Good to the Last Bit: Data-Driven Encoding with CodecDB

Hao Jiang
University of Chicago
hajiang@cs.uchicago.edu

Chunwei Liu
University of Chicago
chunwei@cs.uchicago.edu

John Paparrizos
University of Chicago
jopa@cs.uchicago.edu

Andrew A. Chien
University of Chicago
achien@cs.uchicago.edu

Jihong Ma
Alibaba
jihong.ma@alibaba-inc.com

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

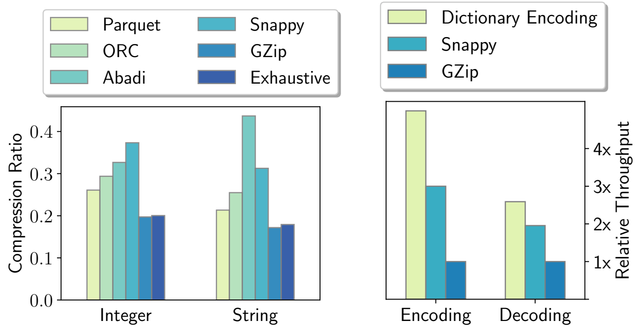
ABSTRACT

Columnar databases rely on specialized encoding schemes to reduce storage requirements. These encodings also enable efficient in-situ data processing. Nevertheless, many existing columnar databases are encoding-oblivious. When storing the data, these systems rely on a global understanding of the dataset or the data types to derive simple rules for encoding selection. Such rule-based selection leads to unsatisfactory performance. Specifically, when performing queries, the systems always decode data into memory, ignoring the possibility of optimizing access to encoded data. We develop CodecDB, an encoding-aware columnar database, to demonstrate the benefit of tightly-coupling the database design with the data encoding schemes. CodecDB chooses in a principled manner the most efficient encoding for a given data column and relies on encoding-aware query operators to optimize access to encoded data. Storage-wise, CodecDB achieves on average 90% accuracy for selecting the best encoding and improves the compression ratio by up to 40% compared to the state-of-the-art encoding selection solution. Query-wise, CodecDB is on average one order of magnitude faster than the latest open-source and commercial columnar databases on the TPC-H benchmark, and on average 3x faster than a recent research project on the Star-Schema Benchmark (SSB).

ACM Reference Format:
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 *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457283

1 INTRODUCTION

Over the past decade, columnar databases dominate the data analytics market due to their ability to minimize data reading, maximize cache-line efficiency, and perform effective data compression. These advantages lead to orders of magnitude improvement for scan-intensive queries compared to row stores [27, 63]. As a result, academic research [1, 2, 28, 52], open-source communities [7, 8], and large commercial database vendors, such as Microsoft, IBM, and Oracle are embracing columnar architectures.



(a) Compression Ratio. **(b) Throughput.**

Figure 1: Comparison of encoding schemes against an encoding selector that exhaustively evaluates encodings. The exhaustive encoding selection compresses as good as GZip and dictionary encoding is much faster than GZip and Snappy for encoding and decoding data.

Columnar databases employ compression and encoding algorithms to reduce the data size and improve bandwidth utilization. Both are important for organizations storing data in public clouds. For example, one S&P 500 employee we spoke with disclosed that their monthly cloud costs for storing Parquet files are in the six-figure range. Therefore, encoding data to reduce the storage size makes significant practical sense. Popular encoding schemes include dictionary encoding, run-length encoding, delta encoding, bit-packed encoding, and hybrids. These methods feature a reasonable compression ratio with fast encoding and decoding steps.

Many database systems support the LZ77-based byte-oriented compression algorithms [61], such as Snappy [23] and GZip [22]. Although the decompression step in these algorithms is slow and hinders query performance, people often believe they feature a better compression ratio over encoding schemes. However, this is not always the case. GZip and Snappy are one-size-fits-all compression algorithms, having no preference for the dataset. Encoding schemes are designed for datasets with particular characteristics. For example, dictionary encoding works best on datasets with low cardinalities, and delta encoding works best on sorted datasets. The nature of encoding schemes requires us to choose the encoding scheme correctly for a given dataset, which is not trivial.

To illustrate this point, in Figure 1a, we compress a large corpus of real-world datasets (see Section 6.1 for details) using GZip,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457283

Snappy, two popular open-source columnar datastores Apache Parquet [8] and Apache ORC [7], and one encoding selection algorithm from previous work [2] implemented on Parquet. We then compress the dataset with all available encoding schemes and choose the one with smallest size (Exhaustive). We see that although GZip yields a better result than Parquet and ORC, the exhaustive encoding selection achieves a similar compression ratio as GZip. In Figure 1b, we compare the throughput of dictionary encoding, Snappy, and GZip on a synthetic IPv6 dataset, and observe that this encoding scheme is 3x-4x faster than GZip in both encoding and decoding. This result implies that a good encoding selector allows us to benefit from both good compression ratios[1] and high query performance.

Despite the prevalence of columnar databases and the importance of encoding selection, limited work exists on selecting encodings for a given dataset. Seminal work by Abadi et al. [2] proposes a rule-based encoding selection approach that relies on the global knowledge of the dataset (e.g., is the dataset sorted) to derive a decision tree for the selection. Open-source columnar stores such as Parquet, ORC and Carbondata [5], and commercial columnar solutions such as Vertica [41] choose to hard-code encoding selection based on the column data type. Unfortunately, these approaches all have significant limitations. Abadi's rule-based algorithm achieves a sub-optimal compression ratio and requires multiple passes on the original dataset, which becomes prohibitively expensive when dataset size increases. Hard-coded encoding selection, as we show in Figure 1a, leads to sub-optimal results in practice.

Besides compression, the encoding schemes also facilitate efficient query processing. Most encoding schemes compress each data record individually, allowing a carefully designed iterator to locate and decode only the raw bytes corresponding to the target records, skipping the records in between [1, 2]. An advanced algorithm makes comparisons on the bit-pack encoded records without decoding any byte [30], making the query even faster. A dictionary-encoded column contains a list of distinct values. Operators, such as aggregation, can use this information to speed up execution.

Nevertheless, many open-source columnar databases [4, 19] have encoding-oblivious query engines. These systems separate the query engine and the encoded data file with a decoding layer. When the query engine reads an encoded column, the decoding layer first decodes the column into in-memory data structures, then passes the decoded data to the query engine. The query engine is blind to the encoding scheme used and has no direct access to the encoded data. This design prohibits the query engine from performing optimization towards encoded columns.

Based on these observations, we design CodecDB, a holistic encoding-aware columnar database. CodecDB demonstrates that by tightly coupling the data encoding selection in the database design, we can significantly improve the end-to-end system performance. The contribution of CodecDB is twofold. First, CodecDB provides a data-driven encoding selector to choose encoding with the best compression ratio for a given dataset. CodecDB identifies features that impact datasets' encoding performance and utilizes machine learning techniques to train a series of models to predict compression and query performance. This approach is beneficial because it requires no prior knowledge from end-users of candidate encodings, domain knowledge, or understanding details of the encoding implementation – all of which are inferred from the dataset. Our experiments show that CodecDB only needs to access a small portion of the dataset when making encoding decisions, without requiring global knowledge of the whole dataset.

Second, the query engine in CodecDB leverages advanced open-source SIMD libraries, parallel hash structures, and encoding-aware query operators to optimize access to encoded data. CodecDB achieves a significant improvement in the end-to-end performance evaluation of queries over encoded data in comparison to open-source, research, and commercial competitors. Specifically, CodecDB evaluation includes query operator micro-benchmarks, TPC-H and SSB benchmarks, and a cost breakdown analysis for a better understanding of the improved performance. This result justifies the design of a tight coupling between the query engine and the data encoding schemes. It also quantifies the benefit such a design could bring to the query performance, which is otherwise lost. We believe that this result can have significant implications considering that several recent large-scale analytic frameworks (e.g., Presto [19]) avoid this coupling to provide a simple execution engine.

We build our prototype of CodecDB with Apache Parquet columnar format [8] for its popularity, extensibility for encodings, and open-source nature. Experiments show that CodecDB's data-driven encoding selection is accurate in selecting the columnar encoding with the best compression ratio and is fast in performing the selection. Specifically, we achieve over 96% accuracy in choosing the best encoding for string types and 87% for integer types in terms of compression ratio. The time overhead of encoding selection is sub-second regardless of dataset size. We evaluate the query performance of CodecDB against the open-source database Presto [19] and a commercial columnar solution, DBMS-X, using the TPC-H benchmark. We also compare against a recent research project MorphStore [15], Presto and DBMS-X using the SSB benchmark. CodecDB is on average an order of magnitude faster than the competitors on TPC-H, and on average 3x faster than the competitors on SSB. CodecDB also has a lower memory footprint in all cases.

We start with a thorough review of the relevant background (Section 2) and present our main contributions as follows:

- We present CodecDB, an encoding-aware columnar database that achieves both storage and query efficiency (Section 3).
- We propose a data-driven method for encoding selection on a given dataset to minimize storage space with high accuracy and efficiency (Section 4).
- We implement an encoding-aware query engine to greatly improve query efficiency on encoded columns (Section 5).
- We extensively evaluate our ideas (Section 6).

Finally, we conclude with a discussion of related work (Section 7) and the implications of our work (Section 8).

2 BACKGROUND

We review the prevalent encoding schemes mentioned in the paper. The purpose of data encoding is to transform the original data into a compact format to reduce storage requirements, improve transferring speed, and potentially improve computation on the encoded data. Typical encoding schemes include lightweight encoding methods, such as bit-packed and run-length encoding, and byte-level

[1]We define the compression ratio as $\frac{\text{compressed size}}{\text{uncompressedsize}}$ [61].

compression methods, such as GZip and Snappy. These approaches achieve different compression ratios and encoding speeds.

Bit-Packed Encoding: The goal of this scheme is to store numbers using as few bits as possible. Given a list of non-negative numbers, $[a_0, a_1, \ldots, a_n]$, bit-packed encoding finds a w satisfying $a_i < 2^w$ and represents each number using w-bit losslessly. The bits are then concatenated in sequence as the encoding output. *Null suppression* [2] shares the same idea but uses two bits to indicate the byte length for the encoded values and, therefore, it encodes values by using only as many bytes necessary to represent the data. Bit-packing requires knowledge of the observed max value.

Delta Encoding: Delta encoding stores the deltas between consecutive values. Given a list of values, $[a_0, a_1, \ldots, a_n]$, delta encoding constructs b_i deltas as follows: $b_0 = a_0, b_1 = a_1 - a_0, b_2 = a_2 - a_1, \ldots, b_n = a_n - a_{n-1}$. The result can then be bit-packed. As the deltas between numbers are generally smaller than the numbers themselves, bit-packing the deltas generally allows a better compression ratio than bit-packing the original data. *FOR* and *PFOR* [39] share a similar idea, but store all values as offsets from a reference value (referred as "fixed" in Table 1) rather than the previous value (referred as "prior" in Table 1). Parquet supports DeltaLength encoding for string type, which stores the binary string consecutively as-is, and encode the string length using delta encoding.

Run-length Encoding (RLE): This scheme encodes a consecutive run of repeating numbers as a pair *(num, run-length)*. The list $[a_0, a_0, a_1, a_2, a_2, a_2, a_3, a_3, a_3, a_3]$ becomes $[a_0, 2, a_1, 1, a_2, 3, a_3, 4]$. The result may be then bit packed. The hybrid of bit-packing and RLE is used by default in Parquet's RLE implementation.

Dictionary Encoding: This scheme uses a bijective mapping (a dictionary) to map input values of variable length to compact integer codes. The dictionary used in the encoding process is prefixed or attached to the encoded data. Dictionary allows conversion from data of arbitrary types to integer codes, enabling more efficient encoding through hybrid schemes, such as RLE. Dictionaries may either be global (e.g., one dictionary per column) or local (e.g., page or block-level dictionary), which is the case in Parquet.

Bit Vector Encoding: This scheme stores values using bit vectors. Each distinct value corresponds to one bit vector showing its distribution over all positions. Bit Vector Encoding is useful when the data cardinality is very low. The list shown in the RLE example would be encoded as four bit vectors: $a_0 : [1100000000], a_1 : [0010000000], a_2 : [0001110000], a_3 : [00000001111]$.

Byte-Level Compression: Popular byte-level compression techniques, such as GZip and Snappy, originate from the LZ77 algorithm [61], which looks for repetitive string occurrences within a sliding window on the input stream. When a repetition is found starting at location i, LZ77 outputs a tuple (*prev_location*, *msg_length*, *next_char*), where *prev_location* points to the nearest previous occurrence, *msg_ length* indicates the length of the repetition, and *next_char* is the first character after the repetition. The search for repetition then restarts at location i + *msg_length* + 1. For better compression, GZip further encodes the tuples using Huffman encoding, while Snappy outputs the tuples directly for encoding speed consideration. On decoding, LZ77 maintains an in-memory buffer for decoded content, reads the tuple sequence and appends the characters to the buffer. If a tuple refers to a previous occurrence, the algorithm looks back in the buffer to find it.

Table 1 shows the encodings schemes supported by state-of-the-art columnar databases and storage formats. We merge similar encodings and omit encodings rarely supported or only used in a specialized context. Parquet [8] is an open-source column-oriented storage format. A Parquet file consists of several row groups (i.e., horizontal partitions). Each row group contains several column chunks (i.e., columnar data for the row group), and each column chunk consists of data pages that serve as a unit of encoding/-compression. Queries can locate and access each row group and column chunk independently, facilitating parallel processing. Data within the same column chunk are physically adjacent on disk for compression and query I/O. Parquet supports a wide variety of encodings and has an open design to support new encoding schemes. It is supported by a variety of engines like Hive [53], Impala [37], Pig [43], and Spark [58]. For these reasons and due to its popularity and open-source nature, we build CodecDB on top of Parquet.

3 SYSTEM OVERVIEW

CodecDB consists of a storage engine and a query engine. We demonstrate the system architecture in Figure 2.

The storage engine trains a machine learning model for the encoding selection task. When CodecDB runs for the first time, the pre-processing module executes the following tasks. First, a data collection task reads the training dataset prepared by the end-user or a default provided dataset, splits each table into columns, determines the column data type, and encodes each column using all available encoding schemes. The feature extraction task then extracts features from both the raw data columns and the corresponding encoded files. The extracted features are then used to learn how to rank encodings based on the compression ratio. We describe our encoding selection process in more detail in Section 4.

When the pre-processing tasks complete, CodecDB gets the runtime module of the storage engine online and is ready to encode input datasets. When the user loads a new data table, the runtime module samples each column, computes features using the samples, runs encoding selection with the features, and encodes the column. It also records encoding-related metadata such as the column's dictionary and bit-width of the encoded records. CodecDB persists the metadata on disk as a plain text file and maintains it in memory as a hashmap. The query engine uses the metadata to optimize access to encoded data. We scan the entire data column in the pre-processing step when extracting features for better accuracy. In the runtime phase, we sample from the column for performance consideration. We discuss more on sampling in Section 6.2.2.

The query engine consists of two major components: an execution engine and a thread pool manager. The execution engine is the core component responsible for executing queries. The execution engine reads a query plan and builds an optimized acyclic directed graph of query operators. The query engine associates one worker task with each operator and sends the task group to the thread pool manager for execution. The execution engine returns the results to the end-user when the task group finishes execution. In Section 5, we demonstrate more features of the query engine, including lazy evaluation, data skipping, and batch execution.

CodecDB provides support to common operators, including filter (selection), join, aggregation, and sort. We optimize these operators

Table 1: Popular encodings supported by non-commercial columnar database systems

	RLE	Dict	Delta/ FOR/PFOR	BitVector	BitPacked/ Null Suppression	Dict-RLE/BP
C-Store	✓	✓ (global)	✓ (prior)	✓	✓	✗
Parquet	✓	✓ (local)	✓ (fixed)	✗	✓	✓
Carbondata	✓	✓	✓ (fixed)	✗	✓	✗
ORC	✓	✓ (local)	✗	✗	✗	✗
MonetDB	✗	✓ (global)	✓ (fixed)	✗	✗	✗
Kudu	✓	✓	✗	✗	✗	✗
CodecDB	✓	✓ (global)	✓ (prior)	✓	✓	✓

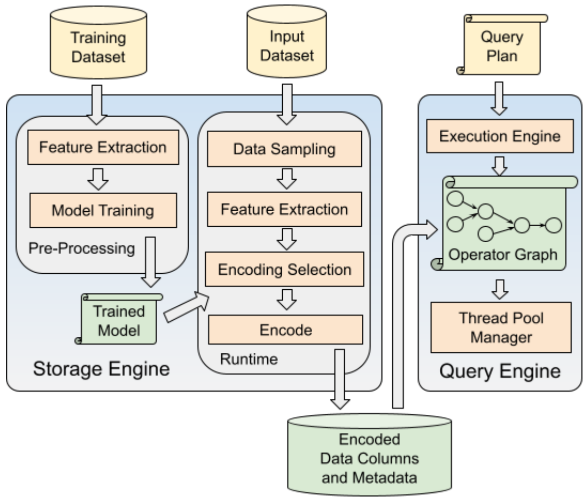


Figure 2: CodecDB System Architecture

to access encoded data, which is the main reason for CodecDB's performance improvement. We demonstrate the design of these operators in Sections 5.3, 5.4, and 5.5. The current prototype of CodecDB focuses on the execution optimization to encoded data. It does not include a query optimizer and relies on an external component to provide a feasible query plan.

4 LEARNING TO SELECT ENCODINGS

Lightweight encodings are each designed to accompany datasets with specific characteristics. Encoding selection is thus crucial to system performance, and hard-coded encoding selection often fails to achieve a desirable result. In this section, we introduce our data-driven encoding selection solution for optimizing compression ratios in CodecDB. We model the encoding selection as a learning-to-rank problem, identify a series of features affecting the compression ratio of encoding schemes, collect a large amount of data columns from real-world applications, and train a model to estimate the compression ratio of a given encoding scheme on a dataset.

4.1 Learning a Ranking Model

To train a ranking model, we consider a set of data columns $C = \{c_1, c_2, ..., c_m\}$, and a set of encoding schemes $E = \{e_1, \ldots, e_n\}$. Each data column c_i is associated with a list of compression scores $S_i = \{s_{i1}, s_{i2}, ..., s_{in}\}$, where s_{ij} corresponds to the relevance of encoding scheme e_j to column c_i. In CodecDB, we let s_{ij} be the compression ratio of e_j on c_i. We then create a feature vector $f_{ij} = F(c_i, e_j)$ for each encoding-column pair (c_i, e_j). The pair of feature vector and score (f_{ij}, s_{ij}) then form an instance in the training set.

The objective is to learn a scoring function $score : (C, E) \rightarrow \mathbb{R}$, which takes an input of data column and encoding, and output a score, that minimizes the total loss with respect to the training set:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} loss(score(c_i, e_j), s_{ij}) \qquad (1)$$

Algorithms based on neural networks have shown great promise in learning such functions for various applications [11, 12, 32, 54]. We exploit a simple neural network that takes the column-encoding pair (c_i, e_j) as input and learns score s indicating the compression ratio e_j can achieve on c_i. We describe our network configuration in Section 6.2. An intuitive explanation to this model is that if a new column-encoding instance presents a similar feature set to some known instances, it should also yield a similar compression ratio.

4.2 Feature Extraction

The features reflect a given dataset's characteristics that affect its compression ratio under the encoding schemes we study. We expect our method to make encoding selection by only accessing the first several blocks of the file rather than scanning and parsing the entire file. Therefore, these features must be computed on a subset of the records in the column. In this section, we describe the features we develop in CodecDB to assist encoding selection, where we use N to denote the number of values in a target column, and $[a_1, a_2, \ldots, a_n]$ to represent the values in the column.

Value Length: We compute each value's length in the target column as the number of characters in its plain string representation and compute statistical information including mean, variance, max, and min. The compression ratio of bit-packed encoding is closely related to the length distribution of data records. Shorter records compress better.

Cardinality Ratio: Cardinality ratio is the ratio of number of distinct values vs. the number of values in the dataset:

$$f_{cr} = \frac{C_N}{|N|}$$

Where C_N is the cardinality of N. To process datasets with large cardinalities, we adopt a linear probabilistic counting algorithm proposed by Whang et al. [55]. We maintain a bitmap B, compute a hash value for each record and insert a bit into the corresponding location of the bitmap. Let o be the number of occupied bits in the bitmap, the cardinality then can be estimated as follows:

$$C_N \approx -|B| \log \left(1 - \frac{o}{|B|}\right)$$

Cardinality ratio has a direct relationship with dictionary encoding. A dataset with high cardinality ratio is unlikely to be compressed well by dictionary encoding as there are too many distinct entries.

Sparsity Ratio: Sparsity ratio is the number of non-empty records vs. total number of records:

$$f_{ne} = \frac{|\{i|a_i \text{is not empty}\}|}{|N|}$$

A high sparsity ratio means there are many empty entries in the dataset, and implies a better compression ratio with schemes that looks for repetitions in the dataset, such as dictionary encoding and byte-compression.

Entropy We treat the dataset as a byte stream, and compute its Shannon's entropy:

$$f_e = \sum_{c_j \in C} -p(c_j) \log p(c_j)$$

where $C = \{c_k | \exists i, c_k \in a_i\}$ is the collection of characters in the string, and $p(c_j) = \frac{\sum_{i,k} \mathbb{I}(a_i[k]=c_j)}{\sum_i |a_i|}$ is the frequence of character c_j. We also compute Shannon's entropy separately for each value in the column, then collect the statistical information, including mean, variance, max, and min of the entropy values. Shannon's Entropy provides a lower bound for the theoretical best compression ratio that can be achieved by any encoding / compression schemes. In general, a lower entropy value means less information is included in the dataset, which implies a better compression ratio for dictionary encoding, bit-packed encoding, and byte-compression.

Repetitive Words: As described in Section 2, most popular byte-compression algorithms that belong to the LZ77 family work by encoding repetitive occurrences of strings as tuple (*prev_location*, *msg_length*, *next_char*), which basically refer to the nearest previous occurrence of the string. The compression ratio of the LZ77 family can be computed as $ratio = \frac{L_c}{L_s} \sum_{m=1}^{M} K_m$ where L_s is the input data length, L_c is the tuple length, M is the maximal message length, and K_m is the number of messages of length m. As L_c, L_s, and M are all constants, we can explore the efficiency of the compression algorithms by making an approximation on K_m, the number of distinct repetitive words in the dataset.

We use a block-based analysis algorithm similar to what is used in LZ77. We treat the input dataset as a byte stream and read a block of size S from it. Starting from the beginning of the block, we scan the content and record the substring $s(i, j)$ we met so far, where i is the start point of current scan, and j is the current read position. If $\exists\ k < i, s(k, k + j - i) = s(i, j)$ (i.e., $s(i, j)$ occurred before) we restart the scan starting from $j + 1$. When reaching the end of the block, we record the total number of new messages discovered, as well as their length distribution. For efficiency, we represent a string with its Karp-Rabin fingerprint [33]. Given a string $a = a_0a_1a_2 \dots a_n$, a large prime number p and a random $r < p$, the Karp-Rabin fingerprint $krf(a)$ is defined to be

$$krf(a) = (\sum_{i=0}^{n} a_i r^i) \mod p$$

This representation also allows easy substring concatenations as $krf(concat(a, b)) = krf(a) + r^{|a|} krf(b)$. Converting a string to its fingerprint allows our algorithm to make faster string comparisons and requires less space for intermediate result. The probability that two different substrings have the same fingerprint is very low [33] and we ignore such cases.

Sortedness: The sortedness of a dataset evaluates how "in order" a dataset is. Many encodings schemes can achieve better a compression ratio from a well-sorted dataset. For example, run-length encoding can generate longer runs on a sorted dataset than the same dataset with records randomly organized. Delta-bitpacked hybrid encoding can also benefit from the sorted dataset as the delta values between sorted entries tend to be smaller and thus can yield to shorter bit-packed entries. A previous method [2] use a boolean value to represent whether a dataset is sorted or not. However, we observed that a continuous variable more robustly captures the sortedness property of a dataset, as in practice, many datasets can be "partially" sorted and these partially sorted datasets still benefit from certain encodings. For example, a dataset is 90% sorted will have longer runs than a non-sorted dataset.

We adopt three methods of evaluating the sortedness of a column, f_s, and include all of them in feature sets. Kendall's τ [34] and Spearman's ρ [16] are two classical measures of rank correlation. For our purpose of evaluating the sortedness of a given dataset, Kendall's τ is computed as

$$\tau = 1 - \frac{2\left|\{(a_i, a_j) | i < j, a_i > a_j\}\right|}{n(n-1)/2}$$

and Spearman's ρ is computed as

$$\rho = 1 - \frac{6 \sum_{i=1}^{n} (s_i - i)^2}{n(n^2 - 1)}$$

Both methods generate a real number in $[-1, 1]$. 1 means the dataset is fully sorted, and -1 means the dataset is fully inverted sorted. However, most lightweight encodings will work just as well on a fully inverted sorted dataset if it works well on a fully sorted one. Observing this, we define a variant, called absolute Kendall's τ.

$$\tau_{abs} = 1 - |1 - 2\tau|$$

and τ_{abs} has a value range of $[0, 1]$, and approaches 0 when the dataset is close to either fully sorted or fully reverse sorted.

Computing the sortness features on the entire column have a time complexity of $O(n^2)$, which is prohibitively time-consuming. Therefore, we adopt a sliding window method. We slide a window of size W over the dataset and with probability p perform computation on pairs within that window. There are in total $n - W + 1$ such windows, and for each window, the time complexity is $O(W^2)$. The time complexity will be $p \cdot (n - W + 1) \cdot O(W^2)$. By setting p to $\Theta(\frac{1}{W^2})$, we can perform the computation in $O(n)$.

4.3 Dataset Collection

We derive our training set from various structured data collections, including open city data portals, scientific computation cluster logs, machine learning datasets, traffic routes, and data challenge competitions. We describe the dataset in detail in Section 6.1. We develop a data collection framework consisting of a file reader, a feature extractor, and a data store. The file reader uses file extensions to determine the format and invokes a corresponding parser, which supports common tabular file formats, including CSV, TSV, JSON, and Microsoft Excel formats. The file reader splits a file into columns and infers each column type, then extracts features on the generated columns. The framework stores the generated columns as separate files in the file system and metadata and extracted features in a DBMS.

We use the encoding algorithms shipped with Apache Parquet. We apply all viable encodings to each column to find the encoding compressing the column with minimal size. The encoding scheme having the smallest compression ratio is chosen as the "ground-truth" in the training phase. We train the model using data sources with extensive coverage and expect users can use it in various scenarios. However, including new encoding types would require re-training of the model.

5 ENCODING-AWARE QUERY ENGINE

We build an encoding-aware columnar query engine in CodecDB. With it, we demonstrate that a holistic system design with a tight coupling between encoded columns and database operators significantly improves the end-to-end query performance. The query engine maintains the encoding information of data columns and applies operators optimized for the corresponding encodings. These operators rely on recent algorithms that provide higher throughput, consume less CPU time, and less memory footprint than previous similar solutions. The query engine also provides features targeting big data analysis, such as lazy evaluation and batch execution. Next, we introduce the operators and these features of the query engine.

5.1 In-Memory Data Structures

CodecDB keeps the execution results of operators in in-memory data structures, also known as mem tables. When the operator performs a selection on one input table, we store the result in a bitmap, with each bit marks each row's validity. Our bitmap provides SIMD-based logical operations of bitmaps and a fast iterator allowing users to access all marked positions. As the input tables can be arbitrarily large, CodecDB provides a sectional bitmap consisting of multiple small bitmaps, with each section corresponding to a data block. Individual sections can be cached to an external storage, or be compressed individually to reduce memory footprint. CodecDB supports compressing the bitmap using run-length encoding.

CodecDB provides row-based and columnar mem tables, which can be used to store results from aggregation, join and sort operators. The mem tables support common primitive types, including int32, int64, float and double, and variable-length binary type. CodecDB uses a 'zero-copy' strategy for binary data. Each binary field in mem table is a `struct binary {uint8_t* ptr, uint64_t len}`, where `ptr` is a pointer to the start of the binary record, and `len` the length of the record. When loading a binary column from external data files, CodecDB decodes the binary records into the internal buffer and returns a reference to that record. Subsequent operations that move binary records between mem tables only involve copying the pointer, not moving the data.

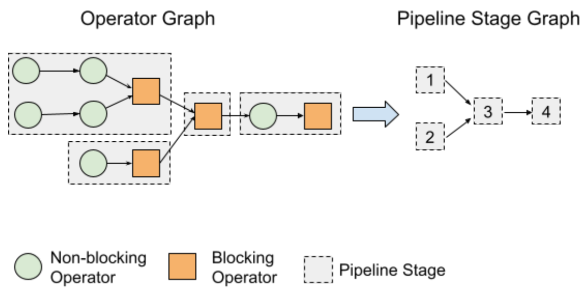


Figure 3: Lazy Evaluation groups operators to pipeline stages

5.2 Operator Evaluation

CodecDB evaluates all operators in an operator graph in parallel to utilize multi-core platforms. The parallelism happens on two levels, operator level and data block level. When executing a query, CodecDB treats each operator as a task and submits the task group to an operator thread pool. Each task is blocked until all its ancestors finish. Operators not related (e.g., two filters on different columns) can thus run in parallel. Operators also access their input data in parallel by splitting the input as multiple data blocks and use a data processing thread pool to process the blocks. All operators share the same data processing thread pool, and we configure the thread pool size to limit the memory footprint used by each query.

CodecDB utilizes pipelined evaluation of operators and late materialization to reduce memory footprint of temporary relations. To group query operators into pipelines, we implement lazy evaluation of operators. Lazy evaluation does not execute an operator immediately when it is invoked. Instead, it maintains a record of operators that have been called, and executes them all together with a pipeline when reaching a blocking operator, such as sort and aggregation. With lazy execution, we convert the operator graph into a directed acyclic graph of pipeline stages. Each of these stages contains multiple non-blocking operators and one blocking operator. Evaluating a blocking operator executes the corresponding pipeline stage. We show an example of this process in Figure 3. The blocking operators separate the operators into four pipeline stages. The first two stages have no dependencies and execute in parallel, and other stages start execution when their ancestors finish execution.

CodecDB categorizes an operator as non-blocking if it can execute locally on a single data block, without global information from the entire data file. Filtering and probing a hash table are examples of non-blocking operators. An operator is blocking if it reads multiple data blocks. Building a hash table, aggregation, and sorting are examples of blocking operators.

To execute pipeline stages, CodecDB designs a data stream framework and implements a demand-driven pipeline based on it. A stream of type T, represented by `Stream<T>`, provides two functions: *map(function<S(T)>)* and *foreach(function <void(T)>)*. Users create a pipeline by obtaining a data stream, calling *map* to add operations to the pipeline, and calling *foreach* to execute the pipeline. For example, we show how to build a pipeline to count the positive values in an integer column. We first get a stream of data

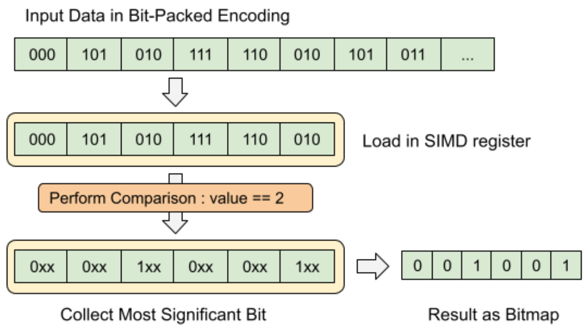


Figure 4: SBoost *in-situ* Scan for Bit-packed Data

blocks `Stream<Block>` from the column, and call *map* with a *function<Bitmap>(Block)>)*. This function scans a block and returns a bitmap marking positions of all positive numbers. We then call *foreach* with a *function<int(Bitmap)>*, which counts the size of each bitmap and sum them up to get the result. The call to *foreach* triggers the pipeline execution and returns the results.

CodecDB supports batch executions of operators accessing the same data column to reduce disk read and improve cache locality. It searches in the execution graph for operators reading the same data column and groups them. When the first operator in the batch group executes, the engine reads disk files, feed it to all operators in the group, and caches the result. When subsequent operators run, the engine directly fetches results from the cache.

After filtering a data column and obtaining a bitmap, CodecDB uses the bitmap to retrieve data from other columns, known as late materialization [52]. We optimize data retrieval by implementing data skipping in all column readers. Data skipping allows readers to jump to the next valid record marked by the bitmap, skipping all records in the middle without reading them. Data skipping save both disk IO and CPU cost. Without data skipping, column readers need to read all records, decode them, and discard those not required. CodecDB implements data skipping on three levels:

- Data Block Level. When the corresponding bitmap section is empty, CodecDB skips the entire data block. Skipping a data block saves disk I/O.
- Data Page Level. Parquet splits each data block into multiple data pages and compresses each page independently. CodecDB will skip the whole page without decompressing it if the next row to read surpass the boundary of an unread page. Skipping data pages saves decompression effort.
- Row Level. CodecDB computes the number of bytes corresponding to the number of rows to skip, reads those bytes from the input, and discards them. Skipping rows saves decoding effort.

5.3 Filter Operator

CodecDB provides two families of filter operators optimized for dictionary encoding and delta encoding, based on SBoost [30]. SBoost is an open-source library containing fast decoding and in-place search algorithms for lightweight encodings utilizing SIMD instructions. We briefly review how it works here.

One core algorithm in SBoost is in-place scanning of a bit-packed data stream, as is shown in Figure 4. The algorithm loads multiple bit-packed data entries into a SIMD register and compares all entries in parallel. It then fetches the most significant bits from each entry as a bitmap, representing the comparison result. SBoost supports all relational operators, including equal, less, greater, and their combinations. The advantage of this algorithm is two-fold: first, the algorithm performs comparison directly on the encoded data and does not decode the data. Skipping the decoding saves a huge computation effort. Second, the algorithm uses SIMD to perform comparisons on multiple data entries in parallel. For example, when scanning a bit-packed stream of size 10 (each entry takes 10 bits), SBoost uses only 8 instructions on average to process 50 entries, achieving over 20x throughput compared to the scalar algorithm that first decodes each entry then performs the comparison.

As we mentioned in Section 2, dictionary encoding maps data entries to integer keys, and bit-pack the keys. CodecDB provides a single column filter operator on dictionary encoding. The operator uses the data column's dictionary to translate the query value to an integer key, then invokes SBoost to perform an in-place search on the bit-packed keys to find the target. This filter operator also supports greater, less, and range comparisons, if the dictionary is order-preserving. In an order-preserving dictionary, for any two entries $key1 = value1$, and $key2 = value2$, we have $value1 > value2 \iff key1 > key2$. With an order-preserving dictionary, we can rewrite any comparison on the encoded values to comparisons on the keys and invoke SBoost to execute the query.

This single column filter operator does not only support comparison predicate but also `LIKE` and wildcard operations. Examples are *p_type like '%BRASS'* in TPC-H query 2 and *l_shipmode in ('MAIL', 'SHIP')* in query 12. The operator translates these queries as a logical disjunction of multiple equality operators. For example, to execute *p_type like '%BRASS'*, the operator scans the dictionary entries and finds all entries ending with '%BRASS', performs equality predicate for each entry as we described in the previous paragraph, and makes a logical `OR` on the result bitmaps to obtain the final result.

We use a similar idea to build a filter operator to compare multiple data columns using the same order-preserving dictionary. One example is comparing two `DATE` columns,e.g., `o_commitdate < o_receivedate`. Giving the two columns sharing the same order-preserving dictionary, the value of `o_commitdate` is less than `o_receivedate` if and only if the corresponding key of commit date is smaller than that of receive date. We extend the SBoost algorithm to support the comparison between two bit-packed data streams and use the result bitmap as the filter output.

Lastly, CodecDB provides a filter operator optimized for delta encoding. Due to the nature of delta encoding, we need to decode the entire data column before making a comparison. SBoost provides a SIMD algorithm to compute the cumulative sum of 8 integers fast, which we use to speed up the decoding process. CodecDB's delta filter loads a list of delta values into memory, invokes SBoost to compute their cumulative sum, and uses SIMD comparison instruction to compare them against the target value.

5.4 Aggregation Operator

CodecDB provides an aggregation operator optimized for data columns with dictionary encoding for the aggregation key. The operator starts by creating an array of aggregation results with the same size as the dictionary. It then reads each row to be aggregated, fetches the integer key from the group by column, and uses it as an index into the array to update the aggregation result. This approach works because the integer key is always a value between 0 and dictionary size. We call this operator array aggregation, for it uses an array to keep the aggregation results.

Compared to the widely used hash aggregation, array aggregation has several advantages. First, hash aggregation needs to compute a hash value from the key. Array aggregation directly uses the stored integer key and does not require additional computation. Second, hash keys can collide, and hash aggregation needs to employ a collision resolution such as open addressing, at a performance cost. Array aggregation has no collisions. Finally, when performing aggregation using multi threads, we first aggregate multiple data blocks in parallel and merge the result. Merging two hash tables is far less efficient than merging two arrays.

Array aggregation executes efficiently for small key spaces, and is only applicable to dictionary encoding. When the key space is large or the column is not dictionary encoded, CodecDB provides a stripe hash aggregation operator, a variation of hash-based aggregation. The operator first splits each data block into stripes by aggregation key. This step guarantees that the same key will always go to stripes with the same index, and tries to spread keys evenly to each stripe. In the current implementation, we compute the stripe index as the key modulo the number of stripes. Next, it performs hash aggregation independently on each stripe in parallel. Finally, it merges stripes with the same index from different blocks together. The major advantage of stripe hash aggregation versus the vanilla version is that it splits a big key space into multiple small ones, and uses several small hashtables instead of a single big one in aggregation. Smaller hashtables facilitate better cache locality, and using several small hash tables allows updates and merges in parallel. These advantages enable stripe hash aggregation to provide better performance than vanilla hash aggregation.

5.5 Other Operators

In this section, we briefly introduce other operators CodecDB provides, not optimized for encoded data.

CodecDB provides nested loop join, block nested loop join, and hash join. For hash join, we adopt phase concurrent hashmap(PCH) proposed by Shun et al. [50]. PCH uses a lock-free algorithm allowing operations of the same type to run in parallel. Multiple threads can perform insertion only, search only, or deletion only at the same time with no data races. In CodecDB hash join has two phases. The first one is building the hash table from one table, involving only insertion operations. The second one is searching the hash table (in a typical hash join) or removing entries from it (in hash-based exist join). The two phases do not overlap as the second phase can only start after the hash table is ready. This allows us to use PCH in all hash-based join operators. CodecDB provides an in-memory sort operator and an external merge sort operator. For top-n queries, it offers an in-memory heap-based top-n operator.

6 EXPERIMENTS

In this section, we show the experimental results demonstrating that CodecDBshows significant improvement in both storage and query efficiency. Storage-wise, we show that CodecDB's data-driven encoding selection can accurately identify the encoding with a good compression ratio for various datasets. We also provide an in-depth analysis of the reason our approach excels competitors. Query-wise, we demonstrate CodecDB's query operators outperform their encoding-oblivious competitors. We also show the query engine outperforms open-source query framework, commercial columnar database, and a recent research project in two established benchmarks, in both query time and memory footprint. We further elaborate on how CodecDB achieve such improvement.

Table 2: Datasets Statistics By Category

Category	Table Count	Column Count	Data Size(GB)
Server Logs	166	3836	20.4
Government	256	5126	26.8
Mach. Learning	111	3113	12.5
Social Network	98	1593	23.9
Financial	91	1954	16.8
Traffic	50	2826	22.8
GIS	16	382	5.2
Other	8	428	1.6

6.1 Environment Setup

The experiment platform has two Intel(R) Xeon(R) Gold 6126@ 2.60GHz, 192G memory, and 250G SATA SSD. It runs Ubuntu 18.10 with kernel version 4.15.0-101. We build our CodecDB prototype in Java (encoding selection) and C++ (storage engine and query engine). The Java part runs with OpenJDK 1.8.0-252 and Scala 2.12.4. The C++ part is compiled using GCC 7.5.0 with `-O3`.

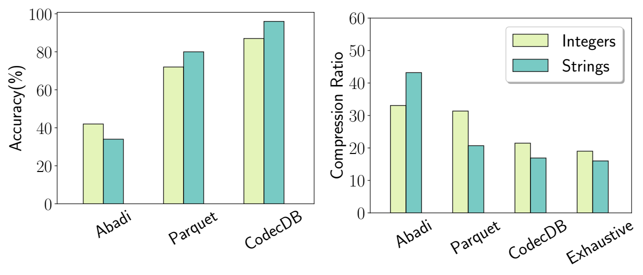
The datasets we use for data-driven encoding selection come from multiple public data sources, covering a wide variety of domains and application scenarios. Table 2 shows the statistical overview of datasets by their categories. These domains generate and store massive amounts of data, facilitating many vital applications.

Columns of string and integer types dominate the dataset (over 76%). Columns of double type also occupy a considerable portion (17%) in the datasets, most of which belong to GIS, machine learning, and financial datasets. However, Parquet only supports Dictionary encoding for double attributes, and double attributes usually have high cardinality, making it unfit for Dictionary encoding. We choose to focus on string and integer types in our experiment.

In query evaluation, we compare against Presto version 0.226 and a commercial columnar solution DBMS-X with the latest version, on the TPC-H benchmark. We also compare against MorphStore [15], Presto and DBMS-X on SSB. All systems run on the same hardware platform mentioned above.

6.2 Data-Driven Encoding Selection for Compression

In this section, we evaluate the accuracy of our neural network based data-driven encoding selection method for improving compression ratios. We use a standard MLP neural network for both the classification and the regression task. We construct a two-layer neural network with 1000 neurons in the hidden layer, using tanh as the activation function. We use sigmoid for output, and cross-entropy as the loss function when performing ranking. We train



(a) Selection Accuracy **(b) Encoded Size**

Figure 5: Accuracy and Encoded File Size of CodecDB's Encoding Selection

the network with Adam [36] for stochastic gradient descent using default hyper-parameters($\alpha = 0.9, \beta = 0.999$). The step size is 0.01, and decay is 0.99. We use 70% of data columns for training, 15% for dev, and 15% for testing. No noticeable impact is observed when we change the way of partitioning the dataset (e.g., 80/20 for training/testing). Feature `Sortness` has a hyperparameter W for sliding window size. We choose window size to be 50, 100, and 200, and include all results in the feature set.

We also compare the accuracy of our method to other candidate approaches. Abadi et al. [2] propose an encoding selection method based on a hand-crafted decision tree. They use features that are similar to what we employ in this paper, including cardinality and sortedness (although binary), and empirically setup selection rules. We refer to this decision tree approach as *Abadi* in experiments.

Apache Parquet has a built-in encoding selection mechanism which simply tries Dictionary encoding for all data types. When the attempt fails, it falls back to a default encoding for each supported data type. In practice, we notice that such a failure is primarily caused by the dictionary size exceeding a preset threshold, which means the dataset to be encoded has high cardinality. So this can also be viewed as a simplified version of a decision tree. We refer to this rule-based approach as *Parquet* in experiments.

In Figure 5a, we show selection accuracy of different approaches, which is the percentage of samples the algorithm successfully choose the encoding with minimal storage size after encoding. For string columns, CodecDB achieves 96% accuracy, a significant improvement from Abadi's decision tree with only 32% accuracy, and Parquet's encoding selection of 80%. For integer columns, CodecDB achieve 87% accuracy, also a substantial gain from Abadi's 40% and Parquet's 72%. Note that we evaluated alternative machine learning models and settled on a neural network as it provides the highest accuracy. Several other models had high accuracy, which also justifies that our features engineering represents critical characteristics of the dataset for encoding. In Figure 5b, we show how much storage reduction each algorithm can bring to the entire dataset, where "exhaustive" is the observed best encoding scheme after exhausively testing all valid encoding schemes for an attribute. CodecDB's encoding selection can bring ~30% size reduction compared to Parquet, and delivers a compression ratio close to the exhaustive result. The compression ratio is also competitive against commercial columnar stores. On TPC-H dataset of scale 20, CodecDB compresses data tables to 9.8G, 10% smaller than DBMS-X, which compresses tables to 11G. Presto uses the same data tables as CodecDB.

Next, we evaluate whether some features play more important roles than others. We iteratively remove each feature from the set and retrain the network with the same parameters. The result shows that removing any feature brings a drop in prediction accuracy of 18~25%, with cardinality and length leading the drop. This result is expected as most features are designed to map to specific encoding schemes. For example, sortness is important to delta and RLE encoding, cardinality is crucial to dictionary encoding and affects bit-packed encoding. As a result, removing any of the features leads to a misprediction on a subset of encoding schemes.

6.2.1 *Case Study: Where previous methods fail.*

We have chosen three typical cases to show where Abadi and Parquet's methods under-perform compared to our approach.

Case 1: Abadi Tree for High Cardinality Abadi's approach has the following selection path: if the number of distinct values is greater than 50000, use either LZ compression or no compression based on whether the data exhibits good locality. However, we observe that when the number of distinct values is greater than 50000, there are still over 12% of attributes for which bit-packed encoding achieves better compression than LZ. For these cases, merely removing leading zeros result in better space savings than removing repeating values.

Case 2: Abadi Tree for Run-Length Another selection path in Abadi's approach is that when average run-length is greater than 4, it uses run-length encoding. However, we found that there are over 23% of columns having an average run-length greater than 4, where dictionary encoding performs best. This difference can be a factor of encoded key size compared with the value size, local dictionaries that leverage partially sorted datasets to provide small keys, and bit-packing or run-length dictionary hybrids.

Case 3: Parquet for Bit-Packed Parquet by default always tries to use dictionary encoding. But our data shows that for integer columns, there are only 72% of attributes have dictionary encoding as the ideal encoding. A considerable amount of the remaining integer columns can be compressed well by bit-packed encoding, which Parquet fails to choose.

We find that these methods typically suffer from the following problems that CodecDB's approach addresses:

- Unable to extend to new encoding schemes or encoding scheme variations
- A single property (e.g., run-length, cardinality) cannot distinguish different groups
- Expert knowledge-based parameters can be inaccurate (and expensive to obtain)

CodecDB does not hard-code the decision but relying on the data characteristics to make the decision, allowing it to make a better choice than previous methods.

6.2.2 *Encoding Selection on a Partial Dataset.*

We have demonstrated that a neural network-based data-driven encoding selection method outperforms the current state-of-art from academic research and open-source implementations. However, most features we employ require scanning the entire column, which is time-consuming. To mitigate this problem, we read only the first N

bytes from the column, compute features, and make decisions as in the original method. This approach eliminates the correlation between dataset size and time needed for encoding selection, making it possible to make selection decisions in constant time.

To empirically estimate how much accuracy we can achieve with only partial knowledge of the dataset, we vary N to be $10K$, $100K$, and $1M$ bytes. This experiment is conducted only on data columns whose size are larger than 10MB to avoid oversampling. Not surprisingly, the prediction accuracy decreases when a smaller M is used. However, we still manage to achieve reasonable accuracy. Our result shows that when $N = 1M$, we have 85% accuracy on integer and 94% accuracy on string. With $N = 10K$, we can get 83% accuracy on integer dataset and 92% accuracy on string dataset. which is still better than state-of-art.

Random sampling [26, 42] is another widely adopted sampling method in previous works. We also compare the result of random sampling with our approach of head sampling. When applying random sampling, the accuracy of encoding selection drops drastically to 65%, and we noticed that the misprediction primarily occurred on data columns suitable for delta encoding and run-length encoding requirements to data locality. Delta encoding measures the difference between adjacent values, and run-length encoding counts the consecutive repetition of values. As the sampled data from random sampling failed to preserve this locality, prediction using randomly sampled data does not yield a satisfactory result.

6.2.3 Performance Overhead. In this section, we study the data-driven method's performance overhead, with time consumption only involving feature extraction and model execution. The model training process is conducted offline and not included. When we set $N = 1M$, the average time for calculating the features on a data column is 57ms, and that for executing the model is 3ms. We also compare the data-driven method running time against exhaustive encoding selection, which encodes a data column with all encoding candidates and choose the one with the smallest size. Our experiment includes four encoding types for integer data and three encoding types for string data. As the encoding time is proportional to input file size, we execute CodecDB's feature extraction on the entire data column to make a fair comparison. The result shows that CodecDB is in average 2.5x faster compared to exhaustive encoding when scanning the entire column. With our default setting of sampling the first 1M bytes, CodecDB can be three orders of magnitude faster than the exhaustive approach on a 1GB data column. When the selection involves more encoding types, CodecDB will benefit more compared to the exhaustive approach.

6.3 Encoding-Aware Query Execution

In this section, we explore CodecDB's query engine performance. We start by showing micro-benchmark results of CodecDB's operators. We then show that CodecDB outperforms competitors on TPC-H and SSB. We also provide breakdown analysis that helps explain how CodecDB achieves such improvement.

In Figure 6, we test CodecDB operators on various TPC-H scales and show that CodecDB operators always outperform their encoding-oblivious competitors. We also describe these competitors after describing each operator. We first test filter operators on the dictionary encoded column. The "Single Column Compare" in Figure 6

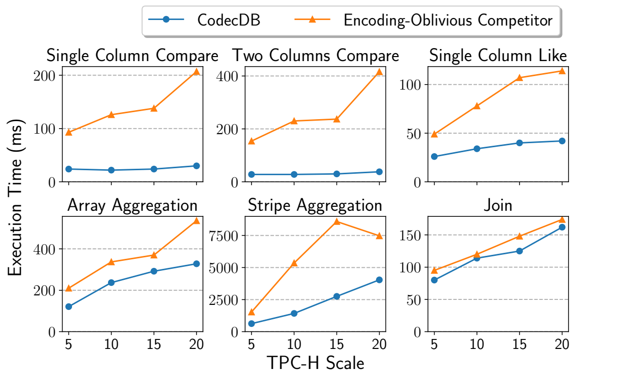


Figure 6: CodecDB operators outperform encoding-oblivious operators

corresponds to the predicate `l_shipdate <= '1998-09-01'`, "Two Columns Compare" corresponds to `l_commitdate < l_receiptdate`, and "Single Column Like" corresponds to `p_container LIKE 'LG%'`. The competitors in these cases are encoding-oblivious, who decode records from columns and make comparisons. We see that CodecDB's operators bring 5-20x performance boost compared to encoding-oblivious solutions, and has more advantage when dealing with large datasets. While encoding-oblivious solutions' time consumption almost doubles when moving from the TPC-H scale 5 to 20, CodecDB's time consumption only increases by 30%. Next, we test the aggregation operators. In "Array Aggregation", we count `lineitem` group by `l_receiptdate`. CodecDB uses an array of size 2560 to perform aggregation, and the competitor uses the Google sparsehash [24] hash table. In "Stripe Aggregation", we count `orders` group by `o_custkey`. CodecDB splits the input into 32 stripes, and the competitor uses a sparsehash hash table that does not split the input. In both cases, we see a 2-3x performance improvement. The last experiment in the micro-benchmark is our hash join operator based on phase-concurrent hashtable. The competitor uses a sparsehash hash table. We join `orders` with `customer` on the foreign key, with condition `c_mktsegment='HOUSEHOLD'`. The results show 10-15% improvement, primarily because we can build a hash table using multiple threads.

Next, we compare CodecDB query engine against popular columnar database solutions on TPC-H benchmarks of scale 20. We choose two candidates, Presto [19] and DBMS-X. Presto is an open-source distributed SQL query engine designed to query large data sets. Presto supports the Parquet storage format and executes queries in an encoding-oblivious way, making it a good candidate to show the advantage of encoding-aware query execution. DBMS-X is a commercial big data analytic system leveraging columnar storage. We encode the tables in Parquet format with column chunk size of 128M, and page size 1K. For all systems, we limit the number of concurrent threads per query to 20.

We setup Presto to use a single node, with the maximal memory per query set to 20G. Presto reads the same Parquet tables as CodecDB does. DBMS-X lacks support for many Parquet encodings, prohibiting it from reading our Parquet tables. Instead, we use its native table format with auto compression. For query efficiency, we load data into DBMS-X's Read-Only Storage, a highly optimized

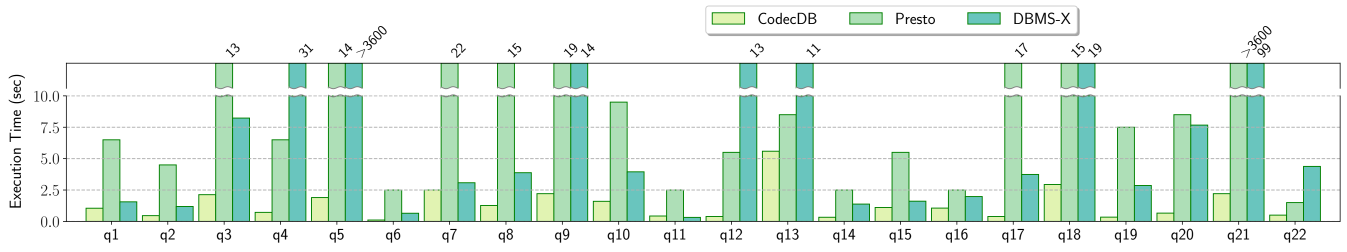


Figure 7: CodecDB outperforms encoding-oblivious columnar databases in TPC-H Benchmarks (Scale 20)

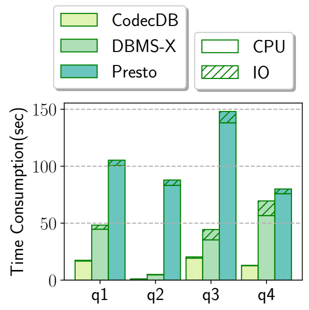


Figure 8: Time Breakdown of TPC-H Queries

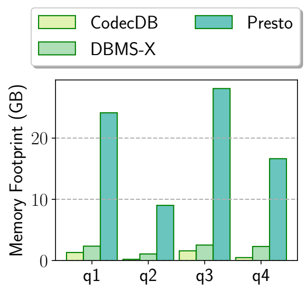


Figure 9: Memory Footprint of TPC-H Queries

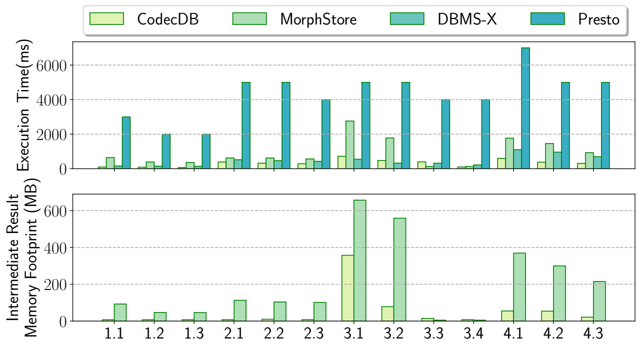


Figure 10: CodecDB is faster than MorphStore on SSB, and consumes less memory on Intermediate Results

read-oriented disk storage structure. As CodecDB is not equipped with a query optimizer, we use Presto to generate a query plan for each query, replace the operators with CodecDB's corresponding version, and manually code the query plan into CodecDB. We measure the time Presto and DBMS-X spend on generating query plans, and deduct it from the execution time to ensure a fair comparison.

We run all TPC-H queries with CodecDB, Presto and DBMS-X and show the result in Figure 7. We limit the bar graph's height and show the time consumption on the top of the bar. DBMS-X on Query 5 and Presto on Query 21 do not finish after 1 hour, we record these two outliers as ">3600" seconds, and ignore them in subsequent analysis. In general, we see a substantial performance improvement of CodecDB versus competitors. For all 22 queries, CodecDB is, in average 11.43x faster than Presto and 9.81x faster than DBMS-X, excluding the outliers mentioned above. The best result versus Presto is on Query 17, where CodecDB is 46x faster. The one for DBMS-X is Query 20, where CodecDB is 44x faster.

On queries with at least one predicate on the dictionary encoded column, CodecDB performs extremely well. We see that most queries satisfy this situation in practice. 17 out of 22 TPC-H queries (exceptions are Q9, Q11, Q13, Q18, and Q22) contains at least one such predicate. In these queries, we see at least 10x performance improvement compared to competitors.

In Figure 8, we make a time consumption breakdown of the first four TPC-H queries to understand better how CodecDB improves query performance. We see that all four queries are CPU-bound. CodecDB's encoding-aware query execution significantly reduces the CPU execution time. Besides, data skipping helps CodecDB to reduces the IO cost. Both contribute to efficient query execution. In Figure 9, we compare the memory footprint of the four queries, collected using `/proc/<pid>/stat`. We see that CodecDB saves up to 80% memory footprint compared to DBMS-X. CodecDB can execute directly on encoded data without decoding them into memory, and skip data records not accessed by the query. Both contribute to the improvement. The two experiments demonstrate the benefit that a query engine tightly coupled with encoded columns brings.

Finally, we compare CodecDB query engine with MorphStore [15] on the Star-Schema Benchmark(SSB) with scale 10. MorphStore is a columnar database that compresses intermediate results with lightweight encodings to reduce memory footprint and uses SIMD to speed up query on compressed data. It shares many similar design concepts with CodecDB. We use SSB as MorphStore does not support TPC-H benchmark. We compare the two systems on query execution speed and memory footprint of intermediate results. We also execute SSB with DBMS-X and Presto, and include their results for reference. For MorphStore, we obtain the query execution time from its running artifacts and compute the intermediate result memory footprint using the minimal compressed size. For Presto and DBMS-X, we only include the query time, as the systems are not instrumented to measure the size of intermediate results.

We show the result in Figure 10. CodecDB is again faster on most queries than all competitors. It runs SSB queries up to 5x and in average 3x faster than MorphStore and consumes less memory on intermediate results. The reason is three-fold. First, CodecDB uses a late materialization execution strategy and generates fewer intermediate results than MorphStore. Second, CodecDB relies heavily on bitmaps as intermediate results. Bitmaps are smaller in size and facilitate faster intersection/union operations. Finally, late materialization allows CodecDB to push down most filtering operations to SBoost, further speeds up the query execution. For example, in Q1.1, to perform a predicate+interesction operation, CodecDB uses 55ms and only generates a single bitmap of 7MB as an intermediate result. MorphStore takes 500ms on the same operation and generates 12 intermediate results sum up to 94MB. These experiments demonstrate that CodecDB is efficient in execution time and provides an alternative solution to reducing query memory footprint.

7 RELATED WORK

Encoding in Databases. Application of encoding techniques has been studied for various components in databases, including data table [14, 29, 38, 48], data column [44, 45, 52], index [9, 35], data dictionary [42], hash tables [25], database duplication [57], and search trees [59].

Selecting the proper encoding scheme for a database system is a trade-off between size reduction and CPU overhead in the encoding/decoding process. Classical byte-oriented encoding techniques such as GZip and Snappy have been widely supported in various DBMS systems [8, 13, 14, 29, 52]. However, studies [1, 62] suggest that these schemes come with notable CPU overhead from decompression and may significantly impact system performance. Lightweight, attribute-level compression, such as run-length and bit-packed encoding, can be beneficial to query-intensive systems.

Columnar databases, such as C-Store [52] and MonetDB [27], physically consecutively persist attributes, and allow a higher performance of lightweight encoding. Previous works [10, 31, 39] also show that for specific data sets, lightweight encoding achieves a comparable compression ratio with far lower CPU time than GZip.

Reasonable size reduction, significant low CPU overhead, and *in-situ* query execution make lightweight encoding algorithms more favorable than byte-oriented compression in columnar data stores [2].

Query Execution on Encoded Data. Compression-aware database design is necessary for query execution on compressed data. Chen et al. [14] design a cost-based optimizer for compressed database tables, and Kimura et al. [35] propose an algorithm for selecting compressed indices for a set of queries under a limited space budget.

Hardware acceleration also demonstrates high potential in speeding up the decoding operation. Willhalm et al. [56] use SIMD instruction to speed up the decoding process for bit-packed data. Jiang et al. [30] propose a SIMD-based algorithm for decoding delta-encoded data. Rozenberg et al. develop Giddy [49], a library for executing fast decoding algorithms using GPUs. Fang et al. [20, 21] propose using a co-processor for data extraction and transformation tasks in columnar encoding and compression to speed up queries. Variations of encoding formats that are optimized for hardware execution, such as VarInt [17, 51], Horizontal Bit-Interleaving [40, 47], and SIMD-Delta [39] are also proposed.

Lightweight encoding has an advantage over byte-oriented compression algorithms in that they preserve attribute boundary during encoding [1, 2], and allow algorithms [3, 30] to query on encoded data without decoding, significantly reduce the CPU overhead and enable more efficient query execution.

Cost Estimation and Encoding Selection. We can estimate an encoding scheme's efficiency from both space and time aspects, e.g., the compression ratio and encoding/decoding overhead. Popular approaches for estimating compression ratio include mathematical modeling [14, 42], regression on statistical features [9], and random sampling [26, 35]. Kimura et al. [35] propose using a bipartite graph between column and index to deduce compressed index size. For encoding/decoding overhead, most previous work [14, 35] model it as a weight in addition to normal access cost.

Encoding selection tackles a relevant problem of choosing an efficient encoding scheme for a given a data table or column. Abadi et al. [2] introduce a hand-crafted decision tree for encoding selection on a given dataset based on experience and global knowledge of a dataset (i.e., cardinality and if a column is sorted). Lemire et al. [39] focus on integer data and propose rules to choose between PFOR and bit-packed encoding.

In practice, many implementations solve the problem by hard-coding a "not too bad" default encoding per data type. Apache Parquet [8] and CarbonData [5] uses dictionary encoding for all data types and will fall back to some default encoding if the dictionary size exceeds a preconfigured limit. Apache ORC [7] uses RLE for integer and Dictionary-RLE for string types. Apache Kudu [6] uses a dictionary for string type and bit shuffle for all other data types.

CodecDB proposes using a neural network-based learning approach to rank and select from various encoding schemes. Learning to Rank is a widely adopted approach in data mining and information retrieval, in which learning algorithms are applied to datasets of labeled documents [11], document lists [12], and labeled features [18], to learn a utility function evaluating the importance of each target document. Neural network-based learning to rank approach [11, 60] has demonstrated great potential in various domain applications such as e-commerce search [32], image annotation [54], and behavior analysis [46].

8 CONCLUSION

We propose CodecDB, an encoding-aware columnar database that exploits a design tightly coupled with encoding schemes. CodecDB combines autonomous data-driven encoding selection and encoding-aware query execution to improve both storage and query efficiency on encoded columnar data. CodecDB's storage engine analyzes data column characteristics to choose the encoding scheme that best fits a given data column, achieving a compression ratio comparable to GZip. CodecDB's query engine utilizes the encoding knowledge of data columns to improve query efficiency on encoded data.

Extensive experimental results show that on both encoding selection and query execution, CodecDB brings substantial improvement compared to prior research, widely-used open-source implementations, and commercial products. Overall, as a system, CodecDB demonstrates the great potential of the system design philosophy of encoding-awareness. In the future, we plan on expanding CodecDB to support query-aware encoding selection, include new encoding schemes, lossy compression, and further explore more encoding-aware algorithms for other database operators, such as joins.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful feedback. This work was supported in part by a Google DAPA Research Award, the CERES Center for Unstoppable Computing, and gifts from FutureWei, NetApp, Cisco Systems, and Exelon Utilities. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[3] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 337–350, Oakland, CA, may 2015. USENIX Association.

[4] Apache Foundation. Apache Arrow. https://arrow.apache.org/.

[5] Apache Foundation. Apache CarbonData. https://carbondata.apache.org/.

[6] Apache Foundation. Apache Kudu. https://kudu.apache.org.

[7] Apache Foundation. Apache ORC. https://orc.apache.org.

[8] Apache Foundation. Apache Parquet. https://parquet.apache.org/.

[9] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, George Mihaila, Kenneth Ross, Sherman Lau, Cathy McArthur, Zoltan Toth, and Reza Sherkat. Efficient Index Compression in DB2 LUW. *Proc. VLDB Endow.*, 2(2):1462–1473, aug 2009.

[10] Peter Boncz, Thomas Neumann, and Viktor Leis. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.*, 13(12):2649–2661, jul 2020.

[11] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to Rank Using Gradient Descent. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM.

[12] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to Rank: From Pairwise Approach to Listwise Approach. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 129–136, New York, NY, USA, 2007. ACM.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, jun 2008.

[14] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query Optimization in Compressed Database Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 271–282, New York, NY, USA, 2001. ACM.

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

[16] Wayne W. Daniel. Spearman rank correlation coefficient. In *Applied Nonparametric Statistics*. Boston: PWS-Kent, 2 edition, 1990.

[17] Jeffrey Dean. Challenges in Building Large-scale Information Retrieval Systems: Invited Talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 1–1, New York, NY, USA, 2009. ACM.

[18] Fernando Diaz. Learning to Rank with Labeled Features. In *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval*, ICTIR '16, pages 41–44, New York, NY, USA, 2016. ACM.

[19] Facebook. Presto. http://prestodb.github.io/.

[20] Yuanwei Fang, Chen Zou, and Andrew A. Chien. Accelerating raw data analysis with the accorda software and hardware architecture. *Proc. VLDB Endow.*, 12(11):1568–1582, July 2019.

[21] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. UDP: A Programmable Accelerator for Extract-transform-load Workloads and More. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 55–68, New York, NY, USA, 2017. ACM.

[22] GNU project. GNU GZip. https://www.gnu.org/software/gzip/.

[23] Google. Snappy. http://google.github.io/snappy/.

[24] Google. sparsehash. https://github.com/sparsehash/sparsehash/.

[25] T. Gubner, V. Leis, and P. Boncz. Efficient query processing with optimistically compressed hash tables strings in the ussr. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 301–312, 2020.

[26] S. Idreos, R. Kaushik, V. Narasayya, and R. Ramamurthy. Estimating the compression fraction of an index using sampling. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 441–444, March 2010.

[27] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[28] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. An Architecture for Recycling Intermediates in a Column-store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 309–320, New York, NY, USA, 2009. ACM.

[29] Balakrishna R. Iyer and David Wilhite. Data Compression Support in Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 695–704, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[30] Hao Jiang and Aaron J. Elmore. Boosting Data Filtering on Columnar Encoding with SIMD. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, pages 6:1–6:10, New York, NY, USA, 2018. ACM.

[31] Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, and Aaron J. Elmore. Pids: Attribute decomposition for improved compression and query performance in columnar storage. *Proc. VLDB Endow.*, 13(6):925–938, February 2020.

[32] Shubhra Kanti Karmaker Santu, Parikshit Sondhi, and ChengXiang Zhai. On Application of Learning to Rank for E-Commerce Search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 475–484, New York, NY, USA, 2017. ACM.

[33] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[34] M. G. Kendall. A New Measure Of Rank Correlation. *Biometrika*, 30(1-2):81, 1938.

[35] Hideaki Kimura, Vivek Narasayya, and Manoj Syamala. Compression Aware Physical Database Design. *Proc. VLDB Endow.*, 4(10):657–668, jul 2011.

[36] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[37] Marcel Kornacker, Victor Bittorf, Taras Bobrovytsky, Casey Ching Alan Choi Justin Erickson, Martin Grund Daniel Hecht, Matthew Jacobs Ishaan Joshi Lenni Kuff, Dileep Kumar Alex Leblang, Nong Li Ippokratis Pandis Henry Robinson, David Rorke Silvius Rus, John Russell Dimitris Tsirogiannis Skye Wanderman, and Milne Michael Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[38] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 311–326, New York, NY, USA, 2016. ACM.

[39] D. Lemire and L. Boytsov. Decoding Billions of Integers Per Second Through Vectorization. *Softw. Pract. Exper.*, 45(1):1–29, jan 2015.

[40] Yinan Li and Jignesh M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 289–300, New York, NY, USA, 2013. ACM.

[41] Micro Focus International plc. Vertica Encoding Types. https://www.vertica.com/docs/9.2.x/HTML/Content/Authoring/SQLReferenceManual/Statements/encoding-type.htm.

[42] Ingo Müller, Cornelius Ratsch, and Franz Färber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*, 2014.

[43] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[44] John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J. Elmore, Michael J. Franklin, and Sanjay Krishnan. Vergedb: A database for iot analytics on edge devices. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[45] Marcus Paradies, Christian Lemke, Hasso Plattner, Wolfgang Lehner, Kai-Uwe Sattler, Alexander Zeier, and Jens Krueger. How to Juggle Columns: An Entropy-based Approach for Table Compression. In *Proceedings of the Fourteenth International Database Engineering; Applications Symposium*, IDEAS '10, pages 205–215, New York, NY, USA, 2010. ACM.

[46] Diego Perna and Andrea Tagarelli. An Evaluation of Learning-to-Rank Methods for Lurking Behavior Analysis. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, UMAP '17, pages 381–382, New York, NY, USA, 2017. ACM.

[47] Orestis Polychroniou and Kenneth A. Ross. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 9:1–9:6, New York, NY, USA, 2015. ACM.

[48] Gautam Ray, Jayant R. Haritsa, and S Seshadri. Database Compression: A Performance Enhancement Tool. 09 2004.

[49] Eyal Rozenberg and Peter Boncz. Faster Across the PCIe Bus: A GPU Library for Lightweight Decompression: Including Support for Patched Compression Schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, DAMON '17, pages 8:1–8:5, New York, NY, USA, 2017. ACM.

[50] Julian Shun and Guy E. Blelloch. Phase-Concurrent Hash Tables for Determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 96–107, New York, NY, USA, 2014. Association for Computing Machinery.

[51] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. SIMD-based Decoding of Posting Lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 317–326, New York, NY, USA, 2011. ACM.

[52] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[53] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - A Petabyte Scale Data Warehouse Using Hadoop. pages 996–1005, 01 2010.

[54] Zhang Weifeng, Hua Hu, and Haiyang Hu. Neural ranking for automatic image annotation. *Multimedia Tools and Applications*, 77(17):22385–22406, Sep 2018.

[55] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. A Linear-time Probabilistic Counting Algorithm for Database Applications. *ACM Trans. Database Syst.*, 15(2):208–229, jun 1990.

[56] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.*, 2(1):385–394, aug 2009.

[57] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. Online Deduplication for Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1355–1368, New York, NY, USA, 2017. ACM.

[58] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[59] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-preserving key compression for in-memory search trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1601–1615, New York, NY, USA, 2020. Association for Computing Machinery.

[60] P. Zhao, O. Wu, L. Guo, W. Hu, and J. Yang. Deep learning-based learning to rank with ties for image re-ranking. In *2016 IEEE International Conference on Digital Signal Processing (DSP)*, pages 452–456, Oct 2016.

[61] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[62] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.

[63] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. Vectorwise: A Vectorized Analytical DBMS. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 1349–1350, Washington, DC, USA, 2012. IEEE Computer Society.