Accelerating String-Heavy Queries with LLM Token Tables

Tobias Schmidt
Technische Universität München
tobias.schmidt@in.tum.de

Nicolas Schmitt
Technische Universität München
nicolas.schmitt@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Andreas Kipf
University of Technology Nuremberg
andreas.kipf@utn.de

ABSTRACT

Strings are the most common data type in modern database systems, yet they are often treated as an afterthought in high-performance data formats. While numerical data benefits from specialized, lightweight compression schemes, text is typically handled by general-purpose algorithms such as Zstd, LZ4, or Snappy, which require full-block decompression before processing. In this paper, we explore the potential of repurposing Large Language Model (LLM) tokenizers as a lightweight string compression scheme for databases, similar to FSST, but with a global token table shared across all tables and columns. Operators such as joins and aggregations can exploit this consistent encoding to defer decompression and process encoded values directly.

We implement a global token table based on GPT-4's tokenizer in Umbra and demonstrate execution time improvements of up to 2× on string-heavy workloads, while reducing storage and memory consumption by up to 1.65×. Tokenizers integrate well with other compression algorithms, such as FSST, OnPair, or Zstd, while maintaining good compression ratios and high decompression throughput exceeding 6 GB/s on a single CPU core.

PVLDB Reference Format:
Tobias Schmidt, Nicolas Schmitt, Thomas Neumann, and Andreas Kipf. Accelerating String-Heavy Queries with LLM Token Tables. PVLDB, 19(11): 3635 - 3648, 2026.
doi:10.14778/3836663.3836714

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/umbra-db/token-vldb2026.

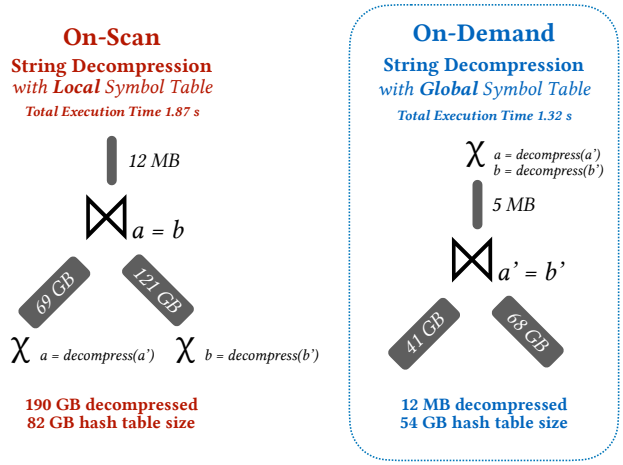


Figure 1: Text compression using a global dictionary allows for deferring decompression (on-demand). By performing the join on the compressed representations of the `text` and `body` string columns from the StackOverflow dataset, Umbra achieves a speedup of 1.4× while requiring significantly less memory during query processing.

1 INTRODUCTION

Strings are Everywhere. Snowflake, Redshift, and Tableau unanimously report `text/varchar` as the most common logical data type in their systems [67, 69, 70]. Not only is most data stored as strings, but almost half of the columns that are used for joins and aggregations in production workloads are strings [67]. Yet a long line of work on data formats and compression for database systems has focused on the representation of numeric values and their efficient decoding [10, 12, 20, 43, 44, 47, 77]. Strings have been treated as an afterthought in data formats for high-performance systems; most rely on general-purpose algorithms such as Snappy, LZ4, or Zstandard (Zstd). However, these off-the-shelf solutions operate as opaque blocks, preventing direct query execution on compressed data and requiring costly decompression cycles.

Columnar storage formats, such as Apache Parquet, partition tables into blocks of a few megabytes and compress each block using light- or heavyweight schemes [9, 13, 77]. For text data, most formats employ either *block-level* general-purpose compression algorithms, which require full-block decompression before access, or *field-level* schemes such as FSST [17] or dictionary encoding [72]. Field-level encodings compress data by storing strings in a separate table and substituting them with fixed-size codes. In this work, we refer to such a table as a *dictionary* if it exhaustively stores all full strings from a corpus, as in dictionary encoding. We refer to it as a *symbol table* if it stores frequently occurring substrings from the corpus, as in FSST. Both field-level algorithms are typically applied to individual blocks to maximize compression ratios by exploiting data locality. As a result, encodings vary across blocks, and strings cannot be compared or joined without first being decoded. In contrast, a global string encoding shared across all tables, columns, and blocks enables query execution on compressed data and delayed decoding. By maintaining a consistent, compressed representation

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836714

throughout the database, query engines can compare values directly without decoding and defer decompression.

Consider, for instance, the following query on the StackOverflow dataset [62] that finds matching posts and history entries based on a string join key:

```
select text from posts, posthistory
where text = body                  -- string join keys
  and text <> '' and text <> '""' -- filter empty strings
```

If the body and `text` columns share a common encoding, the join operator can process encoded values directly. As shown in Figure 1, compressed execution minimizes memory overhead and accelerates processing by deferring decompression until after the join.

In this paper, we tackle the problem of finding a *good, global* string encoding that enables *efficient* compression and decompression. We propose a single symbol table shared by all columns and tables in the database for text encoding, thereby accelerating query execution. Usually, constructing such a global table requires access to sample data and extensive training, but luckily, others have already done the heavy lifting for us. Large Language Models (LLMs) use tokenizers to map ASCII/UTF-8 text to a compact numerical representation, similar to the symbol tables used by FSST and OnPair [31]. Even for the latest proprietary models, such as OpenAI's GPT-5 series, the tokenizers are publicly available, allowing us to extract a symbol or *token table* for text compression. Since vanilla token tables are optimized for natural-language text, we adapt them for databases by extending them with common combinations observed in real-world workloads and limiting their size to support high-performance encoding and decoding.
The key features of our approach are:

- **Field-level access.** Like FSST, token tables support fast random access to individual values.
- **High throughput.** Up to 6 GB/s decoding and 300 MB/s encoding speed on a single core.
- **Good compression.** About 1.4× for arbitrary text and more than 2.0× for English text.
- **Second-level compression.** Integrates with local compression schemes to further reduce storage size.
- **Minimal memory footprint.** Less than 1 MB for a 65,536-entry token table.
- **Compressed execution in databases.** Reduces memory consumption and execution times on string-heavy queries.

The rest of this paper is organized as follows: First, Section 2 reviews related work. Next, Section 3 adapts LLM token tables for database string column compression and presents highly optimized encoding and decoding implementations. Section 4 compares token tables to possible global string encoding approaches in databases, detailing their trade-offs. In Section 5, we combine tokenizers with block-local compression and propose an optimized implementation of OnPair [31]. Section 6 discusses the integration of tokenizers into databases and how query execution performance can be improved. Following that, Section 7 evaluates our approach against standard storage formats across both standalone text compression and string-heavy query processing workloads. Finally, Section 8 summarizes our findings.

2 RELATED WORK

Compression in Databases. A long line of work has focused on integrating compression into database management systems [7, 34, 60]. In particular, the development of high-performance columnar analytical systems, such as C-Store [66] and MonetDB [18], demonstrated that compression is more than a storage optimization; it is fundamental to efficient query processing. The gains are threefold: (1) reduced storage costs, (2) more efficient utilization of system resources (I/O bandwidth, memory, and caches), and (3) faster query processing directly on compressed data.

Modern data formats, such as DataBlocks [44], BtrBlocks [43], FastLanes [11], Vortex [65], Lance [54], and others rely on lightweight compression schemes including frame-of-reference, run-length encoding, bit-packing, dictionary encoding, delta encoding, floating-point compression [12, 47], and the Fast Static Symbol Table (FSST) [17]. Lightweight schemes achieve high compression ratios, especially on numerical values, while allowing for fast decompression. Recent advancements demonstrate that recursively applying multiple encodings and exploiting inter-column correlations can further enhance compression efficiency [11, 43, 49].

Apache Parquet [27] and Apache ORC [26] are widely adopted open storage formats that support various compression schemes [48, 73]. Their strong compression performance and open specifications have made them the de facto standard for data lakes, such as Delta Lake [14] and Apache Iceberg [28]. They combine lightweight with heavyweight and general-purpose compression algorithms, including LZ4 [1], Snappy [33], and Zstd [25]. For strings, Parquet and ORC use either local dictionaries or a plain (uncompressed) representation, depending on string length and the number of distinct values [48]. However, both lack specialized string-aware encoders and must resort to byte-oriented general-purpose compressors; because these compressors treat strings as opaque byte sequences, they are unaware of string boundaries and sacrifice query processing speed for storage density.

String Compression. Boncz et al. propose FSST, a fast and space-efficient string compression scheme that encodes strings using a fixed-size symbol table, as an alternative to general-purpose compressors for strings in columnar data formats [17]. OnPair extends the FSST concept by using Byte-Pair Encoding (BPE) [30] to build larger symbol tables. It iteratively replaces the most frequent pairs of symbols in the data with new symbols until all slots in the symbol table are filled [31]. Re-Pair follows a similar approach to BPE but does not limit the symbol table size and instead builds a hierarchical dictionary of rules that can be applied recursively to compress strings [45]. Re-Pair in combination with front coding has been considered for dictionary compression in SAP HANA [46, 52]. Unlike symbol-table approaches, white-box compression [32] exploits the string structure. It extracts correlations, repeating substrings, and numeric values into separate physical columns and compresses the parts independently.

Compressed Execution. Leveraging data compression during query execution has been proposed early on for database systems [34, 38, 60], focusing primarily on lightweight compression schemes that are fast and easy to decode [72, 77]. HyPer [41], Umbra [53], and LiquidCache [36] all support filter evaluation directly on encoded data and selective decoding, speeding up execution by

reducing decoding overhead and data movement [44]. While HyPer decodes data during table scans, DuckDB supports lightweight compression (dictionary, constant, and sequence encoding) on vectors that are pushed through its execution pipelines [23].

For text data, recent systems have investigated compressed execution on dictionary-encoded strings. The BLU accelerator for IBM DB2 supports global codes that allow compressed execution on dictionary ids in joins and aggregations. However, global codes are not shared between tables, and before joining values from one table with another, the codes of the smaller table must be mapped to the dictionary of the larger table [59]. Similarly, Google PowerDrill [35], SAP HANA [64], and MorphStore [21] build exhaustive, order-preserving dictionaries [15] for each column. While per-column dictionaries support comparisons of encoded strings from the same column, comparing values across columns is not possible because the dictionaries differ. CodecDB implements encoding-aware query operators that exploit dictionary encoding to improve query efficiency [40]. Shared dictionaries have not seen widespread adoption, as they are memory-intensive [46], difficult to maintain beyond main memory, and do not fit into the decoupled, immutable architecture of data lakes. To our knowledge, leveraging a global symbol table for compressed execution has not yet been proposed.

3 TOKENIZERS

Large Language Models (LLMs) have gained tremendous popularity in recent years, largely due to their ability to "understand" and generate human-like text. Internally, these models operate not on ASCII/Unicode characters but on a numerical representation, so-called tokens. A token can represent a word, a subword, or even a single character. Intuitively, common words are often represented as single tokens, while rare character sequences are broken down into smaller subword or character-level tokens [63]. A crucial component in both training and inference of LLMs is the tokenizer, which converts input text into a sequence of tokens before feeding it to the model [58].

The tokenizers differ between models, as different strategies and training data are used. **Byte-Pair Encoding (BPE)** is a popular and fast byte-level tokenization strategy used by several state-of-the-art LLMs [51, 71, 76], including OpenAI's GPT models, the open-source Llama models, xAI's Grok models, and many others [19, 74]. It is based on the general-purpose data compression algorithm proposed by Gage, which iteratively merges the most frequent byte pairs in the training data to create a fixed-size vocabulary [30]. While BPE has not seen wide adoption for data compression, the algorithm is essential for language models [51, 57].

While variants of Byte-Pair Encoding (BPE) are already used by databases for encoding strings [17, 31], we adapt LLM tokenizers for global text compression, enabling the system to keep data in a compressed format while processing it. The idea is simple: LLMs are trained on multi-terabyte corpora of real-world text data from the internet, and therefore, the tokenizers created during training are well-suited to compress variable-length text data found in databases. In this chapter, we will first explore different tokenizers employed in today's models and analyze their compression performance. Following this, we explore practical strategies for adapting tokenizers for text compression and high-performance data processing.

```
u128* token_table; // Byte sequences for each token
u8* token_lengths; // Length of each token in bytes
u64 decode(u16* in, u8* out, u64 num_tokens) {
  u16* reader = in; u8* writer = out;
  for (;reader < in + num_tokens; reader++) {
    u8 len = token_lengths[*reader];
    memcpy(writer, token_table + *reader, len);
    writer += len;
  }
  return writer - out; // Number of bytes written
}

unordered_map<u128, u16> token_map; // Bytes to token id
u64 encode(u8* in, u16* out, u64 length) {
  u8* reader = in; u16* writer = out;
  while (reader < in + length) {
    u128 text = *((u128*) reader); // Read 16 bytes
    u128 mask = ~((u128) 0); // All bits set to 1
    for (u32 length = 16; length <= 16; --length) {
      u128 prefix = text & mask; // Extract prefix
      if (token_map.contains(prefix)) {
        *writer++ = token_map[prefix]; // Write token id
        reader += length; // Advance reader
        break;
      }
      mask >>= 8; // Check shorter prefix
    }
  }
}
```

Figure 2: Encoding and decoding text using a token table. The `encode` function greedily matches the longest possible token from the vocabulary. The `decode` function maps each token id back to its corresponding byte sequence. The code operates on 16-bit token ids.

3.1 Overview of Open-Source Tokenizers

While many LLMs are proprietary, some tokenizers, or at least the vocabularies, are publicly available. The token vocabulary is a fixed mapping from token ids to the corresponding byte sequences and can be used for encoding and decoding text. Figure 2 shows an unoptimized implementation for both operations. The simple greedy encoding algorithm iteratively matches the longest possible token from the vocabulary at the current position in the input text and writes the token id to the output sequence. Decoding is straightforward: each token id directly maps to a byte sequence in the token table. We copy the bytes to the output buffer and advance the write pointer by the token's length.

We extracted several token vocabularies from publicly available open-source and proprietary models. Table 1 lists the tokenizers, their size, and the LLMs that use them. The tokenizers differ in size: the smallest, Llama2, only contains 32 K tokens, while more recent models have vocabularies of up to 256 K tokens. These tokenizers are trained on larger datasets across different languages and domains, resulting in larger vocabularies. In particular, Unicode characters from non-Latin scripts, such as Cyrillic, Arabic, or Chinese, increase the number of tokens. Larger vocabularies represent them as single tokens, whereas smaller vocabularies break them down into individual bytes.

We evaluate the compression performance of the tokenizers on four datasets: **(a) `dbtext`**, a collection of string columns used

Table 1: **Available tokenizers, including their respective models, vocabulary sizes, and average token lengths.**

Tokenizer	Models	#Tokens	avg(length)
gpt3	OpenAI GPT-3	50 K	6.37 B
gpt4	OpenAI GPT-3.5/-4	100 K	6.30 B
gpt5	OpenAI GPT-4o/-4.1/-5	200 K	6.87 B
claude	Anthropic Claude 2/2.1	65 K	6.08 B
gemma2	Google Gemma2	256 K	6.54 B
gemma3	Google Gemma3	262 K	7.58 B
grok1	xAI Grok1	131 K	6.75 B
grok2	xAI Grok2	131 K	6.27 B
llama2	Meta Llama2	32 K	5.53 B
llama3	Meta Llama3/3.1	128 K	6.39 B
deepseek2	DeepSeek-V2	100 K	6.66 B
deepseek3	DeepSeek-V3/-R1	129 K	6.56 B
qwen3	Alibaba Qwen3	152 K	6.33 B

to evaluate FSST [17], **(b) `bitext`**, more than 100 text columns extracted from the Public BI benchmark [70], **(c) `languages`**, 7 common languages from the Wikipedia dump [29], and **(d) `onpair`**, 4 datasets used to evaluate OnPair [31]. The `dbtext` dataset by Boncz et al. consists of diverse string columns, including machine-readable identifiers, human-readable names, real-world text, and domain-specific codes. The Public BI benchmark (`bitext`) consists of a large collection of real-world datasets from 2018 that were available on Tableau Public[1]. In total, it has 965 text columns, but we removed 857 columns because they have fewer than 10% unique values, and other compression techniques, such as a string dictionary, are more effective. The filtered dataset consists of 106 real-world text files from various domains and different sizes (between 11 KB and 2.5 GB). The `languages` dataset contains a collection of articles (roughly 1 GB per language) from the Arabic, Chinese, English, German, Japanese, Russian, and Spanish Wikipedia websites.

Figure 3 shows the compression ratio of the 13 evaluated tokenizers across four datasets, sorted by median performance. Since tokenizers differ in vocabulary size, we bit-pack token ids to the minimum required bit-width. For example, GPT-5 (200K tokens) requires 18 bits per id, whereas Llama3 (128K tokens) requires 17 bits. Meta's Llama3 and OpenAI's GPT-4 tokenizers achieve the highest compression. While on most files the tokenizers achieve compression ratios above 1.0×, i.e., the tokenized data is smaller than the original, for some files the size increases. Llama2, for instance, has only 32 K tokens and lacks character combinations for non-natural-language text, forcing it to fall back to single-byte tokens. Consequently, the compression ratio drops to 0.5, as each input character (8-bit) is mapped to a 15-bit token id. Larger vocabularies do not guarantee superior performance: Google's Gemma 3 and 2 have more than 256 K tokens, yet their compression ratios are worse than those of smaller tokenizers. Our evaluation dataset consists primarily of real-world text columns from Tableau Public and `dbtext`, including random text patterns such as identifiers, email addresses, dates, and passwords. LLMs, on the other hand, are optimized for natural language processing through extensive training on articles, books, and web pages that contain repeating language patterns.

[1] https://public.tableau.com

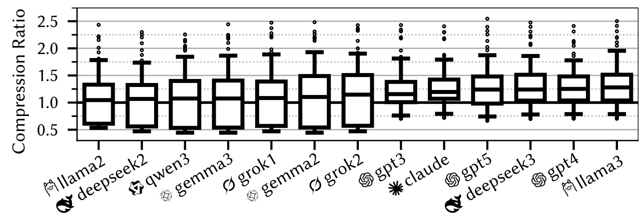


Figure 3: Compression ratio for different tokenizers. The tokenizers are sorted by their median compression ratio on the four datasets `dbtext`, `bitext`, `languages`, and `onpair`.

3.2 Adapting Tokenizers for Data Compression

While LLM tokenizers are promising for compressing natural language text, two major limitations complicate their use in general-purpose database systems: **(a) Non-natural language text:** The text datatype is often used to store a wide variety of data, including machine-generated identifiers. For data with more random patterns, tokenization can increase the size rather than reduce it (cf. Figure 3). Tokenizers, therefore, need to be adapted to better capture these patterns and avoid inefficient encodings. **(b) Impractical token id sizes:** Tokenizers with more than 65,536 tokens require more than two bytes for each token id. This hurts either the compression ratio, by padding token ids to 32-bit integers, or the compression speed, as packing and unpacking to non-byte-aligned sizes is expensive. Ideally, token ids fit within 16 bits, enabling efficient encoding and decoding with simple, highly optimized routines [31].

To address these limitations, we patch the tokenizer vocabularies by removing long tokens and limiting the entry count to 65,536. Note that tokens are eliminated by length (longest first) and not by frequency, as LLM tokenizers do not expose the per-token frequencies. Additionally, we add missing byte sequences that are more common in database text, such as numbers, dates, and email addresses. Patching tokenizers involves the following five steps:

- **(1) Single-byte tokens:** We first ensure that all 256 possible byte values are included in the vocabulary, allowing for a fallback encoding strategy for any input text, even binary data.
- **(2) Short tokens:** Next, we add sequences of up to 4 bytes from the original tokenizer, as these are more likely to appear.
- **(3) Two-byte ASCII tokens:** We add all missing combinations of alphanumeric ASCII characters (a-z, A-Z, 0-9) of length two to capture random text patterns such as identifiers, email addresses, or passwords. In addition, we also combine the alphanumeric characters with the four special characters `+`, `-`, `/`, `' '` (space), and `'\n'` (newline) to capture common patterns and base64 encoded binary data.
- **(4) Three-byte digit tokens:** As text columns in databases often contain numeric data, we append all three-digit combinations (000-999) to encode numbers more efficiently [70].
- **(5) Long tokens:** Finally, after patching the new tokens, we fill the remaining slots with tokens from the original tokenizers, starting with the shortest ones until we reach 65,536 tokens. We prefer tokens consisting only of printable ASCII characters to prioritize Latin script, particularly English, which is the most widely used language on the web [6].

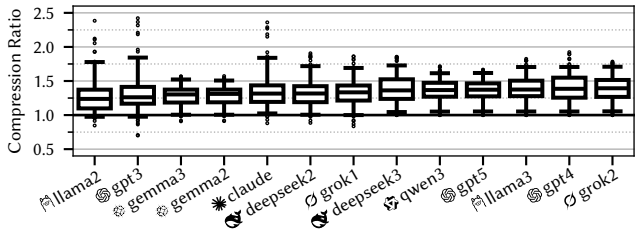


Figure 4: Compression ratios of the patched tokenizers on the four datasets. We sort the tokenizers by their median compression ratio across all datasets. grok2 and gpt4 achieve the best compression after patching their vocabularies.

Figure 4 shows the compression ratios of the patched tokenizers on the four datasets. Now Grok2 and GPT-4 encode the datasets best: most files compress by a factor of 1.25×to 1.5×, and the median compression ratio is close to 1.4×. Most importantly, after patching, the compression ratio for the best six tokenizers is always above or at least 1.0×, meaning that the data size never increases.

While patching stabilizes compression performance across more diverse datasets, it worsens performance on natural language text, as we prioritize shorter tokens over longer ones. Table 2 lists the number of tokens by their length in bytes for the four best-performing tokenizers after patching. For Grok2, the vocabulary includes all tokens up to 5 bytes, supplemented by 4.9 K 6-byte tokens. In addition, patching adds 2,335 two-byte ASCII tokens and 1,000 three-byte digit tokens that were missing from the original tokenizer. GPT-4, in contrast, required only 2,744 synthetic two-byte tokens since its native vocabulary already includes all three-digit permutations. This leaves more space for longer tokens, with 13.1 K 6-byte tokens and 3.8 K 7-byte tokens, which are particularly effective for natural language text. GPT-4 compresses English Wikipedia articles by 1.82×, whereas Grok2 achieves only a ratio of 1.70×.

In summary, patching adapts LLM tokenizers for database text compression, preventing size increases across the evaluated datasets while ensuring robust encoding for all files. However, incorporating additional short tokens may be necessary to better capture real-world patterns and further optimize compression performance. For the remainder of this paper, we will use the patched version of GPT-4 because it slightly outperforms Grok2 on English text, which we anticipate is the predominant language in real-world applications.

3.3 Efficient Encoding and Decoding

Achieving fast encoding and decoding is crucial for the adoption of tokenizers in data processing applications. In the following, we discuss practical implementation choices for high-performance decoding and encoding. We exploit the patched vocabularies to eliminate branches and expensive memory accesses.

Decoding. The simplicity of the decoding process is one of the main advantages of token tables for data compression. The algorithm shown in Figure 2 already achieves single-threaded decoding speeds of more than 5 GB/s. Loop unrolling further improves performance and enables high instruction-level parallelism, as the CPU can interleave multiple instructions and hide memory latencies. Since the patched tokenizers shown in Table 2 have tokens of up to 7 bytes, we optimize the `memcpy` call by copying 8 bytes rather than the actual length. On x86-64 CPUs, this translates to a single 64-bit `load-and-store` instruction instead of two for moving 16 bytes.

Table 2: Number of tokens after patching, grouped by their length in bytes, for the four best-performing tokenizers from Figure 4. All tokenizers contain exactly 65,536 tokens.

Tokenizer	Token Length (bytes)							
	1	2	3	4	5	6	7	8
gpt5	256	7.1 K	18.2 K	26.2 K	13.8 K	0	0	0
llama3	256	7.0 K	15.4 K	18.6 K	17.2 K	7.1 K	0	0
gpt4	256	6.6 K	11.9 K	15.1 K	14.8 K	13.1 K	3.8 K	0
grok2	256	7.3 K	16.4 K	18.5 K	18.1 K	4.9 K	0	0

We also experimented with SIMD instructions to decode in parallel, but on AMD Zen5 and Intel Granite Rapids, we observed a 20 % performance loss. The SIMD execution is dominated by random memory accesses: decoding 32 16-bit token ids with AVX512 requires 6 `gather` instructions, which provide no speedup over scalar loads. Furthermore, additional instructions are required to move 16-bit token ids into wider registers, which causes the slowdown.

Encoding. The encoding process is more complex; it requires finding the longest matching token. The naïve implementation from Figure 2 iterates through all prefixes of the input text and performs a hash table lookup to check if the prefix is part of the vocabulary. To improve the performance, we apply three optimizations:

(1) Optimized hash tables: We evaluate several hash table implementations, including a lossy design that eliminates some entries to simplify collision resolution.

(2) Short-token array: Fixed-sized lookup array to map 1- and 2-byte prefixes, taken from the FSST implementation [17].

(3) Long-token lists: For tokens longer than 3 bytes, we create lists of tokens with the same prefix (first 4 bytes) and perform a linear search, adapted from the OnPair [31].

The patched GPT-4 tokenizer has only tokens up to 7 bytes; consequently, we must test for 7 possible prefixes, starting with the longest one. Additionally, the prefix with a single byte will always match; otherwise, the encoding process fails. We therefore assign the single-byte tokens to the first 256 token ids and cast the byte value directly to 16 bits, giving the correct id and eliminating the need for a hash table lookup. Hence, only 6 hash table lookups are required for tokens of lengths 2-7 bytes.

Figure 5 shows the encoding speed for different hash table implementations. Besides GPT-4, we also consider Grok2 and GPT-5, which have token lengths of up to 6 and 5 bytes, respectively, and therefore require fewer hash table lookups. Boost's `unordered_map` implementation performs best of the off-the-shelf solutions and improves encoding speeds up to 4× over `std::unordered_map`. It also outperforms the open-addressing hash table with robin hood collision resolution [4] used by OnPair [31]. The boost hash table relies on bitmasks to speed up negative lookups, which dominate the encoding process, as most (long) prefixes do not match any token in the vocabulary [2].

The token table is static and known in advance, so we can further reduce lookup cost by using a custom perfect hash table, which

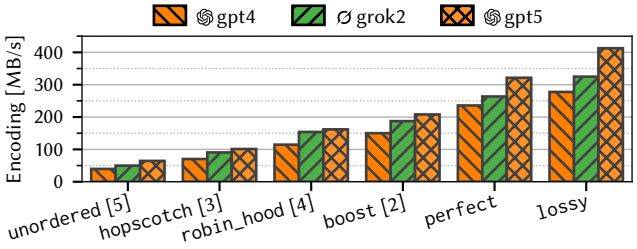


Figure 5: Encoding throughput for different hash table implementations. We report the median throughput over all four datasets. Our lossy hash table design performs best.

```
u64 magic = 0x2127'599b'f432'5c37ull;
u64 fasthash(u32 key) { // Hash 32-bit integer
    return ((u64) key) * magic; // imul
}
u64 fasthash(u64 key) { // Hash 64-bit integer
  u128 m = ((u128) key) * ((u128) magic); // mulx
  return (m >> 64) ^ ((u64) m); // xor
}

template <typename K, typename V>
struct lossy_map {
  pair<K, V>* buckets; // Key-value pairs
  u32 shift; // 64 - log2(size)
  V* find(K key) {
    u64 slot = fasthash(key) >> shift;
    if (buckets[slot].first == key)
      return &buckets[slot].second; // Found
    if (buckets[slot + 1].first == key)
      return &buckets[slot + 1].second; // Next slot
    return nullptr; // Not found
  }
}

template <typename K, typename V>
struct unchained_map {
  u32 *dir; // Directory with value ranges
  V* values; // Values
  u32 shift; // 64 - log2(size)
  pair<V*, V*> find(K key) {
    u64 slot = fasthash(key) >> shift;
    u32 begin = dir[slot], end = dir[slot + 1];
    // Range of entries that could match the key
    return pair(values + begin, values + end);
  }
}
```

Figure 6: Optimized hash table implementations for token lookup. The `lossy_map` performs only one random memory access per lookup at the cost of discarding a few entries. The `unchained_map` returns a range of entries that share the same prefix, which can be searched linearly.

guarantees constant-time, branchless lookups [55]. Each lookup requires two hash functions and two memory lookups, one to retrieve an additional offset and one to retrieve the element:

```
buckets[map(hash1(key)) + offsets[map(hash2(key))]]
```

The `map` call maps the hash value to the size of the hash table. Building the hash table might require multiple attempts due to collisions, but this is a one-time cost and can be done offline. The perfect hash table speeds up the encoding by another 1.5× compared to the boost implementation. Figure 6 shows the C++ code for lookup in the perfect hash table, as well as a simple but extremely fast hash function based on multiplication with a large odd constant, which can be efficiently implemented using the `imul` and `mulx` instructions on x86-64 CPUs [22].

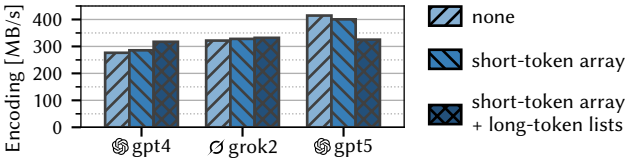


Figure 7: Encoding throughput with using the short-token array and long-token lists. While these optimizations improve performance for gpt4, they yield only a minor impact on grok2 and actually reduce throughput for gpt5.

Perfect hashing still requires two random memory accesses per lookup; we implement a lossy open-addressing hash table with linear probing that checks only the current and next elements. This scheme limits the number of random accesses per lookup by accessing only two adjacent elements, but fails when three or more entries collide. We keep only the first two entries and drop the rest; the encoder fails to match the dropped tokens and falls through to a shorter prefix for example in the short-token array. For the patched GPT-4 tokenizer less than 1 % of the tokens are discarded (629 of the 65,536). The lossy hash table improves throughput by another 1.2× compared to perfect hashing, while the compression ratio drops by only 0.006× on average. Compared to the naïve implementation using `std::unordered_map`, our hash table implementations improve encoding throughput from 40 MB/s to more than 277 MB/s for GPT-4.

Next, we apply the short symbol optimization from FSST [17]. It stores every possible one- and two-byte sequence in a 65,536-entry array to eliminate an additional hash table lookup. During construction of the array, 2-byte tokens are prioritized, and all remaining "free" entries in a row are filled with the same 1-byte code for the first byte. This layout allows the system to determine the best 1- or 2-byte match with a single, branchless memory access. Even though this optimization removes one hash table lookup, the overall performance only improves by 3 % (cf. Figure 7). For our cheap lossy hash table, the gains are limited; however, for less optimized hash tables, such as the perfect or boost implementations, the improvement is more significant, up to 10 %.

The long-token lists optimization is inspired by OnPair [31], which groups tokens with the same 8-byte prefix into lists and performs a linear search. We adapt this approach to tokens with 4-byte prefixes, since the patched tokenizers use shorter byte sequences. We use the custom `unchained_map` implementation from Figure 6, which avoids resolving collisions by inserting colliding tokens in the same list even if their prefixes do not match, demonstrating superior performance in cases with many duplicates [16]. When going through the list, matching the full token ensures that entries with different prefixes are eliminated. The lists are sorted by token length in descending order to find the longest match first.

```
u16 short_token_array[65536]; // 1- and 2-byte tokens
lossy_map<u32, u16> token_map3; // 3-byte tokens

// 4- to 8-byte tokens
struct LongToken {
  u64 text; // Full token text
  u64 mask; // Comparison mask
  u16 tokenId; // Token id
  u16 length; // Token length in bytes
};
unchained_map<u32, LongToken> long_token_map;

u64 encode_optimized(u8* in, u16* out, u64 length) {
  u8* reader = in; u16* writer = out;
  while (reader < in + length) {
    u64 text = *((u64*) reader); // Read 8 bytes
    u32 prefix4 = (u32) text; // Extract 4-byte prefix
    u32 prefix3 = prefix4 << 8; // Extract 3-byte prefix
    u16 prefix2 = (u16) text; // Extract 2-byte prefix

    // Check 4- to 8-byte tokens
    auto [begin, end] = long_token_map.find(prefix4);
    for (; begin != end; ++begin) {
      LongToken token = *begin;
      if ((text & token.mask) == token.text) {
        *writer++ = token.tokenId; // Write token id
        reader += token.length; // Advance reader
        goto next; // Continue with next token
      }
    }

    auto it = token_map3.find(prefix3);
    if (it) { // Check 3-byte tokens
      *writer++ = *it; // Write token id
      reader += 3; // Advance reader
    } else { // Find 1- or 2-byte token
      u16 tokenId = short_token_array[prefix2];
      *writer++ = tokenId; // Write token id
      reader += tokenId < 256 ? 1 : 2; // Advance reader
    }
next:
  }

  return writer - out; // Number of tokens
}
```

Figure 8: Optimized encoding function that incorporates the short-token array and long-token lists optimizations. The function first checks for long tokens using the 4-byte prefix, then for 3-byte tokens, and finally falls back to the short-token array for 1- and 2-byte tokens.

Figure 8 shows the fully optimized code for encoding text to tokens, including the short-token array and long-token lists optimizations. Together, these optimizations provide a median encoding throughput exceeding 320 MB/s and close the performance gap between GPT-4 and Grok2. However, GPT-5 does not benefit from these optimizations, and applying them reduces throughput from 400 MB/s to 320 MB/s as it contains shorter tokens than the other tokenizers. The choice of optimization depends on the tokenizer: the short-token array and long-token lists pay off if the vocabulary contains tokens of length ≥ 6 bytes (GPT-4, Grok2); for shorter-token vocabularies (GPT-5, ≤ 5 bytes), neither helps, and the long-token list optimization actively hurts and should be disabled.

4 GLOBAL STRING ENCODING IN DATABASES

There are several possible approaches to a global string encoding, which we compare conceptually and experimentally with LLM tokenizers in this chapter. Besides global symbol tables, which substitute common byte sequences with shorter codes, we consider global dictionaries, which assign a unique integer id to entire strings. We compare the approaches across the following four dimensions:

Compression Ratio: Efficiency of string compression.
Queryability: Suitability for compressed query execution.
Memory Footprint: Total memory consumption.
Maintainability: Long-term maintenance overhead.

A conceptual comparison of the five global string encoding implementations is shown in Table 3, with corresponding experimental results in Table 4. Each approach is discussed in detail below.

Exhaustive (Ex.) Dictionary. An exhaustive dictionary stores every unique string from the database, replacing variable-length data with compact integer ids. For queryability in Table 3, we assume that the dictionary is lexicographically ordered, enabling not just equality but also inequality comparisons and sorting on the keys. While this approach yields high compression ratios, the dictionary's memory footprint can become prohibitive, as it grows with the number of distinct strings in the database (cf. Table 4). Additionally, ordered dictionaries introduce significant maintenance overhead, as inserting new strings might change the ids of existing strings, triggering expensive rewrites across the entire database. In large-scale analytical systems, this massive centralized structure risks degrading performance by displacing data from memory, and makes maintenance prohibitively expensive.

Restricted (Re.) Dictionary. To mitigate the drawbacks of an exhaustive dictionary, we restrict the number of entries. The 2^{16} strings with the highest gain (frequency and length) are stored in the dictionary, while the rest are stored uncompressed as raw text. This hybrid strategy trades compression ratio for a reduced memory footprint. Nevertheless, in workloads with a few highly repetitive values (e.g., enumerations, string foreign keys), the benefits are similar to those of the exhaustive approach.

However, restricting the dictionary complicates query execution and maintenance: Predicate evaluation is no longer uniform. If both operands are dictionary keys, comparing integers is enough; otherwise, the engine must fall back to more expensive string comparisons. While string insertions incur no overhead, evolving data distributions present a significant challenge. The system must continuously monitor for data drift to determine when to update the dictionary. However, changing the dictionary is highly expensive because it alters existing data and triggers rewrites.

Table 3: Evaluation of different global string encodings.

Approach	Compression Ratio	Queryability	Memory Footprint	Maintainability
Ex. Dictionary	++	++	--	--
Re. Dictionary	o	+	o	-
P.-T. Symbol Table	-	+	+	++
F.-T. Symbol Table	+	+	+	-
LLM Tokenizer	o	+	+	++

Table 4: Compression ratio and memory consumption of global string encoding approaches. `PBI` is a 16 GB sample from Public BI datasets, which is disjunct from the workbooks used for pre-training the symbol table.

Approach	Compression Ratio			Memory Footprint (MB)		
	`PBI`	`dbtext`	`onpair`	`PBI`	`dbtext`	`onpair`
Ex. Dictionary	3.33	6.67	14.34	2356.64	43.25	1776.73
Re. Dictionary	2.05	1.23	1.00	1.98	7.58	4.07
P.-T. Symbol Table	1.81	1.25	1.00	1.06	1.06	1.06
F.-T. Symbol Table	2.82	2.32	1.58	1.06	1.06	1.06
LLM Tokenizer	1.36	1.38	1.13	1.06	1.06	1.06

Pre-Trained (P.-T.) Symbol Table. Global symbol tables shared across all database text columns provide an alternative to traditional dictionaries. We first evaluate a pre-trained symbol table built on an external, representative corpus before deploying it. Like LLM tokenizers, this approach requires zero training on a live database and thus significantly lowers integration complexity.

The difficulty of this approach is finding a representative corpus for training. We select a 16 GB sample from the Public BI dataset, which contains real-world text data from business intelligence workloads, and train a symbol table on it using the OnPair algorithm [31]. We aimed to incorporate strings of varying lengths and columns with varying cardinalities in our sample to mirror the distribution of the full corpus. Crucially, the sample preserves all semantic classes of strings from Public BI, including natural language, structured codes, URLs, digit sequences, dates, and categorical enumerations. Regarding compression ratio, the pre-trained symbol table performs well on PBI but poorly on `dbtext` and the `onpair` corpus. For `dbtext`, it performs worst on Chinese and Japanese (expected, as PBI contains no such text), and also poorly on character sequences that are unlikely to appear in regular English text, such as hex numbers, genome strings, and UUIDs.

In contrast to dictionaries, the symbol table's memory footprint is independent of the string count or length; capping its size at roughly 1 MB keeps overhead minimal. Furthermore, since the pre-trained symbol table is static, it requires zero maintenance. However, it degrades queryability in two ways: encoded strings remain variable-length (albeit shorter) and still require expensive string comparisons, and it supports only equality predicates.

Fine-Tuned (F.-T.) Symbol Table. Instead of pre-training a static symbol table, it can also be trained directly on the target workload. Our experiments show that fine-tuning yields high compression ratios, outperforming both pre-trained symbol tables and restricted dictionaries. In terms of queryability and memory consumption, this approach is identical to its pre-trained counterpart. However, workload-specific training introduces a maintenance burden comparable to that of a restricted dictionary. Since changing data distributions can degrade the compression ratio over time, the system must continuously adapt the symbol table. As before, updating the symbol table is expensive, as it rewrites all string columns.

LLM Tokenizers. LLM tokenizers are essentially pre-trained symbol tables on a much larger, more diverse corpus [42]. In our experiments, they deliver more consistent compression ratios than pre-trained symbol tables across different datasets.

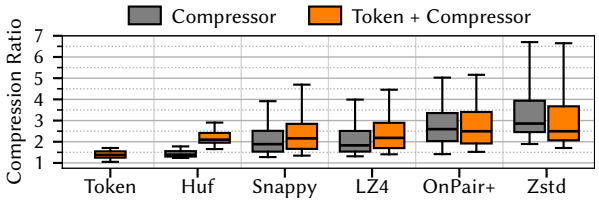


Figure 9: Compression ratios on the four datasets with and without tokenized input. The ratio is reported relative to uncompressed (UTF-8) size.

5 TWO-LEVEL COMPRESSION

Token tables compress data using the vocabulary of LLMs trained on vast corpora of natural text. This static, common encoding of strings allows comparisons without decoding, but sacrifices compression ratio by disregarding the local data distribution within individual datasets. In this chapter, we investigate how tokenizers can be combined with general-purpose compression algorithms, such as LZ4, Snappy, and Zstd, as well as Huffman Coding and OnPair.

Applying a second compression layer to tokenized text is *complementary* and does not forfeit the advantages of compressed execution. We propose a two-level approach to minimize storage footprint and bandwidth consumption upon retrieval; data can be decompressed back into its token representation upon loading into main memory or during table scans. Figure 9 compares the compression ratios on UTF-8 and token encoded text files. We used the four datasets from Section 3; each file is first tokenized, and then the payload is compressed further using existing algorithms (note that we always compress a full file and do not chunk it into blocks).

5.1 General-Purpose Algorithms

Huffman Coding, LZ4, Snappy, and Zstd are widely used compression algorithms. Huffman Coding [37] is a popular entropy encoder due to its simple implementation and fast speed [39]. It builds an optimal prefix code based on character frequencies. The other three compression algorithms are used by database systems to read Apache Parquet files in data lakes [48], load external files, or perform internal data compression. LZ4, Snappy, and Zstd rely on dictionary encoding using LZ77 by Ziv and Lempel [75]. Zstd additionally applies asymmetric numeral systems (ANS) on top of LZ77 to further reduce file size. ANS, like Huffman Coding, is an entropy coder but operates at the sub-bit level, allowing it to represent symbols with fractional bits and to approach the Shannon entropy limit, whereas Huffman is restricted to integer bit lengths [24].

On plain text files, Huffman achieves a factor of 1.39×, similar to the tokenizers. When combining the two encodings, the compression ratio improves significantly: Token+Huf compresses the files by 2.10×. Even though Snappy, LZ4, and the tokenizer approach all implement variants of dictionary encoding, combining them works surprisingly well: Token+Snappy and Token+LZ4 achieve compression ratios close to 2.20× in Figure 9, higher than for either algorithm alone. Only for Token+Zstd, we observe a drop in the compression ratio; this loss likely stems from Zstd's byte-oriented logic, which conflicts with the 16-bit structure of our token ids.

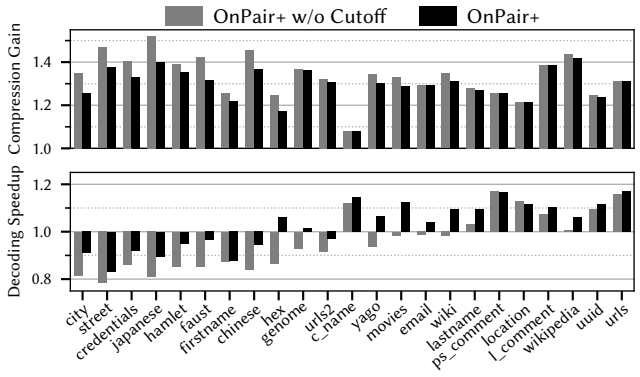


Figure 10: Compression ratio and decompression speedup of OnPair+ (w/o and with cutoff) relative to OnPair16 on the `dbtext` corpus. The datasets on the x-axis are sorted by file size, ranging from 130 KB to 6 MB.

5.2 OnPair

Gargiulo and Venturini recently proposed OnPair [31] as an alternative to FSST for field-level text compression in database systems. OnPair compresses data by substituting variable-length byte sequences with symbols. The encoded text consists of a sequence of ids referencing the entries in a symbol table. Compared to FSST, OnPair uses a dictionary with 65,536 symbols instead of 255, enabling it to capture more patterns at the cost of increasing the id size from 1 to 2 bytes. Compression proceeds in two steps: First, the symbol table is built using Byte-Pair Encoding by iteratively merging the most frequent adjacent symbols. The new symbols receive a fresh id and are appended to the symbol table until it is full or the input is exhausted. Next, byte sequences are replaced with symbol ids to compress the input. To improve decompression throughput, OnPair16 restricts symbol lengths to 16 bytes, bounding the symbol table size to just over 1 MB.

5.2.1 OnPair+. We introduce two optimizations to the original OnPair algorithm that improve compression ratios on small datasets while also increasing decompression throughput. For the remainder of this paper, we denote our optimized variant as OnPair+.

First, we redesign the dictionary's on-disk layout by exploiting a key property of OnPair: every new symbol is formed by merging two existing symbols and assigning a new symbol id to the result. Thus, it is enough to store, for each symbol id, the pair of two-byte symbol ids from which it was constructed. Hence, every symbol can be represented by just 4 bytes, regardless of its length. For small files (<1 MB), reconstructing the symbol table on-the-fly is more expensive than decoding the strings themselves. To minimize this overhead, we introduce a second optimization: an additional cutoff criterion that limits the number of symbols. Specifically, we terminate the symbol table construction once its on-disk size reaches one-sixteenth of the input's uncompressed size. This produces fewer symbols, making the cutoff a trade-off between compression ratio and decompression throughput. We evaluated several thresholds and found that one-sixteenth strikes the best balance.

Combining both optimizations improves the compression ratio by up to 40 % on the `dbtext` corpus, as shown in Figure 10. While smaller files benefit most because OnPair's symbol table takes up a large portion of the compressed size, rebuilding the table is expensive and reduces decompression throughput. Applying the cutoff yields additional gains: it improves throughput by 8 % on average, while losing only 3 % in compression ratio.

5.2.2 Integrating OnPair+ with Tokenizers. Applying OnPair+ and LLM tokenizers in sequence works well intuitively, as both identify and substitute patterns, preserving common patterns while only making them shorter. While OnPair typically starts by merging single-byte characters, for tokenized text we initialize the symbol table with all distinct token ids present in the input. This prevents the algorithm from splitting the 16-bit ids, allowing it to better identify patterns in tokenized text. Because OnPair+ evolves its dictionary by merging adjacent symbols, all newly created symbols naturally consist of token id sequences.

Besides maintaining a stable compression ratio on tokenized data (cf. Figure 9), not splitting token ids enables further optimizations during decompression: we can fuse OnPair+'s and the tokenizer's decoding kernels into a single kernel that decompresses both levels simultaneously. Essentially, we are decoding the token ids in OnPair's symbol table with the token table to directly map symbol ids to the ASCII/Unicode character sequences. Fusing the kernels eliminates the intermediate state between the two decoding steps, and the data is only read and written once, resulting in speedups up to 2× compared to applying the two kernels sequentially.

6 INTEGRATION WITH DATABASE SYSTEMS

The major advantage of using tokenizers for text compression in database systems is that data can be compressed across multiple blocks and columns using a single global vocabulary. Consider, for example, the following query that extracts shipping information on the well-known TPC-H schema:

```
select l_shipdate                          as shipdate,
       l_comment || o_comment              as comment,
       l_shipmode || ' ' || l_shipinstruct
                  || ' ' || o_orderpriority as instruct
from lineitem join orders on l_orderkey = o_orderkey
              join part   on l_partkey = p_partkey
where l_shipmode in ('AIR', 'REG AIR', 'MAIL')
  and l_comment || o_comment like '%regular%'
  and p_container similar to '(JUMBO|LG) (BAG|BOX)'
```

Figure 11 shows the corresponding query plan for this query. With tokenized text columns, decoding the strings can be postponed until an operation that requires the actual string values is performed, such as the `like` or `similar to`, or until the end of query execution, when returning the final result. Block-level compressors, like Zstd or Snappy, in contrast, must decompress the data during the table scan, as they do not allow access to individual values. Field-level compressors, like FSST or OnPair, are the middle ground, as they allow access to individual values, but data must be decoded within the current execution pipeline, as symbol tables are block-local. In the given example, field-level compressors must decode columns from `lineitem` before building the hash table for the join with `orders`, while tokenizing the data allows decoding after the join.

The query plan also shows the operations on the string columns. ① evaluates the `similar to` predicate: the tokenized `p_container`

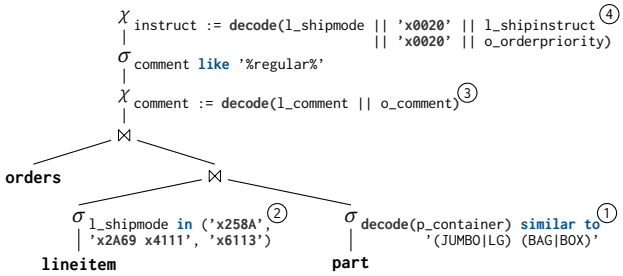
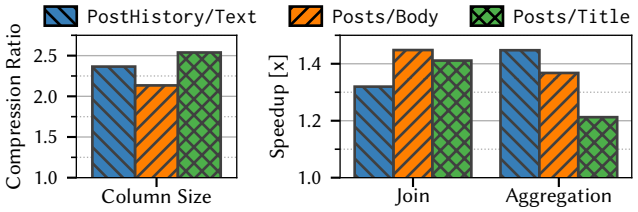


Figure 11: Query plan for the example query with tokenized text columns. We highlight the operations required to decode and filter the text data.



(a) Storage Improvements **(b) Query Execution Speedup**

Figure 12: Compression ratio and join/aggregation speedup on the StackOverflow dataset in Umbra. We report the improvements relative to the baseline w/o tokenization.

column must be decoded before applying the predicate. In ②, the values in the `in` statement are encoded as tokens, allowing the filter to be evaluated on the token ids without decoding the column. The literal `'REG␣AIR'`, for example, is represented by the tokens **`x2A69`** (`'REG'`) and **`x4111`** (`'␣AIR'`) in GPT-4. For the `like` predicate, the concatenated `comment` column is decoded in ③. However, note that both columns pass through the join operator before decoding them; this is not possible with block-level or field-level compression. Finally, in ④, we decode the `instruct` column after concatenating the text, just before returning the value to the user.

Trade-Offs. Tokenizers are not order-preserving, and operations such as range predicates, sorting, similarity joins, and `like` or regex patterns must first decode the data. While there is ongoing research on evaluating text-matching operations on compressed strings [56], for now only equality-based comparisons, `in` predicates, and hash computations can safely be performed on the token ids. Query engines can mitigate the cost of decoding tokenized strings by integrating the conversion into table scans or pushing it up the execution plan and performing it as late as possible. Postponing decoding allows for more efficient query execution, reducing the amount of data materialized between execution pipelines and processing fewer bytes for equality checks and hashing. In addition, fewer rows need to be decoded, as filters, selective joins, and aggregations reduce the amount of data to be processed in the later stages of the execution plan. We expect compressed execution on token strings to yield substantial benefits for real-world workloads. For instance, Snowflake reports that customer queries use text columns in over 50 % of all filter predicates and 46 % of join keys [67].

6.1 Query Execution on Tokenized Data

We integrated GPT-4's tokenizer into Umbra [53] to evaluate the impact of a global token table on query execution. When inserting data, ASCII/UTF-8 strings are converted to a sequence of token ids using the algorithm from Section 3.3 and stored on disk in this representation. Table scans return tokenized strings to the execution pipeline, allowing operators to evaluate predicates, joins, and aggregations directly on token ids.

Internally, Umbra treats token sequences as a new data type (`tokentext`) that behaves like the existing string types, and a simple cast to `text` triggers decoding. Umbra's type system ensures that non-equality operations convert tokenized text to regular strings before processing, while equality checks and hash computations operate on the token ids. To avoid decoding the same value multiple times, we modified Umbra's query optimizer to introduce `map` operators that convert the tokenized columns to strings just before the first operator that requires the actual text values accesses them. All subsequent operators then use the decoded strings.

Figure 12 shows the impact of tokenization on compression and query execution in Umbra on the real-world StackOverflow dataset. The dataset contains all questions, answers, and comments posted on the StackOverflow website until 2024. Unlike TPC-H and TPC-DS datasets, long text values dominate the dataset and consume more than 90 % of the total uncompressed storage size [62]. As the website's entries are programming-related and primarily written by an English-speaking community, the GPT-4 tokenizer achieves high compression ratios. The token representation shrinks the database size by 1.58× from 225 GB to 143 GB on disk.

For individual columns, the improvements are even greater: Figure 12a reports the compression ratio on the two largest text columns in the dataset, the `Text` column from the `PostHistory` table and the `Body` column from the `Posts` table, and `Posts/Title`. The columns differ in average string length: `Text` and `Body` contain long strings with 760 and 1145 Unicode characters, whereas the post titles are shorter, averaging 55 characters. For all three columns, the compression ratio is above 2×, with more than 2.5× on the short titles, which feature more repetitive words and phrases.

Query execution also benefits from the tokenized representation. In Figure 12b, we compare the runtime for finding duplicate values through a self-join or an aggregation that counts occurrences in the three columns, with and without tokenization. Umbra internally uses hash-based algorithms for both operations: for each string, it first computes a hash value for indexing into a hash table, then compares it against the entries to find matches. Both the comparison and the hash computation can be performed on the compressed token ids, resulting in speedups between 1.3× and 1.5×.

6.2 Storage Format

Following the storage design of modern analytical systems [8, 66] and formats such as Apache Parquet, Umbra splits tables into chunks of roughly 256,000 rows (Data Blocks) and compresses each column independently using lightweight encodings [61]. Once data is structured this way, systems must decide at which granularity to access and decode these blocks during query execution. Two

Table 5: Storage size of the (tokenized) text columns in the StackOverflow database using different compression algorithms. Umbra always applies field- and block-level compression on top of dictionary encoding.

		UTF-8		Token	
Uncompressed		213.4 GB	1.00×	129.4 GB	1.65×
field-lvl	Dict Encoding	200.7 GB	1.06×	118.0 GB	1.81×
	+ FSST	123.6 GB	1.73×	110.8 GB	1.93×
	+ OnPair+	87.4 GB	2.44×	87.1 GB	2.45×
block	+ LZ4	105.1 GB	2.03×	83.8 GB	2.55×
	+ Zstd	70.8 GB	3.01×	75.8 GB	2.82×

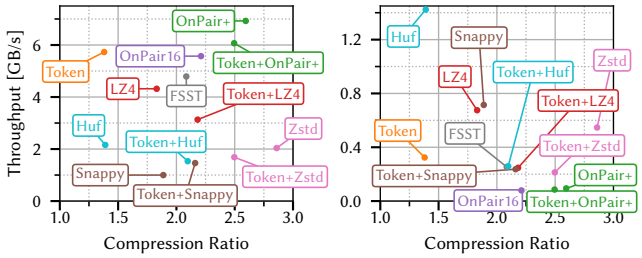
integrations are possible, depending on whether the compression supports field- or block-level access: (1) Decompression while loading the files into the database's internal buffer manager/file cache, or (2) Decoding on access, i.e., the table scan extracts individual strings. The second option requires a field-level encoding scheme such as FSST or OnPair, whereas the first option is possible for all compression algorithms. We implemented both variants in Umbra using the algorithms from Section 5.

Table 5 reports the size of text columns on the StackOverflow dataset. Umbra's table scans decode data on the fly, passing uncompressed strings through the execution pipeline. Specifically, the tokenizer reduces this uncompressed size by 1.65× from 213 GB to 129 GB. Field-level compression techniques reduce the space required to buffer text columns in memory, while block-level compression affects only on-disk storage and retrieval bandwidth consumption. Apart from FSST and OnPair, dictionary encoding is a simple and effective field-level encoder used by many formats.

Umbra always uses dictionary encoding on text columns within a data block, as it allows random access to variable-length strings, almost zero-overhead decoding, and efficient comparison with constants. The dataset contains many long, unique strings, so dictionary encoding yields little savings; however, combining it with tokenized text achieves compression ratios close to FSST. Tokenizers integrate seamlessly with dictionary encoding. OnPair+ achieves the smallest in-memory size across both UTF-8 and Token data for field-level compressors. As observed in Figure 9, the tokenizer does not further improve the compression ratio, and in both cases the text columns consume 87 GB. For block-level compression, Zstd performs best. However, while LZ4 benefits from the tokenizer, Zstd's compression ratio does not improve. Note that dictionary encoding introduces auxiliary integer values to store string lengths and ids, which are not compressed by the field-level algorithms.

6.3 Adaptability to Other Systems

Architecturally, Umbra differentiates itself from other state-of-the-art analytical systems through its compilation-based query execution model. Unlike vectorized systems, which pass small vectors through operators, Umbra compiles optimized machine code for each query and pushes one tuple after another through the compiled pipelines, fusing multiple operators together. Crucially, tokenizers do not interfere with compiled or vectorized execution because tokenized text uses the same representation as regular strings, making it opaque to physical operators. Our integration into Umbra only modified the type system and the storage format. For other systems, like DuckDB or ClickHouse, we expect similar changes.



(a) Decompression/Decoding **(b) Compression/Encoding**

Figure 13: Compression ratio versus (de-)compression throughput. We report results for compressors with and without tokenizing the input first.

7 EVALUATION

Setup. We evaluate the tokenizer and the different compression algorithms in two settings: (1) standalone single-threaded performance evaluation on `dbtext`, `bitext`, `languages`, and `onpair` (cf. Section 3); (2) end-to-end query processing benchmarks using TPC-H, TPC-DS, and Prod-DS at scale factor 100. Prod-DS [68] is a variant of TPC-DS by Szlang et al. that more closely mirrors the workload statistics from Snowflake [67]. In a string-heavy configuration (string level 15), the benchmark changes the join column types to text and extends the query templates to perform additional string operations. The code and all benchmark scripts for the microbenchmarks in the previous section and the end-to-end evaluation in this section are available online: https://github.com/umbra-db/token-vldb2026. The tokenizer and OnPair+ are also available in this repository. For the third-party compressors Snappy [33], LZ4 [1], Zstd [25], and OnPair [31], we use the official implementations. All experiments are performed on a local machine running an AMD Ryzen 9 7950X3D CPU (Zen4) equipped with 128 GB of RAM (DDR5, 3600 MT/s) and a 4 TB Samsung SSD 990 PRO (up to 7.5 GB/s read bandwidth).

Compression Performance. Figure 13a highlights a clear trade-off among standalone compressors. Lightweight token/symbol table-based algorithms (FSST, OnPair(+), and Token), which substitute variable-length byte sequences with fixed-size codes, achieve the highest decompression throughputs, surpassing 5 GB/s. The reason is simplicity: decoding requires only cheap lookups into small tables that fit in L2 caches of modern processors. However, to maintain this efficiency, these methods restrict symbol size and count, sacrificing compression ratio. Among them, OnPair+ establishes itself on the *Pareto frontier*. It achieves a decompression throughput of 7 GB/s, outperforming all competitors by over 20 %, while trailing only Zstd in compression ratio; for context, the single-threaded `memcpy` throughput is 15.7 GB/s. Conversely, Zstd improves compression ratio by 10 % but suffers a 70 % throughput penalty. While tokenization improves the compression ratio or throughput for some algorithms (Snappy, LZ4, Huffman), Zstd and OnPair+ suffer from the overhead of decoding token ids.

Table 6: Runtime comparison of the ten slowest Prod-DS queries, along with the median and average runtime across all 99 queries. Runtimes are given in seconds, and sizes in gigabytes. We report only text-column sizes, since numeric columns are not compressed. FSST, OnPair+ (OP+), LZ4, and Zstd are applied to dictionary-encoded data blocks.

(a) Hot run (all columns buffered in-memory)

		Q51	Q67	Q78	Q14	Q99	Q65	Q04	Q75	Q72	Q62	med	avg	size
Dict	UTF8	7.2	6.5	5.1	3.7	**3.5**	3.0	2.8	**1.9**	1.8	**1.7**	0.34	0.73	35.0
	Token	**4.0**	**6.2**	**3.9**	**2.7**	**3.5**	**1.5**	**2.4**	**1.9**	**1.2**	**1.7**	**0.31**	**0.60**	**28.6**
FSST	UTF8	7.5	6.5	5.2	4.1	**3.5**	3.0	2.8	**2.1**	2.0	**1.8**	**0.39**	0.80	23.7
	Token	**4.1**	**6.3**	**4.2**	**3.2**	**3.5**	**1.7**	**2.4**	2.2	**1.4**	**1.8**	0.40	**0.68**	**22.2**
OP+	UTF8	7.3	6.5	5.1	3.8	**3.5**	2.9	2.8	**2.0**	2.0	**1.8**	**0.36**	0.75	**22.2**
	Token	**4.1**	**6.3**	**4.1**	**3.1**	**3.5**	**1.6**	**2.4**	2.1	**1.5**	**1.8**	0.37	**0.67**	22.6

(b) Cold run (columns fetched from disk and decompressed on-the-fly)

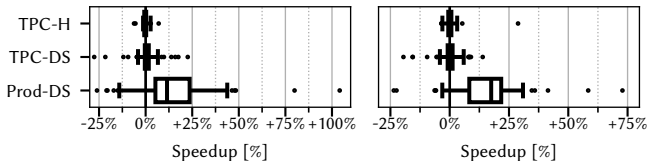
		Q51	Q78	Q67	Q14	Q65	Q04	Q88	Q75	Q64	Q72	med	avg	size
Dict	UTF8	8.6	7.6	7.5	6.8	5.0	4.6	4.4	4.1	4.0	3.5	1.28	1.62	35.0
	Token	**5.0**	**6.0**	**7.0**	**5.2**	**3.1**	**4.1**	**4.1**	**3.8**	**3.3**	**2.6**	**1.10**	**1.36**	**28.6**
LZ4	UTF8	8.0	6.6	7.0	5.3	4.0	4.2	2.6	3.4	2.9	2.6	0.87	1.25	17.0
	Token	**4.6**	**5.4**	**6.7**	**4.3**	**2.5**	**3.7**	**2.4**	**3.3**	**2.6**	**2.1**	**0.83**	**1.10**	**16.4**
Zstd	UTF8	7.9	6.3	6.8	4.9	3.7	4.1	2.7	3.1	2.5	2.4	0.74	1.15	**10.0**
	Token	**4.5**	**5.1**	**6.6**	**3.9**	**2.2**	**3.7**	**2.5**	**3.0**	**2.2**	**1.8**	**0.64**	**1.01**	10.1

For compressing data, LZ77-based algorithms outperform the symbol-table methods by more than 2× in Figure 13b. Nonetheless, our optimized tokenizer reaches 300 MB/s and outperforms even FSST's scalar implementation. Huffman Coding achieves the highest compression throughput, but yields poor compression ratios and low decompression speeds. Finally, while tokenization degrades Zstd and OnPair+ performance, it significantly improves compression ratios for LZ4 and Huffman, and boosts both throughput and ratio for Snappy via two-level decompression.

Query Performance. While for decompression speed and data size, the non-tokenized compressors are optimal, databases can leverage the tokenized representation to accelerate query execution. Figure 14 evaluates this trade-off by comparing Umbra's per-query execution times with and without tokenization. For synthetic workloads (TPC-H/DS), text columns are rarely accessed, so tokenization has no effect on either hot (in-memory) or cold (from disk) runs. On the string-heavy Prod-DS benchmark, tokenization yields up to 2× speedups, with 91 of 99 queries accelerating or achieving parity in hot runs. The remaining eight queries regress slightly due to the decoding overhead but finish in under 5 ms, whereas the most expensive queries (>1 s) show major gains. In cold runs, over half of the queries experience speedups exceeding 15 % as tokenization reduces execution and data retrieval times.

Table 6a reports the hot runtimes for the ten slowest queries, comparing dictionary encoding against FSST and OnPair+ on raw and tokenized strings. Because Prod-DS contains many low-cardinality columns, dictionary encoding alone compresses text columns from 268 GB to 35 GB, with up to 35× on individual columns. Tokenization provides substantial speedups, particularly for long-running queries. Consequently, the cumulative runtime across all 99 queries improves by over 20 %, dropping from 73 s on raw UTF-8 to 59 s on tokenized strings. For instance, Q46 and Q71 require heavy joins and aggregations on string keys; because the tokenized keys fit within the threshold for small string optimization (SSO), they drastically reduce the overhead of string hashing and comparisons [50, 53].

While field-level compression via FSST and OnPair+ minimizes memory consumption, it introduces a slight decoding overhead per access. Even though symbol tables achieve high decompression throughput, the added overhead slows query execution compared to standalone dictionary encoding. For cold (disk-based) runs, block-level compression is more effective, yielding higher compression ratios and minimizing disk I/O. Zstd achieves the best performance even though decompression is slower than LZ4. Furthermore, block-level techniques achieve an additional 2× space reduction.



(a) Hot run **(b) Cold run**

Figure 14: Umbra's performance using tokenized string columns relative to standard UTF-8 representations. Positive values indicate speedups for the token representation; negative values favor UTF-8.

Insights. OnPair+ and Zstd offer the strongest compression for general-purpose text; however, leveraging tokenizers accelerates query processing by reducing data volume and enabling compressed execution. Our evaluation shows that for in-memory (hot) workloads, lightweight dictionary encoding of tokenized strings significantly improves runtime, while for disk-based (cold) workloads, compressing blocks with Zstd minimizes I/O bottlenecks. Consequently, we propose a multi-tiered storage strategy: string columns are tokenized, and blocks use dictionary encoding for efficient query execution. For disk or remote storage, compressing blocks with Zstd provides the best overall performance.

8 CONCLUSION

This paper introduces a symbol-table-based encoding scheme for database text data by repurposing Large Language Model (LLM) tokenizers. We construct a database-wide, global token table that compresses text across multiple tables and columns. Integrated into the high-performance database system Umbra, our approach reduces the size of text data and speeds up analytical queries over string columns. While block-level and field-level algorithms require decompression before processing, our scheme supports compressed execution for core database operators, like joins and aggregations, deferring decoding until absolutely necessary.

We leverage the GPT-4 tokenizer, which was pre-trained on a multi-terabyte, real-world text corpus by OpenAI, providing us with a robust symbol table "for free". To better capture random and numerical patterns that are more common in databases than in natural language, we extend the original token table with additional ASCII character combinations. By adapting existing compression algorithms to operate directly on tokenized data, we achieve both strong compression ratios and high (de-)compression speeds.

REFERENCES

[1] [n. d.]. *LZ4*. Retrieved February 1, 2026 from https://github.com/lz4/lz4

[2] 2026. *Boost.Unordered*. Retrieved February 1, 2026 from https://www.boost.org/doc/libs/1_80_0/libs/unordered/doc/html/unordered.html

[3] 2026. *hopscotch-map*. Retrieved February 1, 2026 from https://github.com/Tessil/hopscotch-map

[4] 2026. *robin-hood-hashing*. Retrieved February 1, 2026 from https://github.com/martinus/robin-hood-hashing

[5] 2026. *std::unordered_map*. Retrieved February 1, 2026 from https://en.cppreference.com/w/cpp/container/unordered_map.html

[6] 2026. *Usage statistics of content languages for websites*. Retrieved February 1, 2026 from https://w3techs.com/technologies/overview/content_language

[7] Daniel J. Abadi. 2008. *Query execution in column-oriented database systems*. Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.

[8] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. 2009. Column oriented Database Systems. *Proc. VLDB Endow.* 2, 2 (2009), 1664–1665.

[9] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*. ACM, 671–682.

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

[11] Azim Afroozeh and Peter Boncz. 2025. The FastLanes File Format. *Proc. VLDB Endow.* 18, 11 (2025), 4629–4643.

[12] Azim Afroozeh, Leonardo Kuffó, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data* 1, 4 (2023), 230:1–230:26.

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

[14] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

[15] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD Conference*. ACM, 283–296.

[16] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *DaMoN*. ACM, 4:1–4:9.

[17] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661.

[18] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[20] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*. OpenProceedings.org, 72–83.

[21] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.* 13, 11 (2020), 2396–2410.

[22] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A Reliable Randomized Algorithm for the Closest-Pair Problem. *J. Algorithms* 25, 1 (1997), 19–51.

[23] DuckDB. [n. d.]. *Execution Format*. Retrieved February 1, 2026 from https://duckdb.org/docs/stable/internals/vector

[24] Jarek Duda. 2009. Asymmetric numeral systems. *CoRR* abs/0902.0271 (2009).

[25] Facebook. [n. d.]. *Zstandard*. Retrieved February 1, 2026 from https://github.com/facebook/zstd

[26] Apache Software Foundation. 2013. *Apache ORC*. Retrieved February 1, 2026 from https://orc.apache.org/

[27] Apache Software Foundation. 2013. *Apache Parquet*. Retrieved February 1, 2026 from https://parquet.apache.org/

[28] Apache Software Foundation. 2018. *Apache Iceberg*. Retrieved February 1, 2026 from https://iceberg.apache.org/

[29] Wikimedia Foundation. [n. d.]. *Wikimedia Downloads*. Retrieved July 9, 2026 from https://dumps.wikimedia.org

[30] Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal archive* 12 (1994), 23–38.

[31] Francesco Gargiulo and Rossano Venturini. 2025. OnPair: Short Strings Compression for Fast Random Access. *CoRR* abs/2508.02280 (2025).

[32] Bogdan Ghita, Diego G. Tomé, and Peter Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *CIDR*. www.cidrdb.org.

[33] Google. [n. d.]. *Snappy*. Retrieved February 1, 2026 from https://github.com/google/snappy

[34] Goetz Graefe and Leonard D Shapiro. 1990. *Data compression and database performance*.

[35] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Ganceanu, and Marc Nunkesser. 2012. Processing a Trillion Cells per Mouse Click. *Proc. VLDB Endow.* 5, 11 (2012), 1436–1446.

[36] Xiangpeng Hao, Andrew Lamb, Yibo Wu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2025. LiquidCache: Efficient Pushdown Caching for Cloud-Native Data Analytics. *Proc. VLDB Endow.* 18, 13 (2025), 5662–5675.

[37] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[38] Balakrishna R. Iyer and David Wilhite. 1994. Data Compression Support in Databases. In *VLDB*. Morgan Kaufmann, 695–704.

[39] Uthayakumar Jayasankar, Vengattaraman Thirumal, and Dhavachelvan Ponnurangam. 2021. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *J. King Saud Univ. Comput. Inf. Sci.* 33, 2 (2021), 119–140.

[40] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *SIGMOD Conference*. ACM, 843–856.

[41] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[42] Andreas Kipf, Tobias Schmidt, Ping-Lin Kuo, Skander Krid, Moritz Rengert, Luca Heller, Andreas Zimmerer, Mihail Stoian, Varun Pandey, and Alexander van Renen. 2026. Waiting to Decompress: The Economics of LLM-Based Compression. In *CIDR*. www.cidrdb.org.

[43] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26.

[44] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD Conference*. ACM, 311–326.

[45] N. Jesper Larsson and Alistair Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

[46] Robert Lasch, Ismail Oukid, Roman Dementiev, Norman May, Süleyman Sirri Demirsoy, and Kai-Uwe Sattler. 2020. Faster & strong: string dictionary compression using sampling and fast vectorized decompression. *VLDB J.* 29, 6 (2020), 1263–1285.

[47] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *Proc. VLDB Endow.* 15, 11 (2022), 3058–3070.

[48] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proc. VLDB Endow.* 16, 11 (2023), 3044–3056.

[49] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. 2024. Corra: Correlation-Aware Column Compression. In *VLDB Workshops*. VLDB.org.

[50] Raymond Chen (Microsoft). 2023. *Inside STL: The string*. Retrieved February 1, 2026 from https://devblogs.microsoft.com/oldnewthing/20230803-00/?p=108532

[51] Sabrina J. Mielke, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y. Lee, Benoît Sagot, and Samson Tan. 2021. Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP. *CoRR* abs/2112.10508 (2021).

[52] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*. OpenProceedings.org, 283–294.

[53] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[54] Weston Pace, Chang She, Lei Xu, Will Jones, Albert Lockett, Jun Wang, and Raunak Shah. 2025. Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. *CoRR* abs/2504.15247 (2025).

[55] Rasmus Pagh. 1999. Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions. In *WADS (Lecture Notes in Computer Science, Vol. 1663)*. Springer, 49–54.

[56] Calin-George Pop, Adrian Riedl, and Thomas Neumann. 2026. Compression-Aware LIKE: Matching Patterns in the FSST Domain. In *DaMoN*. ACM, 5:1–5:10.

[57] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[58] Mohaimenul Azam Khan Raiaan, Md. Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. 2024. A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. *IEEE Access* 12 (2024), 26839–26874.

[59] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (2013), 1080–1091.

[60] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (1993), 31–39.

[61] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. *Proc. VLDB Endow.* 17, 11 (2024), 3290–3303.

[62] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (2025), 4144–4157.

[63] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *ACL (1)*. The Association for Computer Linguistics.

[64] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD Conference*. ACM, 731–742.

[65] Spiral. [n. d.]. *Vortex*. Retrieved February 1, 2026 from https://vortex.dev/

[66] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. ACM, 553–564.

[67] Jan Vincent Szlang, Sebastian Breß, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights From the Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? *Proc. VLDB Endow.* 18, 12 (2025), 5126–5138.

[68] Jan Vincent Szlang, Sebastian Breß, and Evangelia Kalyvianaki. 2026. *Prod-DS*. Retrieved May 15, 2026 from https://github.com/szlangini/prod-ds-kit

[69] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706.

[70] 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.

[71] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. 2020. Neural Machine Translation with Byte-Level Subwords. In *AAAI*. AAAI Press, 9154–9160.

[72] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.* 29, 3 (2000), 55–67.

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

[74] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. (2023).

[75] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343.

[76] Vilém Zouhar, Clara Meister, Juan Luis Gastaldi, Li Du, Tim Vieira, Mrinmaya Sachan, and Ryan Cotterell. 2023. A Formal Perspective on Byte-Pair Encoding. In *ACL (Findings) (Findings of ACL, Vol. ACL 2023)*. Association for Computational Linguistics, 598–614.

[77] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. IEEE Computer Society, 59.