++. The dependencies required by the Decoder are not isolated, exacerbating supply chain issues. Since Hosts require M integrations for M different formats, a security vulnerability becomes statistically inevitable. For example, adding JSON support to the widespread PostgreSQL database system [89] led to a high-severity security vulnerability discovered over 5 years later [73].

2.2 Approach: Extensions

To provide a more open platform and alleviate the issues inherent to the previous approach, some database systems allow users to extend their capabilities with custom code (c.f. Figure 2b). For example, PostgreSQL, the DuckDB [28] and Microsoft SQL Server [63] database systems allow dynamic loading of binary modules written in C/Rust, C++, and C#, respectively. A well-designed extensibility API allows Decoders to achieve near-native **Performance** by allowing implementers to leverage internal Host structures, implement predicate pushdown, etc., as exemplified by DuckDB's vectorized Parquet reader extension.

Adding a new Decoder as an extension requires some work, but not nearly as much as in the native case, thereby improving **Extensibility**. **Portability**, however, is still weak. Every Host has a different extensibility API, and each Decoder extension has to be maintained separately. Moreover, porting a DuckDB extension to a system built for the JVM (Java Virtual Machine) [66], such as the Apache Spark analytics engine [85], would require a nearly complete rewrite in a different tech stack.

Extensions are usually shared binary modules (e.g., an .so file, a .jar file) that are dynamically loaded by the system. This presents the worst case scenario for **Security** as it amounts to running arbitrary code at the same privilege level as the Host, leading to an immediate and inherent RCE (Remote Code Execution) vulnerability. Even even a malfunctioning *trusted* extension can cause arbitrary damage inside the process.

2.3 Approach: Isolated Extensions

In recent years a number of solutions for running arbitrary User Defined Functions – UDFs – with isolation guarantees have been proposed, allowing custom user data processing code to run in an environment separate from the Host. For example, Saur et al. [74] studied *containerized UDFs*, where the untrusted data processing code runs in an isolated environment via Docker [27] containers, though other mechanisms like virtual machines behave similarly. Data warehouses Snowflake [79] and Amazon Redshift [10] implement a similar approach, where user-defined code can be located in the cloud, registered with the Host, and called remotely.

In the isolated scenario, the authors of the encoding scheme need to place their Decoder in the Host's isolation mechanism *and* implement the communication protocol for receiving requests and sending data back to the Host (c.f., Figure 2c). This approach is more **Extensible** and **Portable** than integrated extensions, as

Decoders are isolated and hosted independently. However, vendor-specific constraints still exist (e.g., Redshift forces usage of their own proprietary FaaS service, exacerbating vendor lock-in). **Security** guarantees are excellent, as the extension is fully isolated has no access to data that is not explicitly sent to it.

As Saur et al. show, **Performance** is mainly constrained by the data transfer between the Host and the isolated extension [74]. While the authors report 10% overhead for data transfer to a locally-hosted Docker container using the Arrow Flight [87] format for an ML classification workload, our experiments show that a Decoder workload is less favorable: In run-length encoding, which is a data-intensive, but computationally lightweight scheme, decoding using a vectorized Decoder is fast (6.5 GB/s), but data transfer (1.5 GB/s) dominates the runtime (80%). This shows that low-overhead communication protocols like Arrow Flight can alleviate, but not entirely eliminate data transfer cost. Additionally, the Host may miss out on optimization opportunities e.g. because of the high *logical* isolation between Host's optimizer and runtime, and the Decoder.

2.4 Potential Approach: Static Verification

In search of combining good performance *and* security guarantees, we again turn to compiler research as inspiration. Programming languages generally improve security using either a *memory-safe runtime* (Java, C#, Elixir, etc.) or *static verification* at compile time (Rust, uBPF, etc.). In the following section, we discuss the merits and drawbacks of static verification, and why we ultimately decided on a different approach.

To load Decoder extensions, the Host system has to run arbitrary code inside its own process. The standard mechanism is to *verify* the code first and then *compile* it once it is proven to be safe (c.f. Figure 2d). One of the prominent, *eBPF*, can be used to run custom code in the linux kernel for packet filtering [93], programming NVM storage devices [37, 47], and querying data structures [102]. One can use uBPF, the user space implementation of eBPF, to verify arbitrary decoders and securely run them in the host process.

The uBPF toolchain consists of the eBPF bytecode, a C-to-eBPF compiler, a static verifier for eBPF, and a JIT (just-in-time) compiler allowing **Portable** execution on x86 and ARM. On paper, the verifier also guarantees safe memory accesses, termination, prohibits any system calls, and isolates the decoder from the network and file system. In practice, however, the verifier has bugs and allows memory-insecure programs to pass [14, 67, 84, 94, 97, 103]. Thus, the **Security** of this approach is better than that of native extensions, but not satisfactory without further verification research.

The **Performance** of uBPF is great *in theory*, since it is JIT-compiled into native x86 or ARM assembly. Our experiments show, however, that it suffers from (1) pervasive bounds-checks, (2) limited loop support, and (3) no vectorized instructions. Finally, **Extensibility** suffers from the difficulty of correctly implementing an eBPF algorithm to pass the verifier. The C dialect required is unwieldy, cannot utilize existing libraries tailored for eBPF, and the verifier requirements often clash with compiler optimizations. For example, the C-to-eBPF compiler may elide bounds checks that it deems redundant, causing the verifier to reject the eBPF code.

In conclusion, a statically-verified language for Decoders is a promising idea, but requires further research. In particular, such a

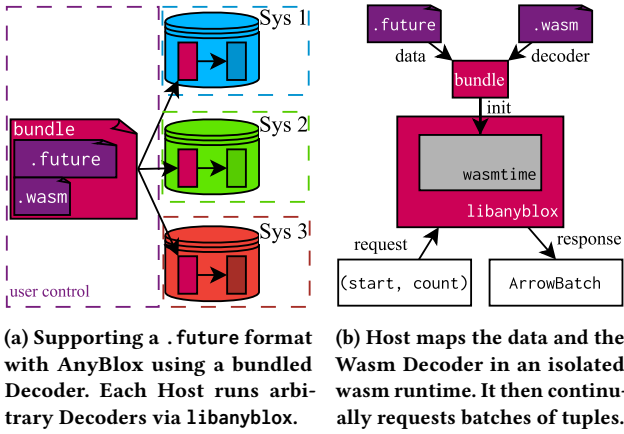


Figure 3: Integration of AnyBlox, Hosts, and a .future format.

language would need to limit its expressiveness, so that the verifier and compiler can be small and easy to audit for correctness, while also remaining expressive enough to not limit extensibility.

3 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: In the data file 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 .any 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 3.1), the format of data returned to the Host (Section 3.2), and the metadata contained in the self-decoding dataset (Section 3.3), as shown in Figure 3.

3.1 WebAssembly Decoders

WebAssembly (Wasm) is a specification of a virtual machine and its portable bytecode [34]. Its strength lies in a machine-verifiable guarantee on memory-safety and isolation from the host system [96]. It has been successfully utilized for isolation in different data processing applications, e.g. for ML workloads in Big Query [55],

stateful serverless FaaS [76], and UDFs in databases [77], which makes WebAssembly a prime candidate for self-decoding data sets.

3.1.1 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 4, AnyBlox easily integrates into a variety of existing Hosts.

3.1.2 Security. WebAssembly’s specification is verified to uphold isolation guarantees [96]. Naturally, the guarantees depend on the *implementation* of the specification being correct. AnyBlox utilizes the Wasmtime implementation and the Cranelift compiler to lower the Wasm bytecode to the native instruction set [17, 18]. Recent work suggests that the main source of vulnerabilities in WebAssembly is the compiler, namely the instruction selection and lowering process [92]. 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 [15].

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.

3.1.3 Performance. Wasm in theory 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 3.6). While we devote Section 5 to performance evaluation, let us note here that research into closing the gap between Wasm and native code is both extensive and ongoing [41, 80]. In 2019 Jangda et al. identified instruction selection as the main source of performance degradation [40], and Yan et al. argue that common LLVM optimizations are ineffective when applied to WebAssembly [98]. 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.

3.1.4 Extensibility. Complex decoding schemes are easily implementable, as most general-purpose programming languages have a WebAssembly toolchain and compiler, including popular backends like LLVM, JVM, and CLI. 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 [54], DELTA, dictionary, RLE [26], as well as data encodings like JSON [51]. Unlike eBPF, 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 as the JIT compiler selects the appropriate instructions for the Host machine.

3.2 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, 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.

3.3 Metadata

The ideal scenario is that data is distributed as a single self-decoding file, necessitating a thin metadata layer in .any files. 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 when using AnyBlox data sources (see Section 4.5).

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 an .any file it can compare its Decoder URI against a list of well-known URIs and instead decode the payload using its native Decoder.

3.4 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. This establishes 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 3b.

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 3.5). Then, the Host can repeatedly request arbitrary tuple ranges, which are decoded into **Apache Arrow Record Batches**.

The architecture is illustrated in Figure 4. To aid our explanation, let us pick an encoding scheme and explain how AnyBlox allows one to use datasets encoded with it in an arbitrary Host that integrates with libanyblox. We will examine **run-end encoding** of a single column of 32-bit unsigned integers. The data is compressed by finding runs of repeating values and encoding them as a pair (*lastRid*, *value*), where *rid* is a sequence number assigned to each row from 0 to *N* and *N* is the total length of the column:

rid	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
val	3	3	3	1	2	4	4	4	4	4	1	1	1	1	1	1
REE	3	2	1	3	2	4	9	1	15							

In the following let us assume that we compressed 10^8 tuples and the encoded dataset contains 10^6 run-end pairs

3.4.1 First batch. Decoding begins by request from the Host, usually when a table scan is scheduled on an AnyBlox data source. The Host might decide to split the work across multiple threads, either at plan time [32] or at runtime [53]. Let us assume some thread is assigned the tuple range [400 000..500 000). The Host first creates a decode job via the AnyBlox API, providing a file descriptor to an open dataset. AnyBlox receives the Decoder bytecode and JIT-compiles it if needed. This compilation is cached, so future uses of the exact same Decoder incur no compilation cost. AnyBlox initializes the WebAssembly sandbox, creates the linear memory chunk accessible for the Decoder, hooks the dataset contents via another memory map, and links the WebAssembly module with the implementation of the `memory.grow(u32)` function which the Decoder can use to allocate a given number of pages.

Control returns to Host, at which point it can start the scan. Modern systems process data in batches to amortize operator communication overheads [32, 53], so the Host requests AnyBlox to decode a batch of tuples (e.g., 10^4), calling the job with parameters *startTuple* = 400 000, *tupleCount* = 10 000¹. AnyBlox delegates this into the WebAssembly module, calling its `decode_batch` function.

When the Decoder is called for the first time it needs to find its bearing in the dataset. To locate the run containing the tuple with *rid* = 400 000 it binary-searches the run-end encoded dataset. After that the Decoder can be implemented by keeping a state (*rid*, *i*, *val*, *rem*), where *rid* is the ID of the next row to return, *i* is the index of the run encoding pair, *val* is the value of the current run, and *rem* is the remaining number of rows in the run. The Decoder loop is then:

```
if rid == start_tuple + tuple_count: return batch;
else if rem > 0: result.add(val), rid += 1, rem -= 1;
else: i += 1, (val, rem) = read pair at index i;
```

To put all values into the result, the Decoder needs to allocate memory. It knows the required length *a priori*, since it needs to output *tupleCount* = 10 000 32-bit integers. This requires 40 000 bytes, plus some overhead for the Arrow Batch structure. In WebAssembly all allocations are performed in full pages, and a page is 64 KiB. The Decoder thus issues a `memory.grow(1)` call at the start. The Host processes the request and returns the index of the newly allocated page. The Decoder can put all the results in that space, and returns a pointer to the completed Arrow Batch as the result of the `decode_batch` call.

AnyBlox translates the pointers from WebAssembly into native addresses for all buffers in the batch and then returns it back to the Host, which processes the Arrow payloads according to its own logic in the AnyBlox operator.

¹This batch size originates from the morsel-driven parallelism paper, which specifies 10^4 as a good morsel size for a simple scan over an integer column in their setup [53].

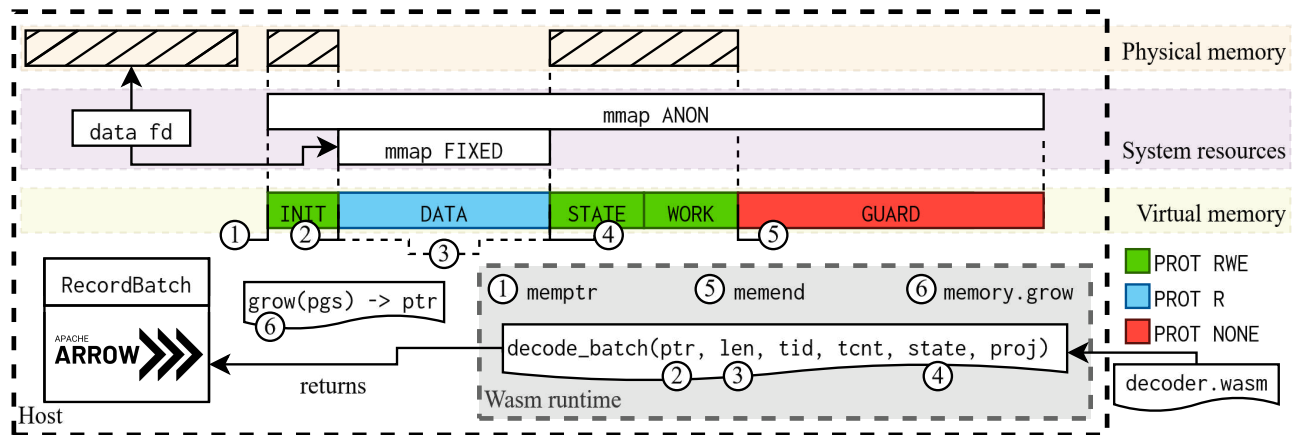


Figure 4: Diagram of an integrated AnyBlox runtime and a Decoder instance. The sandboxed Decoder has access only to the pictured anonymous memory map. Relevant parameters labelled with numbers: (1) pointer to the start of the linear memory; (2) pointer to the start of the encoded data; (3) length of the encoded data; (4) pointer to the state page; (5) end of currently accessible linear memory; (6) AnyBlox-provided memory allocation function.

3.4.2 Future batches. If the Decoder was stateless, calling for another batch ($startTuple = 410\,000$) would require redoing the binary-search and memory allocation. AnyBlox optimizes this by allocating a page for the Decoder to keep arbitrary state in, which it passes as one of the arguments to `decode_batch`. For example, our REE Decoder can keep the (rid, i, val, rem) state on that page, as well as a pointer to its existing Arrow Batch allocation. Once it receives a call and sees that $startTuple = rid$, it can directly resume the main loop, avoiding the initial binary-search. It can also reuse the memory allocated previously, avoiding the `memory.grow` call and improving the performance of the Decoder.

3.5 Decoder API

The Decoder needs to expose only one function, `decode_batch`, 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 length in bytes 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 projection_mask`, a bitmask specifying columns to decode (column projection).

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 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.

3.6 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 [18]. Our core technical contribution is custom linear memory management for 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 0×0 . 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` ought to be provided by the Host, and the standard does not impose a specific mechanism for memory management on the Host’s side.

By default, Wasmtime implements the specification using a well-known Software Fault Isolation technique for 32-bit pointers [30]: 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 maximum 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 allocation 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 operation requires few system calls and is fast even for large datasets. The Decoder can transparently access this data in the same manner as any other area of its linear memory. The overall layout is shown in Figure 4. This approach is similar to how Faasm handles shared memory regions by swapping memory maps [76], 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`. 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 [57]. 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, but it is an interesting technique for future work.

3.7 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 *byte-code* 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 (as discussed in Section 2.4), 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 Spark in its 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 that we discuss in the following. First, WebAssembly by default has a 32-bit address space into which AnyBlox maps datasets, meaning each mapped file has a maximum size of 4GB. WebAssembly recently gained support for using 64-bit addresses [19], but the performance impact of enabling this option is not yet well studied. Most systems using, for example, Parquet, keep file sizes well below 4 GiB [5, 78, 88] 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 ($> 128\text{bit}$) vector instructions. Section 5 studies the performance impact of this limitation.

4 INTEGRATION

To demonstrate the portability of our solution we have integrated them into four fundamentally different systems: **DataFusion** [50], **DuckDB** [71], **Umbra** [65], and **Spark** [99]. 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. In particular we cover differences in:

- **Core paradigms** – Umbra is a compiling system, and DuckDB and DataFusion are vectorized systems – two very different paradigms [43]. Spark focuses on distributed workloads, and only generates code for expressions.
- **Parallelism** – Umbra and DuckDB are both morsel-driven with a central executor [53], while Spark and DataFusion use Volcano-style parallelism with exchange operators [32].
- **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 performance and provides no dedicated extension points; DataFusion was published 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.

4.1 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. Moreover, 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 throughout the engine [50], allowing truly zero-copy data sharing with AnyBlox. DuckDB and Spark, like many modern systems [75, 90], 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 [65], 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.

4.2 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.

4.3 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 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 can keep local state, and, as an optimization, they can request large batches with `decode_batch` and then process them in smaller chunks, e.g., the DuckDB global vector size.

4.4 Embedded AnyBlox: Future-Proof Parquet

AnyBlox can also be embedded inside existing file formats, allowing them to evolve without breaking compatibility. We evaluate this by integrating AnyBlox as an encoding in Parquet. In this scenario, the Host is a Parquet decoding library, specifically Apache’s Rust `parquet` crate [6]. Similarly to how Parquet provides dictionary encoding with a single dictionary page and multiple data pages [9], we prototype a scheme where a column starts with a *Decoder* page and then opaque binary pages that can be decoded by AnyBlox with the provided Decoder. While this approach needs further work, e.g., to support record shredding [61], it shows large potential in both compression rates and throughput, which we show in Section 5.1.3. Enabling Parquet’s generic compression on top of the AnyBlox encoding can further compress the data as well as the embedded Wasm bytecode. This approach allows arbitrary, instance-optimized column encodings to be distributed alongside conventionally encoded columns inside a single Parquet file.

4.5 Embedded AnyBlox: Statistics & Pushdown

As Section 4.4 shows, AnyBlox can bring arbitrary, cutting-edge encoding schemes to existing file formats such as Parquet. These

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

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

file formats usually store additional statistics or query optimizer hints, which query engines use to improve performance. Scans from remote object stores (e.g data lakes), can benefit from min/max values per row group to avoid scanning all of the data. Unique value count statistics for individual columns or blocks can facilitate better query plans, and other specialized formats may benefit from entirely different metadata. When AnyBlox is embedded into other file formats such as Parquet at the row-group or column level, query engines can benefit from their existing statistics utilization, column-level filtering, and row-group-level filtering without change.

4.6 Standalone AnyBlox: Statistics & Pushdown

Without integration into other file formats, standalone AnyBlox only encodes the schema and row count of the encoded data. Further optimizer hints can be added (c.f., Section 3.7), but as a generic framework, AnyBlox does not try to provide all statistics that any system might need. Instead, we follow recent work [49, 70] in suggesting that most statistics and indexes should be decoupled from storage formats and put into query engines or search acceleration layers. AnyBlox allows query engines to scan and build statistics over arbitrary formats, and is therefore a step towards this direction.

Following a similar argument, standalone AnyBlox does not provide a predicate pushdown API: While filtering columns (projection pushdown) is supported, filtering individual rows in a future-proof framework such as AnyBlox requires an equally future-proof and system-independent predicate expression format. Recent work on Substrait [83] highlights industry interest in this topic. Finally, column projections cannot be pushed to remote files since standalone AnyBlox requires the entire data set to be mapped into memory. Section 3.6 discusses a potential workaround for this issue.

5 EVALUATION

While the security guarantees of AnyBlox follow from the WebAssembly sandbox and portability is proven by our system integration in Section 4, 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 different decoders, OLAP workloads, and microbenchmarks.

Experiments are performed on a 32-core Intel Xeon Gold 6430 machine with 256 GB of DDR5 RAM (data rate 4400 MT). Unless stated otherwise, samples are collected after two warm-up runs, which isolates the overhead of disk reads, Decoder compilation, and thread initialization. Section 5.5.5 examines these overheads in detail. We present median results, but unaggregated results are available in our repository.

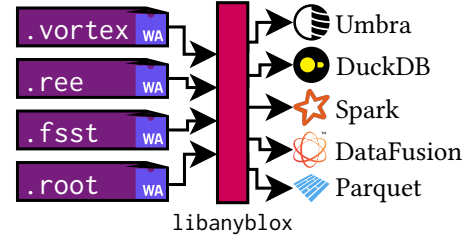


Figure 5: AnyBlox bridges the gap between a variety of storage encodings and arbitrary data processing systems.

5.1 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 and REE, (2) Vortex [81], a novel columnar file format using state-of-the-art compression research [1, 2, 13, 49], and (3) CERN ROOT, a specialized binary format storing exabytes of particle physics data [20]. 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 5 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'
```

5.1.1 Vortex. Vortex [81] is a novel format based on BtrBlocks [49] and recent encoding research [1, 2]. 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.

5.1.2 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 [20]. Though much of this data is tabular, ROOT itself is not a relational format and allows encoding arbitrary C++ 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 [33]. Our AnyBlox decoder (based on a Rust port [16] for a large subset of ROOT) instead makes ROOT files immediately readable by Spark, DuckDB, and Umbra – no conversion required. Single-threaded

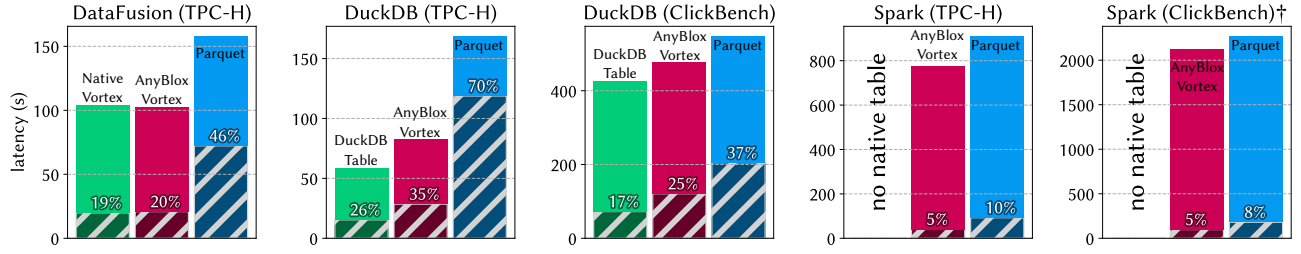


Figure 6: TPC-H and ClickBench evaluated on a single thread with DuckDB, Spark[†], and DataFusion. Time spent in the scan of the main table is shaded.

Umbra surpasses the throughput of the native ROOT framework when performing an aggregation on particle decay data [56]. More importantly, Umbra’s parallel processing automatically works for ROOT files, yielding 6 GB/s throughput 32 threads, while the native framework offers no easy mechanism for parallelism.

5.1.3 Parquet. As Section 4.4 describes, we add AnyBlox as a Parquet encoding such that individual column chunks can use arbitrary WebAssembly decoders. We benchmark this experimental Parquet integration by implementing the standard Parquet RLE+bitpacking hybrid decoder [9] as a Wasm decoder. We compare its throughput on a compressed `l_lineitem` column of TPC-H against Apache’s native Rust Parquet reader [6] and observe a mere 10% throughput drop. We also test the Vortex Decoder on the ClickBench dataset as in Section 5.3.2 to validate the viability of our prototype. AnyBlox achieves a similar compression rate while improving the scan throughput over native Snappy by 25%.

5.1.4 FSST & Run-end encoding. FSST [13] and REE are lightweight encoding schemes that can be easily implemented in any language that compiles to WebAssembly. Both are also implemented in Vortex [81]. We use standalone Rust implementations in the decompression microbenchmarks performed in Section 5.5.

5.2 Vortex Native vs. Vortex in AnyBlox

We begin by showing that (1) AnyBlox can be used to provide both compression *and* performance improvements by introducing novel encoding schemes; and (2) AnyBlox’s sandboxing mechanism introduces acceptable overheads with 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 6, left), and it provides a better compression rate (c.f., Figure 7). 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.

5.3 DuckDB Performance

Next, we test AnyBlox in a full database system – DuckDB. We again compare AnyBlox Vortex to a native Parquet reader, but also include comparison against the internal, non-portable native table format of DuckDB. Furthermore, both Parquet and native DuckDB table are allowed to use predicate pushdown, to fairly portray AnyBlox’s disadvantage in this area as discussed in Section 4.5. The performance results are shown in Figures 7, 6, and 8.

5.3.1 TPC-H. Across all queries and Parquet configurations, the more efficient Vortex encoding outperforms Parquet despite the WebAssembly overhead, and AnyBlox is often close to the native baseline. In the single-threaded `lineitem` scan in Figure 6, both the native format and AnyBlox can 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 8 shows.

5.3.2 ClickBench. TPC-H is an important standard benchmark, but fails to represent the real world, especially when it comes to string data [31, 49, 95]. We therefore turn to the ClickBench benchmark [24], 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 [13] 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 7 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

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

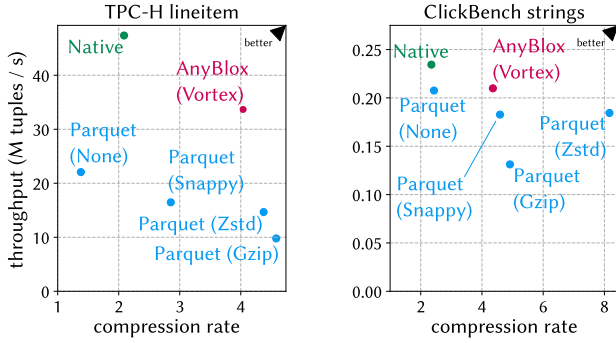


Figure 7: Compression ratio vs workload throughput on DuckDB for native tables, AnyBlox, and all Parquet compression codecs supported by DuckDB.

focusing on string processing. In most queries, total processing time vastly outweighs scan times, with Q29, which evaluates a complicated regular expression on URLs, as a prime example. 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 all tested native Parquet readers, as the following table shows (in millions of tuples per second, full 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 [60]. We further analyze AnyBlox’s scalability in Section 5.5.4.

5.4 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 6 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.

5.5 Decoder Microbenchmarks

To better lay out the performance characteristics of our solution, we now zoom in on specific decoders and compare them with equivalent native implementations. We compile the same code into a native module (using the standard LLVM Rust compiler), and into a Wasm module. This experiment isolates the impact of the Wasm-based design and the Cranelift compiler on raw Decoder throughput.

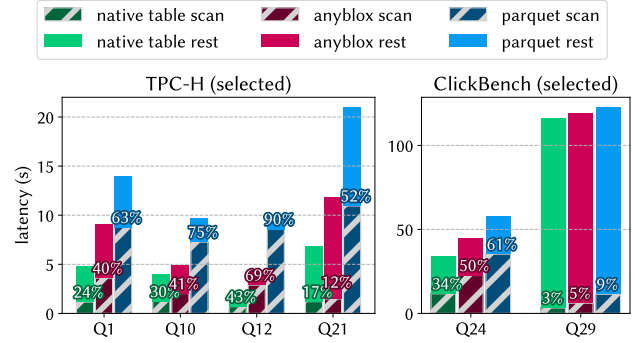


Figure 8: Selected queries from DuckDB TPC-H and ClickBench evaluation. Other systems exhibit similar behavior. Time spent in the scan of the main table is shaded.

We first restrict ourselves to a single thread decoding the dataset in a sequential scan. The results are presented in Figure 9.

The Faasm paper identifies similar worst-case throughput decreases (50% to 66%), but the results are not directly comparable due to a different Wasm runtime and compiler choice. The authors similarly conclude, however, that the large differences in microbenchmarks do not translate to equivalent slowdowns in end-to-end workloads where other overheads dominate [76].

5.5.1 Run-end encoding. We revisit the run-end encoding examined in Section 3.4. Using the *CommonGovernment* dataset available in the Public BI Benchmark [31], we compress its `psc_key` integer column, achieving a compression ratio of roughly 2.62 – approximately 141 million rows get compressed into 27 million run-end-encoded pairs. We implement the previously described decoding logic and compile it into the native module (standard Rust LLVM compiler) and into wasm (Wasm bytecode). We measure the time it takes to decode the entire column sequentially in batches of 200 000. The native code performs $\approx 40\%$ faster in this comparison.

We re-implement the Decoder using vectorized instructions. First we compare the only available Wasm SIMD instruction set, V128, against the equivalent SSE2 on x86. Both utilize similar instructions after Cranelift lowers the Wasm code. The SIMD Wasm code is 2.5× faster than the scalar Wasm code, highlighting the importance of vectorized instructions. The native SSE2 version is $\approx 40\%$ faster than the Wasm V128 version. We further compare to native AVX2 and AVX512 versions, which utilize larger vector sizes and outclass the SSE2 and the Wasm version. We note small vector sizes are not a fundamental limitation of WebAssembly, and there is already a proposal for larger vector sizes in WebAssembly [69].

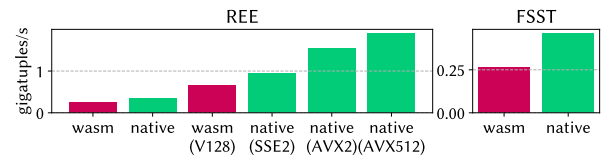
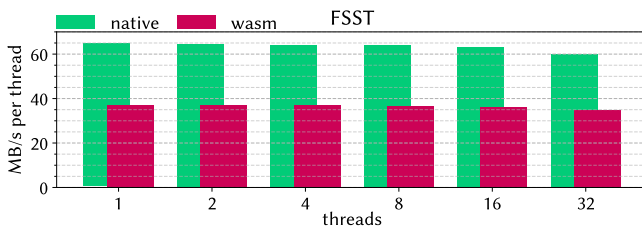


Figure 9: Microbenchmarks of REE and FSST decoders.

5.5.2 FSST. FSST is a simple but effective string compression mechanism proposed in 2020 [13]. Some database systems use it in their internal storage format for tables [65, 71]. It is also used in BtrBlocks [49], and, by extension, Vortex [81]. We compress three key string columns from the Taxpayer dataset from the Public BI Benchmark – these are the columns used in all queries for that dataset. We store both the compressed string data and the buffer of string lengths that is not compressed further. This results in a 1.7 GB AnyBlox file that decompresses into 2.5 GB of Arrow batch payloads in memory. The Wasm decoder loses out by over 43% on this benchmark. FSST is one of the worst cases for Wasm, because it heavily relies on 8-byte reads in its implementation, and Wasm is 32-bit, which forces it to emulate one 8-byte read with two 4-byte reads. Faasm authors also identified this as incurring significant overheads in workloads with heavy large integer arithmetic [76]. This is a fundamental limitation of our approach, as even though a proposal for 64-bit version of WebAssembly exists, it would be incompatible with the SFI mechanism relying on only 8 GiB of memory being addressable.

5.5.3 Batch size. For both of the above Decoders, we investigate the impact of the requested batch size on performance and conclude the results have similar characteristics to vector sizes in vectorized databases [43] and batch sizes in Volcano-style exchanges [32] – the size has to be big enough to amortize call overheads, but increasing it past a certain point brings diminishing returns and requires more work memory. The optimal size varies, but all tested Decoders exhibit stable performance between batches of size 10^4 to 10^7 .

5.5.4 Scaling. Decoding a dataset is embarrassingly parallelizable, and the experiments confirm that scaling is – outside of synchronization of threads and the Wasmtime runtime – nearly perfect:



5.5.5 Initialization. Finally, we want to drill into the fine-grained performance profile of a Decoder job. Many things happen on the way from Host to Decoder and back: The Wasm code has to be compiled, the memory regions initialized, the dataset mapped into the region, and control switched to call into Wasm mediated by Wasmtime. AnyBlox heavily caches every step.

Our REE and FSST Decoders compile in around 2 ms and produce binaries around 30 kB in size. Still, it is important for AnyBlox to cache each compiled decoder to avoid paying this overhead. In practice, we expect systems to often reuse the same Decoders.

When a thread creates an AnyBlox job for the first time, it needs to create a thread-local 8 GiB memory map for the Wasm linear memory. The kernel creates a mapping for each page in that region, which takes between 3 ms and 10 ms (median 8 ms) with 32 threads initializing simultaneously. Since threads reuse memory mappings, this cost is only paid when the Host spins up a new thread.

When loading a new dataset, a thread needs to unmap the existing file descriptor and hook the new descriptor. This takes less

time than a full thread initialization – 700 μ s to 800 μ s for 32 concurrent threads initializing simultaneously. To remove this overhead, integrations can reuse the same open dataset across runs.

Every new job needs to reset the memory it will use by zeroing the state page and removing left-over allocations from the previous job. This is the most optimized scenario and incurs little overhead even when multiple jobs initialize concurrently (tail latency of 250 μ s for 32 concurrent threads). The main contention occurs on the Wasm cache lock. Finally, Wasmtime sets up signal handlers and mediates the call into the compiled Wasm code via a trampoline. The overhead of this call is negligible – below 100 ns – and does not affect performance outside of excessively small batch sizes.

6 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. Files can bundle 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 [39, 58, 59], learned white-box compression [31], and entropy compression [72] 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 [46] – 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 [21, 44, 68]. LLVM showed that an open framework can revolutionize the design of complex modern systems like compilers [52]. Recently, Lamb et al. presented an open and extensible query engine based around composable system design principles [50], proving such an approach can be successful. Execution plan frameworks have been made more extensible using sub-operators [11, 42], portable query optimizers [3, 4], and system-independent persistence mechanisms [91]. Our work joins this effort with an extensible and modular solution for arbitrary storage formats.

ACKNOWLEDGMENTS

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, 101164-556; 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.

REFERENCES

- [1] Azim Afrozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (May 2023), 2132–2144. <https://doi.org/10.14778/3598581.3598587> Publisher: VLDB Endowment.
- [2] Azim Afrozeh, Leonardo X. Kuffo, and Peter A. Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data* 1, 4 (2023), 230:1–230:26.
- [3] Rana Alotaibi, Yuanyuan Tian, Stefan Grafberger, Jesús Camacho-Rodríguez, Nicolas Bruno, Brian Kroth, Sergiy Matushevych, Ashvin Agrawal, Mahesh Behera, Ashit Gosalia, César A. Galindo-Legaria, Milind Joshi, Milan Potocnik, Beysim Sezgin, Xiaoyu Li, and Carlo Curino. 2024. 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*. www.cidrdb.org, Amsterdam, The Netherlands. <https://doi.org/10.48550/ARXIV.2411.13704> arXiv: 2411.13704.
- [4] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3515–3527. <https://doi.org/10.14778/3611540.3611544> Publisher: VLDB Endowment.
- [5] Apache Parquet. 2025. Recommended Parquet File Size on HDFS. <https://parquet.apache.org/docs/file-format/configurations/>
- [6] Apache Software Foundation. 2018. parquet. <https://github.com/apache/arrow/tree/main/parquet>
- [7] Apache Software Foundation. 2023. Apache Arrow. <https://arrow.apache.org/>
- [8] Apache Software Foundation. 2023. Apache Arrow Columnar Format - Variable-size Binary View Layout. <https://arrow.apache.org/docs/format/Columnar.html#variable-size-binary-view-layout>
- [9] Apache Software Foundation. 2024. Encodings | Parquet. <https://parquet.apache.org/docs/file-format/data-pages/encodings/>
- [10] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinsky, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. <https://doi.org/10.1145/3514221.3526045> event-place: Philadelphia, PA, USA.
- [11] Maximilian Bandle and Jana Giceva. 2021. Database technology for the masses: sub-operators as first-class entities. *Proc. VLDB Endow.* 14, 11 (July 2021), 2483–2490. <https://doi.org/10.14778/3476249.3476296> Publisher: VLDB Endowment.
- [12] Allison Black, Duncan R. MacCannell, Thomas R. Sibley, and Trevor Bedford. 2020. Ten recommendations for supporting open pathogen genomic analysis in public health. *Nature Medicine* 26, 6 (June 2020), 832–841. <https://doi.org/10.1038/s41591-020-0935-z>
- [13] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proc. VLDB Endow.* 13, 12 (July 2020), 2649–2661. <https://doi.org/10.14778/3407790.3407851> Publisher: VLDB Endowment.
- [14] Daniel Borkmann. 2023. bpf: Fix incorrect verifier pruning due to missing register precision taints. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=71b547f561247897a0a14f3082730156c0533fed> Publisher: Linux kernel source tree.
- [15] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1975–1992. <https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya>
- [16] Christian Bourjau. 2018. mALICE: An open source framework for analyzing ALICE's Open Data. <https://github.com/cbourjau/alice-rs/>
- [17] Bytecode Alliance. 2016. Cranelift. <https://cranelift.dev/>
- [18] Bytecode Alliance. 2019. wasmtime. <https://github.com/bytecodealliance/wasmtime>
- [19] Bytecode Alliance. 2025. Memory64 Proposal for WebAssembly. <https://github.com/WebAssembly/memory64/>
- [20] CERN. 1995. CERN ROOT Format. CERN. <https://root.cern/>
- [21] Surajit Chaudhuri and Gerhard Weikum. 2000. 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)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1–10.
- [22] Chaoran Chen, Sarah Nadeau, Ivan Topolsky, Niko Beerenwinkel, and Tanja Stadler. 2022. Advancing genomic epidemiology by addressing the bioinformatics bottleneck: Challenges, design principles, and a Swiss example. *Epidemics* 39 (June 2022), 100576. <https://doi.org/10.1016/j.epidem.2022.100576> Place: Netherlands.
- [23] Chaoran Chen, Alexander Taepper, Fabian Engelniederhammer, Jonas Kellerer, Cornelius Roemer, and Tanja Stadler. 2023. LAPIs is a fast web API for massive open virus sequencing data. *BMC Bioinformatics* 24, 1 (June 2023), 232. <https://doi.org/10.1186/s12859-023-05364-3>
- [24] ClickHouse. 2022. ClickBench. <https://github.com/ClickHouse/ClickBench/>
- [25] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (June 1970), 377–387. <https://doi.org/10.1145/362384.362685> Place: New York, NY, USA Publisher: Association for Computing Machinery.
- [26] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *International Conference on Extending Database Technology*. OpenProceedings, Venice, Italy, 72–83. <https://api.semanticscholar.org/CorpusID:1171780>
- [27] Docker Inc. 2025. Docker: Accelerated Container Application Development. <https://www.docker.com/>
- [28] DuckDB Foundation. 2025. The DuckDB database system. <https://duckdb.org/>
- [29] Ecma Technical Committee 39. 1997. The JSON Format. <https://www.json.org/>
- [30] Bryan Ford and Russ Cox. 2008. Vx32: lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, USA, 293–306. event-place: Boston, Massachusetts.
- [31] Bogdan Vladimir Ghita, Diego G. Tomé, and Peter A. Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *Conference on Innovative Data Systems Research*. <https://api.semanticscholar.org/CorpusID:210706292>
- [32] Goetz Graefe. 1990. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. Association for Computing Machinery, New York, NY, USA, 102–111. <https://doi.org/10.1145/93597.98720> event-place: Atlantic City, New Jersey, USA.
- [33] Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T. Watts, and Gustavo Alonso. 2021. Evaluating query languages and systems for high-energy physics data. *Proc. VLDB Endow.* 15, 2 (Oct. 2021), 154–168. <https://doi.org/10.14778/3489496.3489498> Publisher: VLDB Endowment.
- [34] 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 (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. <https://doi.org/10.1145/3062341.3062363> event-place: Barcelona, Spain.
- [35] Robert Haas. 2012. Built-in JSON data type. <https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=5384a73f98d9829725186a7b65bf4f8adb3cfa1>
- [36] Robert Haas. 2021. Allow configurable LZ4 TOAST compression. <https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=bbe0a81db>
- [37] Niclas Hedam, Morten Tycheisen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. 2023. Delilah: eBPF-offload on Computational Storage. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*. ACM, Seattle WA USA, 70–76. <https://doi.org/10.1145/3592980.3595319>
- [38] Dwayne Richard Hipp. 2022. Merge the JSON interface into the core. Add -> and -> operators for JSON that are compatible with by MySQL and PG. <https://sqlite.org/src/info/4cbb3e3efeb40cc4>
- [39] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1733–1746. <https://doi.org/10.1145/3318464.3389734> event-place: Portland, OR, USA.
- [40] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 107–120. <https://www.usenix.org/conference/atc19/presentation/jangda>
- [41] Shuyao Jiang, Ruiying Zeng, Zihao Rao, Jiazhen Gu, Yangfan Zhou, and Michael R. Lyu. 2023. Revealing Performance Issues in Server-Side WebAssembly Runtimes Via Differential Testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 661–672. <https://doi.org/10.1109/ASE56229.2023.00088>
- [42] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (July 2023), 3461–3474. <https://doi.org/10.14778/3611479.3611539> Publisher: VLDB Endowment.
- [43] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222. <https://doi.org/10.14778/3275366.3284966> Publisher: VLDB Endowment.
- [44] Amandeep Khurana and Julien Le Dem. 2018. The Modern Data Architecture: The Deconstructed Database. *login Usenix Mag.* 43 (2018), 36–40. <https://api.semanticscholar.org/CorpusID:56172282>

- [45] Sergey Knyazev, Karishma Chhugani, Varuni Sarwal, Ram Ayyala, Harman Singh, Smruthi Karthikeyan, Dhriti Deshpande, Pelin Icer Baykal, Zoia Comarova, Angela Lu, Yuri Porozov, Tetyana I. Vasylyeva, Joel O. Wertheim, Braden T. Tierney, Charles Y. Chiu, Ren Sun, Aiping Wu, Malak S. Abdalthagafi, Victoria M. Pak, Shivashankar H. Nagaraj, Adam L. Smith, Pavel Skums, Bogdan Pasaniuc, Andrey Komissarov, Christopher E. Mason, Eric Bortz, Philippe Lemey, Fyodor Kondrashov, Niko Beerenwinkel, Tommy Tsan-Yuk Lam, Nicholas C. Wu, Alex Zelikovsky, Rob Knight, Keith A. Crandall, and Serghei Mangul. 2022. Unlocking capacities of genomics for the COVID-19 response and future pandemics. *Nature Methods* 19, 4 (April 2022), 374–380. <https://doi.org/10.1038/s41592-022-01444-z>
- [46] Andrei N Kolmogorov. 1963. On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A* (1963), 369–376. Publisher: JSTOR.
- [47] Kornilios Kourtis, Animesh Kr Trivedi, and Nikolas Ioannou. 2020. Safe and Efficient Remote Application Code Execution on Disaggregated NVM Storage with eBPF. *ArXiv abs/2002.11528* (2020). <https://api.semanticscholar.org/CorpusID:211506592>
- [48] Laurens Kuiper. 2025. Query Engines: Gatekeepers of the Parquet File Format. <https://duckdb.org/2025/01/22/parquet-encodings.html>
- [49] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (June 2023), 118:1–118:26. <https://doi.org/10.1145/3589263> Place: New York, NY, USA Publisher: Association for Computing Machinery.
- [50] Andrew Lamb, Yijie Shen, Daniel Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data (SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 5–17. <https://doi.org/10.1145/3626246.3653368> event-place: Santiago AA, Chile.
- [51] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *VLDB J.* 28, 6 (2019), 941–960. <https://doi.org/10.1007/S00778-019-00578-5>
- [52] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE Computer Society, San Jose, CA, USA, 75–86. <https://doi.org/10.1109/CGO.2004.1281665>
- [53] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. 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)*. Association for Computing Machinery, New York, NY, USA, 743–754. <https://doi.org/10.1145/2588555.2610507> event-place: Snowbird, Utah, USA.
- [54] D. Lemire and L. Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29. <https://doi.org/10.1002/spe.2203> _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2203>
- [55] Justin Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery’s Evolution toward a Multi-Cloud Lakehouse. In *Companion of the 2024 International Conference on Management of Data (SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 334–346. <https://doi.org/10.1145/3626246.3653388> event-place: <conf-loc>, <city>Santiago AA</city>, <country>Chile</country>, </conf-loc>.
- [56] LHCb collaboration (2017). 2011. Matter Antimatter Differences (B meson decays to three hadrons) - Data Files. <https://doi.org/10.7483/OPENDATA.LHCb.AOF7.JH09>
- [57] Linux Kernel Developers. 2024. userfaultfd(2) Linux User’s Manual. <https://www.man7.org/linux/man-pages/man2/userfaultfd.2.html>
- [58] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. 2024. 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. <https://vldb.org/workshops/2024/proceedings/CloudDB/clouddb-2.pdf>
- [59] Xi Lyu, Andreas Kipf, Pascal Pfeil, Dominik Horn, Jana Giceva, and Tim Kraska. 2023. CorBit: Leveraging Correlations for Compressing Bitmap Indexes. In *VLDB Workshops*. <https://api.semanticscholar.org/CorpusID:261125284>
- [60] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! but at what cost?. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*. USENIX Association, USA, 14. event-place: Switzerland.
- [61] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2011. Dremel: interactive analysis of web-scale datasets. *Commun. ACM* 54, 6 (June 2011), 114–123. <https://doi.org/10.1145/1953122.1953148> Place: New York, NY, USA Publisher: Association for Computing Machinery.
- [62] Microsoft. 2016. Microsoft LSP. <https://microsoft.github.io/language-server-protocol/>
- [63] Microsoft. 2025. Microsoft SQL Server. <https://www.microsoft.com/sql-server>
- [64] Microsoft Corporation. 2024. JSON data in SQL Server. <https://learn.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver16>
- [65] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research*. www.cidrdb.org, Amsterdam, The Netherlands. <https://api.semanticscholar.org/CorpusID:209379505>
- [66] Oracle. 2013. The Java Virtual Machine Specification. <https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>
- [67] Manfred Paul. 2021. CVE-2021-3490. <https://nvd.nist.gov/vuln/detail/CVE-2021-3490>
- [68] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (June 2023), 2679–2685. <https://doi.org/10.14778/3603581.3603604> Publisher: VLDB Endowment.
- [69] Petr Penzin. 2020. Flexible Vectors Proposal for WebAssembly. <https://github.com/WebAssembly/flexible-vectors>
- [70] 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*. www.cidrdb.org, Amsterdam, The Netherlands. <https://api.semanticscholar.org/CorpusID:275548402>
- [71] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. <https://doi.org/10.1145/3299869.3320212> event-place: Amsterdam, Netherlands.
- [72] Vijayshankar Raman and Garret Swart. 2006. 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)*. VLDB Endowment, 858–869. Place: Seoul, Korea.
- [73] Red Hat, Inc. 2017. CVE-2017-15098. <https://nvd.nist.gov/vuln/detail/CVE-2017-15098>
- [74] Karla Saur, Tara Mirmira, Konstantinos Karanasos, and Jesús Camacho-Rodríguez. 2022. Containerized execution of UDFs: an experimental evaluation. *Proc. VLDB Endow.* 15, 11 (July 2022), 3158–3171. <https://doi.org/10.14778/3551793.3551860> Publisher: VLDB Endowment.
- [75] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3731–3744. <https://doi.org/10.14778/3685800.3685802> Publisher: VLDB Endowment.
- [76] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. <https://www.usenix.org/conference/atc20/presentation/shillaker>
- [77] Moritz-Felipe Sichert. 2024. *Efficient and Safe Integration of User-Defined Operators into Modern Database Systems*. PhD Thesis. Technische Universität München. <https://mediatum.ub.tum.de/1713746>
- [78] Snowflake. 2025. Introduction to External Tables in Snowflake. <https://docs.snowflake.com/en/user-guide/tables-external-intro>
- [79] Snowflake, Inc. 2025. Introduction to external functions | Snowflake Documentation. <https://docs.snowflake.com/en/sql-reference/external-functions-introduction>
- [80] Benedikt Spies and Markus Mock. 2021. An Evaluation of WebAssembly in Non-Web Environments. In *2021 XLVII Latin American Computing Conference (CLEI)*. 1–10. <https://doi.org/10.1109/CLEI53233.2021.9640153>
- [81] Spiral. 2024. vortex. <https://github.com/spiraldb/vortex>
- [82] Mihail Stoian, Alexander van Renen, Jan Kobiolka, Ping-Lin Kuo, Josif Grabocka, and Andreas Kipf. 2024. Lightweight Correlation-Aware Table Compression. *ArXiv abs/2410.14066* (2024). <https://api.semanticscholar.org/CorpusID:273482434>
- [83] substraat-io. 2021. Substraat: Cross-Language Serialization for Relational Algebra. <https://github.com/substraat-io/substraat>
- [84] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 689–703. <https://doi.org/10.1145/3627703.3629562> event-place: Athens, Greece.
- [85] The Apache Software Foundation. 2013. Apache Spark. <https://spark.apache.org/>
- [86] The Apache Software Foundation. 2016. The ORC File Format. <https://orc.apache.org/>
- [87] The Apache Software Foundation. 2025. The Apache Arrow Flight Format. <https://arrow.apache.org/docs/format/Flight.html>
- [88] The Apache Software Foundation. 2025. Parquet File Size in Apache Impala. https://impala.apache.org/docs/build/html/topics/impala_parquet_file_

- size.html
- [89] The PostgreSQL Global Development Group. 1996-2025. The PostgreSQL Open Source Relational Database. <https://www.postgresql.org/>
 - [90] TileDB, Inc. 2024. TileDB. <https://docs.tiledb.com/main/background/architecture>
 - [91] Emil Tsalapatis, Ryan Hancock, Rakeeb Hossain, and Ali José Mashtizadeh. 2024. MemSnap μ 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)*. Association for Computing Machinery, New York, NY, USA, 622–638. <https://doi.org/10.1145/3620666.3651334> event-place: <conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>.
 - [92] Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, and Fraser Brown. 2024. 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)*. Association for Computing Machinery, New York, NY, USA, 231–248. <https://doi.org/10.1145/3617232.3624862> event-place: <conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>.
 - [93] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacifico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1 (Feb. 2020). <https://doi.org/10.1145/3371038> Place: New York, NY, USA Publisher: Association for Computing Machinery.
 - [94] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *Computer Aided Verification, Constantin Enea and Akash Lal (Eds.)*. Springer Nature Switzerland, Cham, 226–251. https://doi.org/10.1007/978-3-031-37709-9_12
 - [95] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*. ACM, 1:1–1:6.
 - [96] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 53–65. <https://doi.org/10.1145/3167082> event-place: Los Angeles, CA, USA.
 - [97] Andrew Werner. 2023. [BUG] verifier escape with iteration helpers (bpf_loop, ...). <https://lore.kernel.org/bpf/CA+vRuzPChFNXmouzGG+wsy=6eMcf1mFG0F3g7rbg-sedGKW3w@mail.gmail.com/> Publisher: Linux kernel mailing list.
 - [98] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference (IMC '21)*. Association for Computing Machinery, New York, NY, USA, 533–549. <https://doi.org/10.1145/3487552.3487827> event-place: Virtual Event.
 - [99] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*. USENIX Association, USA, 2. event-place: San Jose, CA.
 - [100] Matei A. Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research*. <https://api.semanticscholar.org/CorpusID:229576171>
 - [101] Zhan Zhang. 2015. SPARK-2883 Spark Support for ORCFile format. <https://issues.apache.org/jira/browse/SPARK-2883>
 - [102] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393. <https://www.usenix.org/conference/osdi22/presentation/zhong>
 - [103] Eduard Zingerman. 2024. [PATCH 6.6.y 00/17] bpf: backport of iterator and callback handling fixes. <https://lore.kernel.org/all/20240125001554.25287-1-eddyz87@gmail.com/> Publisher: Linux kernel mailing list.