Making encodings easier to adopt

Jialin Ding
Princeton University
jialind@princeton.edu

Modern enterprises collect, store, and analyze increasingly large volumes of data. This relentless explosion in data volumes has driven the need for more effective encoding algorithms for reducing data size. In particular, this paper looks at *lossless* encodings, which support perfect reconstruction of the original data via decoding. Lossless encodings are widely used in database systems because they are able to not only compress data sizes, thereby reducing the cost of storing data, but also to minimize the I/O overhead of reading data, since compressed data requires less I/O bandwidth to read from persistent storage. In modern cloud-native database systems, where data is stored durably on cloud object stores and a hot set of data is cached on a compute node's local storage, encodings also enable more data to be cached as part of the hot set.

The idea of encodings is not new, and many tried-and-true encodings have been used in database systems for decades, including run-length encoding, dictionary encoding, delta and frame-of-reference encoding, and bitmap indexes. However, recent years have seen a rapid succession of newly-proposed encoding algorithms, driven by several trends. First, the rise of neural networks and LLMs has made floating-point numbers increasingly important due to their use in model weights and embeddings, and many new encoding algorithms specifically target floating-point numbers. Second, the increased use of specialized data types for certain domain-specific applications, such as the ROOT storage format for particle physics data, motivates the development of specialized encoding algorithms. Third, modern applications care not only about minimizing the size of encoded data, but also about the latency and throughput of encoding and decoding data, and different encoding algorithms make different tradeoffs between these competing objectives.

However, despite the plethora of newly-proposed encodings, very few have been adopted into widely-used commercial database systems. Furthermore, encoding algorithms which make it into one database system might not make it into others, with the result that each database system supports a slightly different subset of encoding algorithms. The reason for this haphazard adoption is simple: encoding algorithms are complex, and adoption requires extensive engineering effort for implementation and testing of correctness, performance, and security. For the developers of many database systems, the effort of adopting new encoding algorithms is simply too high. The authors call this the "$M \times N$ problem" – supporting M encoding algorithms in N database systems requires $M \times N$ effort.

The authors of this paper propose (and implement) a solution to the adoption problem. They present Anyblox, a layer of abstraction which bridges encoding algorithms and database systems by enabling "self-decoding" datasets. Their core idea is that the logic for decoding a piece of encoded data is stored with the encoded data itself. In particular, the decoding logic is implemented as lightweight WebAssembly bytecode, which can be compiled and executed on most popular architectures. This way, database systems no longer need to implement their own decoding logic for each unique encoding. Instead, a database system only needs to be able to interpret and execute the decoding logic shipped with the encoded dataset. Anyblox transforms the $M \times N$ problem into a much more tractable $M+N$ problem, where each of the M encoding algorithms implements decoding logic once and each of the N database systems implements support for Anyblox once. This is similar to how LLVM works for bridging programming languages and CPU architectures, which the authors draw direct inspiration from, and also how ODBC/JDBC works for bridging applications and database systems.

Anyblox's "implement once, use anywhere" philosophy improves the *portability* of encodings across different database systems, but it naturally raises concerns about *performance* (since certain performance optimizations are only possible with tight integration between the host database system and encoding) and *security* (since it may be difficult to guarantee that the decoding logic won't corrupt system state). The authors address both concerns: WebAssembly uses sandboxing to provably ensure that decoding logic is isolated from the host database system, thereby ensuring security. Furthermore, WebAssembly bytecode is known to achieve near-native speeds on many hardware platforms, which limits the hit on performance. (Interestingly, these natural advantages of WebAssembly have been identified by related efforts as well [1].) The authors also make clever use of memory mapping for zero-copy decoding and thread-local caching, which makes decoding even more efficient. The authors demonstrate that Anyblox can be integrated into popular database systems like DuckDB with minimal engineering effort, and the resulting integration achieves minimal performance degradation compared to native encodings.

Anyblox is a significant contribution in that it addresses a major issue which is often understudied by the research community: not how to achieve better performance or lower storage cost, but rather how to make adoption of new techniques as easy and frictionless as possible for existing systems.

The authors are open about Anyblox's current limitations, especially in terms of performance. For example, database systems often get a performance boost by evaluating predicates directly on encoded data, but Anyblox is built to fully decode data into its uncompressed form and therefore cannot take advantage of this performance boost. The challenge of continuing to close the performance gap while maintaining the portability advantages of Anyblox will be an exciting direction for future research.

1. REFERENCES

[1] X. Zeng, R. Meng, M. Prammer, W. McKinney, J. M. Patel, A. Pavlo, and H. Zhang. F3: The open-source data file format for the future. *Proc. ACM Manag. Data*, 3(4), Sept. 2025.