Blitzcrank: Fast Semantic Compression for In-memory Online Transaction Processing

Yiming Qiao
Tsinghua University
qiaoym21@mails.tsinghua.edu.cn

Yihan Gao
gaoyihan@gmail.com

Huanchen Zhang*
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT

We present Blitzcrank, a high-speed semantic compressor designed for OLTP databases. Previous solutions are inadequate for compressing row-stores: they suffer from either low compression factor due to a coarse compression granularity or suboptimal performance due to the inefficiency in handling dynamic data sets. To solve these problems, we first propose novel semantic models that support fast inferences and dynamic value set for both discrete and continuous data types. We then introduce a new entropy encoding algorithm, called delayed coding, that achieves significant improvement in the decoding speed compared to modern arithmetic coding implementations. We evaluate Blitzcrank in both standalone microbenchmarks and a multicore in-memory row-store using the TPC-C benchmark. Our results show that Blitzcrank achieves a sub-microsecond latency for decompressing a random tuple while obtaining high compression factors. This leads to an 85% memory reduction in the TPC-C evaluation with a moderate (19%) throughput degradation. For data sets larger than the available physical memory, Blitzcrank help the database sustain a high throughput for more transactions before the I/O overhead dominates.

PVLDB Reference Format:
Yiming Qiao, Yihan Gao, and Huanchen Zhang*. Blitzcrank: Fast Semantic Compression for In-memory Online Transaction Processing. PVLDB, 17(10): 2528 - 2540, 2024.
doi:10.14778/3675034.3675044

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/YimingQiao/Blitzcrank.

1 INTRODUCTION

In-memory database management systems (DBMSs) offer low latency and high throughput for online transaction processing (OLTP) workloads when the working set fits in memory [50–52]. For data sets beyond physical memory, their performance degrades quickly because of expensive random I/Os to fetch tuples. Although DRAM price has been decreasing, memory is still a limiting resource because of the increasing price gap between DRAM and SSDs [26, 61].

Applying compression can increase the capacity of in-memory DBMSs with the same hardware cost, thus reducing or even eliminating disk accesses to improve performance [56, 57]. However, most existing compression techniques are designed for column-stores: they target read-mostly workloads with large batched processing [7, 8, 11, 12, 20, 23, 32, 34, 43]. To compress in-memory row-stores efficiently, the compression schemes must satisfy additional requirements. First, random access to tuples must be fast (e.g., sub-microsecond) because OLTP applications demand low query latencies [58, 59]. For example, Amazon found that a 100 ms increase in latency would lead to a 1% drop in sales [27, 35]. Second, the compression algorithms must handle newly inserted/updated tuples efficiently because OLTP workloads are typically write-heavy [49]. A frequent reconstruction of the compression model is usually unacceptable because it brings too much performance overhead [44, 46]. Unfortunately, existing compression schemes are inadequate when serving OLTP workloads: they suffer from either low compression factor (i.e. the original size divided by the compressed size) due to a coarse compression granularity or suboptimal performance due to the inefficiency in handling dynamic data sets.

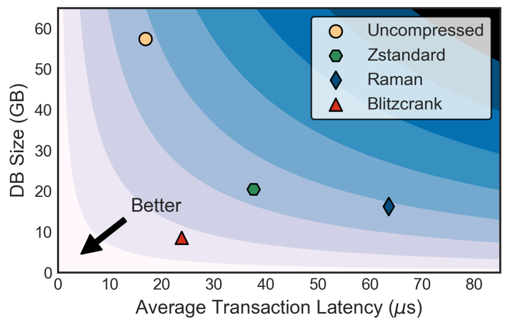


Figure 1: DB Size vs. Latency - Blitzcrank makes the size-latency trade-offs more attractive compared to other tools in TPC-C.

Coarse Retrieval Granularity. Modern general-purpose block compression algorithms such as Zstandard have high decompression throughput (up to 500 MB/s). They are widely used in operating systems, databases, file systems, and computer networks [6, 36, 55]. However, to access a single tuple, they must decompress the entire compression block [62], causing *high random-access latency*. One solution is to compress each tuple individually, but the compression factor suffers because the algorithms prefer a longer context to create an effective dictionary in the sliding window [62, 63].

Inefficient Handling of New Tuples. The classic Raman's approach [46] concatenates Huffman-coded values into variable-length tuples and then reorders the rows and columns so that it achieves a better compression factor using delta encoding. Although this solution compresses row-oriented data well, it cannot compress

Proceedings of the VLDB Endowment, Vol. 17, No. 10 ISSN 2150-8097.
doi:10.14778/3675034.3675044

*Huanchen Zhang is also affiliated with Shanghai Qi Zhi Institute.

unseen values unless it initiates an expensive model reconstruction because such a solution relies on static dictionaries.

The above approaches are considered "syntactic" because they treat the uncompressed data simply as consecutive bytes [38]. Semantic compression, on the other hand, leverages the high-level semantics in a relational table, such as value distributions and functional dependencies between columns to achieve better compression [46]. Unlike the above syntactic methods that rely on static dictionaries, the semantic approach can use the same probability models to compress new tuples effectively as long as the attribute values follow the modeled distributions [10]. Existing semantic compression methods, however, provide limited support for different data types, and their model inferences are slow. For example, Squish and the more recent DeepSqueeze take 324 and 127 seconds, respectively, to compress a 75 MB relational table [25, 28].

In this paper, we show that semantic compression can be fast, and it has potential beyond large archive compression. We present **Blitzcrank**, a high-speed semantic compressor designed for OLTP databases. Blitzcrank improves compression through both data modeling and data encoding [15].

- For data modeling, Blitzcrank introduces novel semantic models that allow fast encoding/decoding for both discrete- and continuous-value columns. It takes Blitzcrank less than one second to compress the aforementioned 75 MB table.
- For data encoding, we propose *delayed coding*, a novel fine-grained encoding algorithm that offers near-entropy compression as with Arithmetic Coding [33] while achieving a faster decompression speed compared to the modern asymmetric number system (ANS) [22].

We first compared Blitzcrank against state-of-the-art compressors that apply to *row-stores*, including Zstandard [17] and Raman's approach [46], in standalone microbenchmarks based on real data sets. Blitzcrank achieves a sub-microsecond latency (fastest among the baselines) for decompressing a random tuple while obtaining high compression factors. We then integrated Blitzcrank, along with baseline compressors, into the in-memory OLTP database, Silo [51] and evaluated it using the TPC-C benchmark [19]. The results are summarized in Figure 1. Compared to uncompressed tables, Blitzcrank reduces memory usage by 85% with a moderate (19%) throughput degradation. Compared to using Zstandard, Blitzcrank achieves a 2.4× higher compression factor and is 76% faster. When the data set exceeds the physical memory limit, Blitzcrank greatly helps the database sustain high throughput and execute 4× more transactions within the same amount of time.

The paper makes the following contributions. First, we identify the inefficiency of existing compression algorithms for OLTP databases from both data modeling and data encoding perspectives. Second, we introduce novel semantic models designed for fast inferences for discrete and continuous data types. Third, we propose the new delayed coding that is significantly faster than variants of arithmetic coding while achieving near-entropy compression. Finally, we build Blitzcrank based on the above technologies and show that semantic compression can be fast enough to make trade-offs between performance and space much more attractive than previous solutions when integrated into an in-memory OLTP database, such as Silo [51].

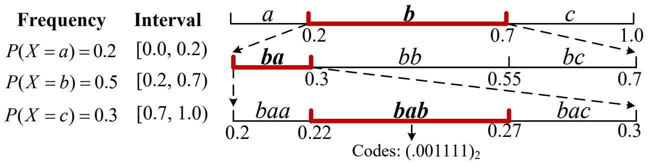
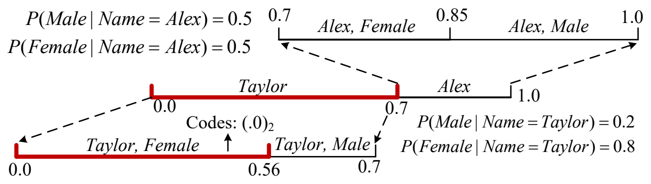


Figure 2: An Example of Arithmetic Coding - Arithmetic coding maps each possible string to disjoint probability intervals.



$P(Male \mid Name = Alex) = 0.5$
$P(Female \mid Name = Alex) = 0.5$
0.7
Alex, Female
0.85
Alex, Male
1.0
Taylor
Alex
0.0
0.7
1.0
Codes: $(.0)_2$
$P(Male \mid Name = Taylor) = 0.2$
$P(Female \mid Name = Taylor) = 0.8$
Taylor, Female
Taylor, Male
0.0
0.56
0.7

Figure 3: An Example of Column Correlation - Probabilities of column "gender" depends on the "name" column value.

2 PRELIMINARIES

This section provides the necessary background information to understand the design of Blitzcrank. Section 2.1 describes the classic arithmetic coding, which is the basis of our proposed delayed coding. Section 2.1 introduces the existing structure learning techniques adopted in Blitzcrank to leverage functional dependencies between columns for compression.

2.1 Arithmetic Coding

Arithmetic coding is one of the most widely used entropy codings for lossless compression [15, 33]. Unlike Huffman coding [42] that encodes symbols individually, arithmetic coding compresses the entire message into a single fraction $0 \leq q < 1$ with arbitrary precision. Compared to Huffman coding, arithmetic coding can achieve a higher compression factor. Arithmetic coding represents the current information as an interval, defined by two numbers (initially $[0, 1)$). Each encoding step in arithmetic coding divides the current interval into smaller sub-intervals according to the probability distribution of the alphabet and selects the one that represents the next symbol to be encoded. For example, as shown in Figure 2, the probability distribution of alphabet $\{a, b, c\}$ is 0.2, 0.5, and 0.3, respectively. To encode a message "*bab*", we divide the initial interval $[0, 1)$ into three sub-intervals $[0, 0.2)$, $[0.2, 0.7)$, and $[0.7, 1)$ and select $[0.2, 0.7)$ to represent "*b*". To encode the next symbol "*a*", we further divide $[0.2, 0.7)$ into $[0.2, 0.3)$, $[0.3, 0.55)$, and $[0.55, 0.7)$ based on the symbol probabilities and update the current interval to $[0.2, 0.3)$ which now represents "*ba*". This process continues until we reach the end of the message and obtain the final interval $[0.22, 0.27)$. We then select a fraction q within the final interval that has the shortest binary representation (e.g., $q = (.001111)_2$) as the message's code.

2.2 Structure Learning

Structure learning refers to the process of identifying correlations between columns to achieve better compression [10, 21, 25, 28]. For example, the `gender` column is often highly correlated with the `name` column in a relation. As depicted in Figure 3, 80% of `Taylors` are female, while 50% of `Alexes` are male. Instead of using static probabilities (e.g., 50% male and 50% female) for the

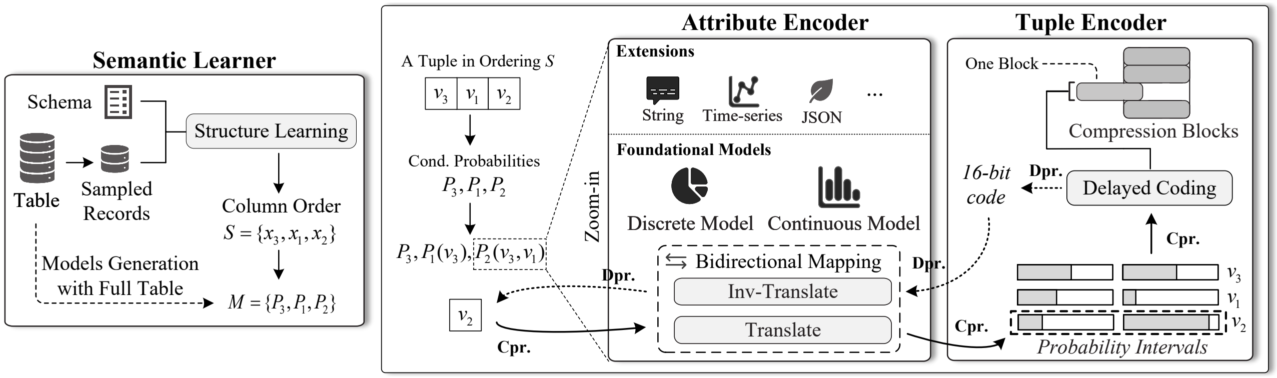


Figure 4: Blitzcrank- Semantic Learner (SL), Attribute Encoder (AE), and Tuple Encoder (TE) are three components of Blitzcrank.

gender column, we model its distribution using probabilities conditioned on the name column: $P_{\text{gender}}(\text{Female}|\text{Name} = \text{Taylor}) = 0.8$, $P_{\text{gender}}(\text{Female}|\text{Name} = \text{Alex}) = 0.5$. Then, the more common tuple (Female, Taylor) is mapped to a larger interval $[0, 0.56)$ with a short binary code $(.0)_2$, thus achieving better compression.

We use a Bayesian network (BN) to learn the best column ordering S (e.g., {name, gender}) for compression. The output also includes a model set M, where each model is a probability distribution P_x for column x conditioned on the values of all the columns preceding x in S. Determining the optimal ordering S is an NP-hard problem [30]. We, therefore, use a greedy algorithm [25] that selects the column that produces the smallest compressed size conditioned on the existing columns in S for each iteration.

3 OVERVIEW

The goal of Blitzcrank is to reduce the memory footprint of an in-memory OLTP database while imposing as small performance overhead as possible. To achieve this, Blitzcrank must be able (1) to handle newly inserted tuples with unseen values efficiently and (2) to deliver low latency and high compression factor for individual tuples. For (1), we build semantic models that describe the values' (conditional) probability distributions instead of using static value dictionaries (Section 4). For (2), we propose delayed coding that offers fast and fine-grained encoding/decoding with near-entropy compression (Section 5). Blitzcrank is optimized for single-tuple retrieval. A larger compression granularity may improve the overall compression factor, but it introduces decompression overhead for point accesses common in OLTP workloads.

Blitzcrank sits above the table storage to compress and decompress tuples while remaining transparent to the execution engine of an OLTP database. When the execution engine inserts a tuple into a relation, Blitzcrank compresses that tuple before sending it to the table storage. Upon receiving a tuple-fetching request, Blitzcrank retrieves the compressed tuple from storage and decompresses it. The execution engine then consumes the tuple and executes the query without being aware of Blitzcrank.

As shown in Figure 4, Blitzcrank consists of three components: Semantic Learner (SL), Attribute Encoder (AE), and Tuple Encoder (TE). Specifically, the SL determines the compression ordering for the columns using structure learning and generates conditional probability models for the AE. When the tables are small, Blitzcrank leaves them uncompressed. SL is triggered when the size of a table reaches a predefined threshold (default: 2^{16} rows). For compression, the AE takes in a tuple and translates the value of each attribute into an interval in $[0, 1)$ according to the models from SL. The AE then sends the sequence of intervals to the Tuple Encoder which uses delayed coding to produce a compressed record with a near-optimal size. For decompression, the tuple is first decoded into 16-bit codes at TE. The AE then invokes the Inv-Translate (which refers to the probability models) to recover each tuple value.

Semantic Learner approaches the optimal column compression ordering S and a set of models M through a greedy algorithm of structure learning, as described in Section 2.1. To speed up the structure learning on large tables, we perform the algorithm on a set of randomly selected tuples from the table. Once the column ordering S is obtained, the SL further scans the full table to generate accurate conditional probability models $P_x \in M$. P_x is implemented as an unordered map from each value combination of the proceeding attribute models to a probability distribution p_i of attribute x. We refer to p_i as a semantic model. Semantic models can compress values unseen before because they estimate the value distribution rather than statically mapping values to codes in a dictionary. We introduce two fundamental types of semantic models optimized for decompression speed in Section 4.

Attribute Encoder converts each value into an interval and vice versa according to the semantic models. Translate maps an attribute value v to an interval $[l, r)$, $0 \le l < r \le 1$ (symbol-to-interval), while Inv-Translate takes in a code $s \in [l, r)$ and recovers the attribute value v (code-to-symbol). Inv-Translate is critical to the decompression performance. Classic arithmetic coding performs a binary search to determine the matching interval $[l, r)$ for a code s with a time complexity of $O(\log N)$ where N is the number of unique values in a column [38]. We optimize this procedure to constant time in Blitzcrank, as detailed in Section 4.1.

Tuple Encoder receives a sequence of intervals representing each value within a tuple and compresses them into a block of 16-bit integers using delayed coding. At a high level, delayed coding uses a 16-bit unsigned integer s_{int} to encode each interval $[l, r)$ such that $s_{\text{int}}/2^{16} \in [l, r)$. These integers are selected judiciously so that some integer codes are stored implicitly using the redundant information of the other integer codes. In this way, delayed coding not only supports fast decoding (because of the fixed-length codes) but also achieves near-entropy compression. We introduce delayed coding in detail in Section 5.

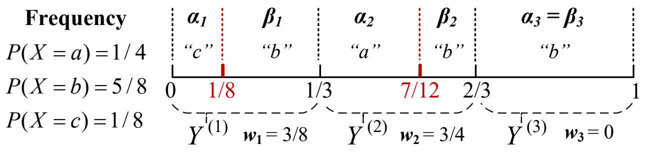


Figure 5: Interval Allocation By Pairing Symbols - There are three interval pairs $\{Y^{(1)}, Y^{(2)}, Y^{(3)}\}$. In each pair, two symbols $\{(\alpha_N, \beta_N)\}$ and the symbol boundary $\{w_N\}$ are saved.

4 SEMANTIC MODELS

A semantic model maps a value to an interval based on the estimated (conditional) probability distribution of the values within a column. In this section, we first introduce two fundamental models for discrete/categorical columns and continuous/numeric columns, respectively. We then show in Section 4.3 how to construct models for other data types (e.g., string) using the fundamental models.

4.1 Discrete/Categorical Model

As in classic entropy encodings, we construct the semantic model for a discrete/categorical column by counting the frequency of each symbol (i.e., value) and computing their cumulative distribution function (CDF). Each symbol is then mapped to its corresponding probability interval on the CDF. For example, the semantic model for column $\{a, b, b, a, c, b, b, b\}$ is $\{a \leftrightarrow [0, 0.25), b \leftrightarrow [0.25, 0.875), c \leftrightarrow [0.875, 1)\}$. `Inv-Translate` a code (e.g., $(.01)_2$) back to its symbol (e.g., "b") requires a binary search to find the interval that contains the code. The logarithmic complexity slows down the decompression, especially when the number of distinct values of a categorical column is large. We, therefore, propose a constant-time algorithm for the `Inv-Translate` function, inspired by the alias method [31].

Constant-Time Inv-Translate. Let $\pi_1, \pi_2, \cdots, \pi_N$ be the interval length (i.e., probability) of each of the N symbols. Given a code $0 \leq s < 1$, if $\pi_1 = \pi_2 = \cdots = \pi_N$, then s belongs to the $\lfloor s \cdot N \rfloor$th interval. Therefore, the key to achieving constant-time `Inv-Translate` is to "create" a uniform distribution. The core idea of the algorithm is to pair the intervals so that the combined probability of each pair forms a uniform distribution.

Suppose we want to create N pairs, each having a combined interval length of $1/N$. At each iteration of the algorithm, we pair the shortest interval with the longest one in the remaining intervals. When their combined interval length is greater than $1/N$, we split the longer interval in two and put the exceeded part back into the interval collection for further pairing. The algorithm terminates when there is ≤ 1 interval left in the collection. Figure 5 shows an example where the interval for symbol "b" is divided and mapped to three pairs. The following theorem proves the validity of this algorithm for any discrete probability distribution.

Theorem 1. *Every probability vector $\pi_1, \cdots, \pi_N$, can be expressed as an equiprobable mixture of N two-point distributions. That is, there are N pairs of integers $(\alpha_1, \beta_1), \cdots, (\alpha_N, \beta_N)$ and probabilities $w_1, \cdots, w_N$ such that*

$$\pi_i = 1/N \cdot \sum_{j=1}^{N} (w_j \mathbb{1}_{\{\alpha_j = i\}} + (1 - w_j)\mathbb{1}_{\{\beta_j = i\}}) = 1/N \cdot \sum_{j=1}^{N} Y_i^{(j)}.$$

for $1 \leq i \leq N$, where $Y^{(1)}, \cdots, Y^{(N)}$ are two-point distributions.

Algorithm 1: Inv-Translate

```
1 Given $\{(\alpha_N, \beta_N)\}$ and $\{w_N\}$ defined in Theorem 1.
  Function Inv-Translate($s$):
2     $c = s \cdot N$           /* $s \in [0, 1)$                  */
3     $j = \lfloor c \rfloor + 1$       /* Determine the index of $Y$      */
4     $q = c - \lfloor c \rfloor$       /* Get the position in $Y^{(j)}$   */
5     return $(q < w_j)$ ? $\alpha_j : \beta_j$
```

Proof. See Appendix B in our technical report [45]. □

The constant-time `Inv-Translate` function is presented in Algorithm 1. Given the binary mixtures $\{Y^{(j)}\}$ with parameters $\{(\alpha_N, \beta_N)\}$ and $\{w_N\}$ defined in Theorem 1, `Inv-Translate` first computes the index (i.e., $j = \lfloor s \cdot N \rfloor + 1$) of the binary mixture $Y^{(j)}$ that contains the input code s. Then the function determines the code's position within $Y^{(j)}$ and returns the corresponding symbol.

4.2 Continuous/Numeric Model

Previous semantic compression algorithms use a bisection method [25, 54] to compress continuous values such as floating-point numbers. This approach, however, generates many low-entropy intervals, thus affecting the efficiency of the subsequent Tuple Encoding. Blitzcrank, therefore, introduces a novel two-level quantization model for continuous-value columns. This model not only supports arbitrary precision to guarantee lossless compression but also leverages the distribution skew in the column for better compression.

Two-Level Quantization Model. The first-level quantization is based on an equi-width histogram of the values in a column. The goal at this level is to maximize compression by assigning larger intervals (i.e., shorter codes) to more frequent value ranges. Specifically, we divide the values into a predefined T (e.g., $T = 512$) disjoint value ranges, each having a bucket width of $w = (\hat{v}_{\max} - \hat{v}_{\min})/T$, where $\hat{v}_{\max}/\hat{v}_{\min}$ is the estimated (or obtained directly from table statistics) maximum/minimum value of the column. We then obtain the frequency of each bucket by scanning the column once, and we assign each bucket i an interval $[l_i, r_i)$ proportional to its frequency, similar to the categorical model.

To guarantee a lossless compression (i.e., to distinguish between the values within a bucket), we apply a second-level quantization where we divide the value range of the bucket equally into G segments so that the width of each segment is smaller than or equal to the column's required precision p. We set $p = 10^{-7}$ and $p = 10^{-17}$ for the `float` and `double` types, respectively. Besides the bucket's interval $[l_i, r_i)$, each segment j in the bucket is assigned another interval $[l_{\epsilon_j}, r_{\epsilon_j})$ with an equal length of $1/G$. If the user specifies a precision requirement for a `float`/`double` column (e.g., 2 decimal places), we enable lossy compression by adjusting p accordingly (e.g., $p = 10^{-2}$) to achieve better compression.

Given a value $\hat{v}_{\min} \leq v < \hat{v}_{\max}$, the `Translate` function computes its first-level bucket index i and its second-level offset j according to the value-range division because v can be uniquely decomposed as $i \cdot w + j \cdot p \leq v - \hat{v}_{\min} < i \cdot w + (j+1) \cdot p$, where $j \cdot p < w$. v is then converted into two intervals: $[l_i, r_i)$ and $[l_{\epsilon_j}, r_{\epsilon_j})$. In cases where v is an outlier (i.e., $v < \hat{v}_{\min}$ or $v \geq \hat{v}_{\max}$), the algorithm falls

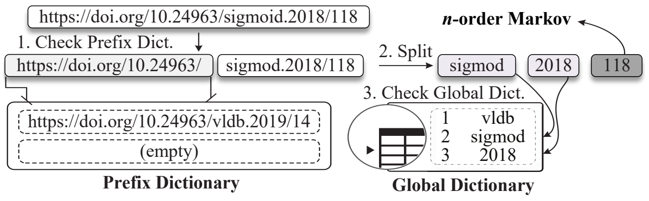


Figure 6: String Model - The URL sample is from the *DBLP*.

back to the slow traditional bisection method. Such a fall back has negligible impact on performance because outliers are usually rare. To recover a value (a value range ≤ the column precision p, to be precise), we invoke the `Inv-Translate` function twice on $[l_i, r_i)$ and $[l_{\epsilon_j}, r_{\epsilon_j})$, respectively.

4.3 Composite Models

Our foundational models can be combined to compress complex attributes. We implement a string model as the example, as shown in Figure 6. It includes a prefix dictionary and a global dictionary. For words not covered by either dictionary, we use a Markov model to encode them letter-by-letter [40].

Efficient Blitzcrank Integration with Intervals. The prefix dictionary compresses strings with similar prefixes and keeps a queue of the latest K (e.g., $K = 4$) strings. Each string is analyzed to find the index i (ranging from 0 to $K - 1$) of a previous string in the queue that shares the longest common prefix, and the count h (an indefinite integer) of identical characters. We use a categorical model and a numeric model to estimate i and h distributions, respectively. Given a new string, the two models output intervals representing i and h. This approach of interval-based representation integrates smoothly with the Blitzcrank Framework.

Adaptive Base Models for Enhanced Compression. Following the prefix dictionary, the remaining substring is split into words using delimiters like spaces and commas. This process involves two models: (1) using a numerical model to count the words in the given substring, and (2) using a categorical model to identify each delimiter. Our model can leverage the skewed distribution of word count and delimiter usage patterns for compression. For example, most sentences have 3-10 words, and the space character is the most frequent delimiter. The final technique is the global dictionary, implemented as a categorical model. It stores words that frequently occur in sentences and is used for dictionary encoding.

Besides the string model, we also design two models: one for encoding JSON collections and another for time-series column encoding, with the latter utilizing the Autoregressive Moving Average (ARMA) [13]. When applied to the data set *Jena Climate* [41], our time-series model achieved a 38% better compression factor than our standard numeric model.

5 DELAYED CODING

The attribute encoder generates a series of intervals, which can be encoded into a bit stream by the arithmetic coding. However, it is slow due to its variable-length codes and extensive floating-point calculations. We propose the delayed coding to address these issues. For ease of understanding, we assume that every symbol can be represented by a single, continuous interval. We relax the constraint in Section 5.6.

Algorithm 2: Inv-Translate with the Integer Probability

```
1 Given $m$, $\{(\alpha_N, \beta_N)\}$, $\{w_N\}$ defined in Theorem 1, where
    $N = 2^m, m \in \mathbb{N}_+$, and $w_i$ is an 16-bit integer, for $i = 1, \cdots, N$.
  Function Inv-Translate($s$):
2     $j \leftarrow s \gg (16 - m)$   /* The higher $m$ bits */
3     $Q \leftarrow s \,\&\, (2^{16-m} - 1)$  /* The lower 16-$m$ bits */
4     return $(Q < (w_j \gg m))$ ? $\alpha_j : \beta_j$
```

5.1 Probability Representation

We begin by investigating fast algorithms to represent and compute with probability intervals. Arithmetic coding is slow due to the many interval product operations required. Recall the example in Section 2.1, when entering a sub-interval of the current interval based on the next symbol's probability, we need to compute a product of intervals. If we use a floating-point number to represent the probability, the interval product ⊠ is defined as:

$$[l_a, r_a) \boxtimes [l_b, r_b) \coloneqq [l_a + (r_a - l_a) \cdot l_b, l_a + (r_a - l_a) \cdot r_b), \quad (1)$$

where $0 \le l_a < r_a \le 1$, $0 \le l_b < r_b \le 1$. However, this design leads to floating-point calculations and the risk of floating-point underflow. An alternative is to use two integers U, d to represent the probability $U/2^d$. In this way, checking for underflow is simply inspecting the exponent of the denominator d.

Integer-based Probability Intervals. In Blitzcrank, a probability is presented as a 16-bit integer U, which logically represents the probability $U/2^{16}$. In this paper, we use $[L, R)$, where $0 \le L, R \le 2^{16}$ are integers, to logically represent the interval $[L/2^{16}, R/2^{16})$. We choose to use 16 bits because using shorter integers increases the decoding overhead while using 32-bit or 64-bit integers must handle integer overflow during multiplication. For an interval whose length is smaller than $1/2^{16}$ (e.g. $[0, 1/2^{32})$), we use the product of two or more intervals to represent it, i.e.

$$[0, 1/2^{32}) = [0, 1/2^{16}) \boxtimes [0, 32768/2^{16}) \longrightarrow [0, 1), [0, 32768).$$

Inv-Translate Becomes Faster. Although the code-to-symbol in Algorithm 1 has constant time complexity, the "floor" operation slows it down. The integer-based probability can make the `Inv-Translate` faster, replacing the "floor" operation with bitwise operations. Algorithm 2 gives the new `Inv-Translate` algorithm with integer-based probability. Note that both the $\{w_N\}$ and the input s in Algorithm 2, using the integer-based probability, are 16-bit unsigned integers. Also, N is always a power of two, a condition that can be satisfied by adding placeholder symbols with a frequency of 0 if necessary.

5.2 Options of Fixed-length Codes

We then introduce an algorithm to extract the code from an interval based on our integer-based probability representation and analyze the bits wasted in terms of code options. In Section 2.1, we explained that for the final interval $[0.22, 0.27)$ in arithmetic coding, using just enough fractional digits to keep the number within this interval is sufficient for encoding. However, this results in variable-length codes, which can slow down decoding due to the inconsistent number of digits across intervals, requiring more branch predictions and checks [47].

An approach to simplify the decoding process is to utilize a fixed number of bits, such as 16 bits, for encoding each interval. In this scenario, adopting integer-based probability, we define the bidirectional mapping of a semantic model as follows:

$$\text{symbol-to-interval} : \{v_1, \cdots, v_N\} \to \{[L_i, R_i), 1 \le i \le N\},$$
$$\text{code-to-symbol} : \{s \in \mathbb{N}, 0 \le s < 2^{16}\} \to \{v_1, \cdots, v_N\},$$

where v_i denotes a unique symbol in this model, allocated with a disjoint interval $[L_i, R_i)$, L_i being the lower bound and R_i being the upper bound of the interval, for $1 \le i \le N$. By converting the interval $[0.22, 0.27)$ to $[14418, 17694)$ using integer-based probability, we can encode it using any 16-bit integer within this range. However, this method requires more than the 6 bits we use in the example of Section 2.1. In fact, for an interval $[L, R)$, the probability of the event it represents is $(R-L)/2^{16}$. Thus, its entropy is $16-\log_2(R-L)$ bits, and we waste $\log_2(R-L)$ bits if using 16 bits to encode it. Notice that we have $R-L$ code options for this interval. The selection of a specific code option itself carries information. Since any of these options can represent the interval $[L, R)$, we can use the options to represent another interval partially.

5.3 Problem Formulation

Leveraging the concept of code options, we formalize the code extraction problem as follows. Given a series of intervals $[L_1, R_1), \cdots, [L_n, R_n)$ with $n > 0$ and $0 \le L_i < R_i \le 2^{16}$, we can select a 16-bit integer $s_i \in [L_i, R_i)$ to encode each interval. Then, can we represent a distinct interval $[L^*, R^*)$ using a sequence $(s_1, \cdots, s_n)$?

Consider a special case where each interval is of length 2 (i.e., $R_i - L_i = 2$), we have two encoding options per interval: either L_i or L_i+1. These options can uniquely represent their respective intervals. Using this binary decision for each interval, we can represent parts of the distinct interval $[L^*, R^*)$. To fully encode $[L^*, R^*)$, we need a sufficient number of intervals. If we have at least 16 intervals, matching the 16 bits of L^*, complete representation is possible. The encoding rule is simple: Let the code $s_i = L_i$ if the i-th bit of L^* is 0; otherwise, s_i takes the value $L_i + 1$. This way, the sequence $(s_1, \cdots, s_n)$ encodes both the series of intervals and the interval $[L^*, R^*)$, leveraging the binary encoding options of each interval.

Consider a general case where each interval's length $k_i = R_i - L_i$ varies within the range $[1, 2^{16})$. This means each interval $[L_i, R_i)$ has k_i coding options: $\{L_i, L_i + 1, \cdots, L_i + (k_i - 1)\}$. Any of these codes can uniquely represent its interval. Moreover, these codes can represent the distinct interval $[L^*, R^*)$ partially as well. In this case, the i-th interval offers a digit in a base k_i system. With n such intervals, they collectively form a mixed radix (base) numeral system [24], with bases $\{k_n\}$. Assuming $\prod_{i=1}^{n} k_i \ge 2^{16}$, the 16-bit number L^* can be converted into this mixed-base system as follows:

$$a_i = L^* \,\%\, k_i, \qquad L^* = L^* \,/\, k_i, \tag{2}$$

for $i = n, \cdots, 1$ in a loop. The resulting value is $a_1 a_2 \cdots a_n$. Setting s_i to $L_i + a_i$ gives a valid code, as $a_i < k_i$ ensures $L_i + a_i < R_i$. Hence, the sequence $(s_1, \cdots, s_n)$ effectively encodes both the series of intervals and the interval $[L^*, R^*)$.

Example: 3-Digit Mixed Radix Numeral System. Given intervals $[1, 4)$, $[2, 6)$, $[3, 10)$, forming a 3-digit mixed radix numeral system with bases $(3, 4, 7)$. This system can encode up to $3\times4\times7 = 84$ distinct states. Assume we use 4 bits to encode each interval. To encode the interval $[13, 14)$, we select $x = 13$ from this range. Decomposing 13 with the bases $(3, 4, 7)$ using Equation (2) gives $13 = \underline{0}\times(4\times7)+\underline{1}\times7+\underline{6}$, leading to the indices $(0, 1, 6)$. To determine s_1, we select the value at index 0 in the first interval $[1, 4)$, resulting in $s_1 = 1$. Similarly, $s_2 = 3$ and $s_3 = 9$. Therefore, the encoded bit stream for the four intervals is $(0001\ 0011\ 1001)_2$.

5.4 Encoding Procedure

Inspired by the mixed radix numeral system introduced above, the encoding procedure of delayed coding is essentially transforming decimal numbers into mixed radix numbers. The encoding of delayed coding has two steps. First, we mark all intervals that can be represented by their former intervals' options. An interval can be marked if and only if the current option number is larger than λ (it takes 2^{16} by default). Second, for each marked interval, we convert its 16-bit code into a mixed-radix number, represented using code options of the intervals that precede it.

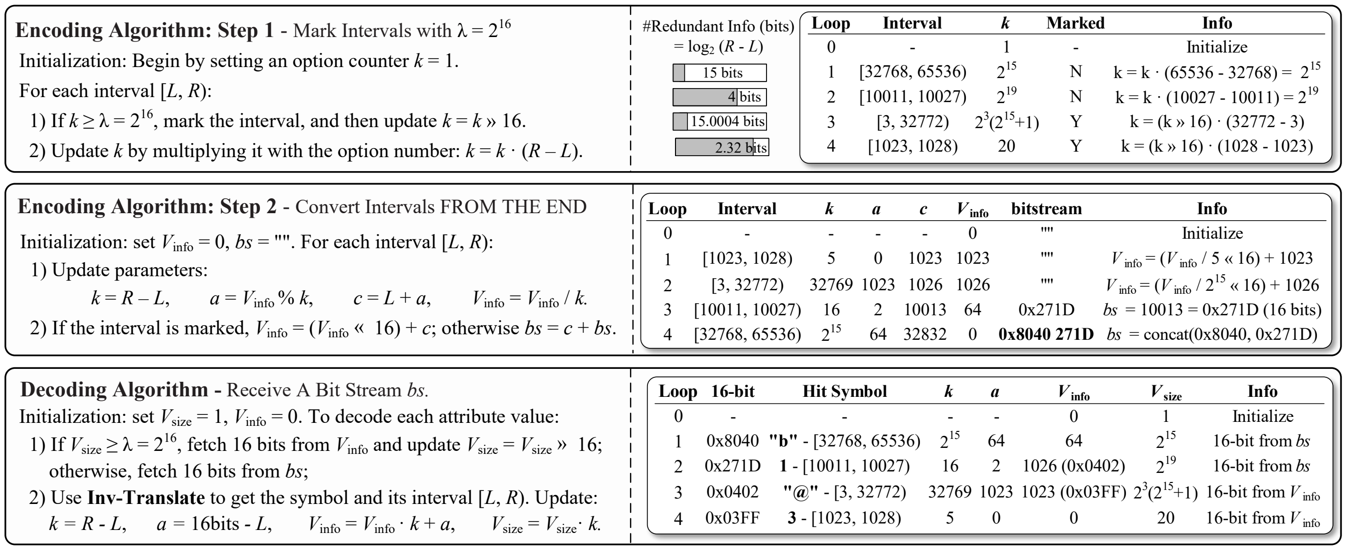
Step 1: Mark Intervals. In Figure 7, we illustrate the encoding process of a tuple ("b", 1,"@", 3) using four categorical models that convert the tuple into intervals. We use an option counter k, initially set to one, to track redundant information. The 1st interval provides 2^{15} options, updating k to 2^{15}. Next, the 2nd interval cannot be marked because the current option counter $k \le \lambda = 2^{16}$. Note that we use fixed-length code (i.e., 16-bit) to encode each interval, and the current numeral system cannot represent a 16-bit integer. The 2nd interval increases k to 2^{19} due to its option of 16. Currently, it is enough to mark the third interval, it consumes 2^{16} options and updates $k = k/2^{16} = 2^3$. The 3rd interval contributes 32769 options, updating $k = 2^3(2^{15} + 1)$. The marking process continues; finally, the last two intervals are marked and will be transformed into mixed radix numbers, represented by their preceding intervals.

Step 2: Convert Intervals From the End. We convert each marked interval into a mixed-radix number in a recursive manner, as illustrated in Figure 8. Specifically, the last two intervals are marked in the marking step. First, we convert the rightmost interval into a mixed-radix number using bases (k_1, k_2, k_3). Then, the last-second interval holds the partial code of the last interval. Since the last-second interval is also marked, it is converted into a mixed-radix number with bases (k_1, k_2). This approach highlights the necessity of processing intervals from the end.

We use a 64-bit integer V_{info} to store the codes for marked intervals temporarily, starting with $V_{\text{info}} = 0$. We use a loop to process each interval $[L, R)$ from the end. For each step, the decimal number to be converted is V_{info}, and the current interval $[L, R)$ provides $k = R - L$ options, as a base-k digit. We compute the digit value a and the left decimal number, using Equation (2):

$$a = V_{\text{info}} \,\%\, k, \qquad V_{\text{info}} = V_{\text{info}} \,/\, k. \tag{3}$$

Therefore, the 16-bit code for the interval $[L, R)$ is computed as $c = L+a$, i.e., we use the code options of it to store a digit value a. For the first processed interval, we get $a = 0$ because $V_{\text{info}} = 0$, but V_{info} can be updated. If the interval $[L, R)$ is marked, $V_{\text{info}} = V_{\text{info}} \cdot k + c$. Otherwise, output the 16-bit code c to the bit stream. The loop continues; finally, we encode the four intervals into 4 bytes.



Encoding Algorithm: Step 1 - Mark Intervals with $\lambda = 2^{16}$

Initialization: Begin by setting an option counter $k = 1$.

For each interval $[L, R)$:

```
  1) If $k \geq \lambda = 2^{16}$, mark the interval, and then update $k = k \gg 16$.
  2) Update $k$ by multiplying it with the option number: $k = k \cdot (R - L)$.
```

#Redundant Info (bits) $= \log_2 (R - L)$

15 bits
4 bits
15.0004 bits
2.32 bits

Loop	Interval	k	Marked	Info
0	-	1	-	Initialize
1	[32768, 65536)	2^{15}	N	$k = k \cdot (65536 - 32768) = 2^{15}$
2	[10011, 10027)	2^{19}	N	$k = k \cdot (10027 - 10011) = 2^{19}$
3	[3, 32772)	$2^3(2^{15}+1)$	Y	$k = (k \gg 16) \cdot (32772 - 3)$
4	[1023, 1028)	20	Y	$k = (k \gg 16) \cdot (1028 - 1023)$

Encoding Algorithm: Step 2 - Convert Intervals FROM THE END

Initialization: set $V_{\text{info}} = 0$, bs = "". For each interval $[L, R)$:

```
  1) Update parameters:
        $k = R - L$,   $a = V_{\text{info}} \% k$,   $c = L + a$,   $V_{\text{info}} = V_{\text{info}} / k$.
  2) If the interval is marked, $V_{\text{info}} = (V_{\text{info}} \ll 16) + c$; otherwise $bs = c + bs$.
```

Loop	Interval	k	a	c	V_{info}	bitstream	Info
0	-	-	-	-	0	""	Initialize
1	[1023, 1028)	5	0	1023	1023	""	$V_{\text{info}} = (V_{\text{info}} / 5 \ll 16) + 1023$
2	[3, 32772)	32769	1023	1026	1026	""	$V_{\text{info}} = (V_{\text{info}} / 2^{15} \ll 16) + 1026$
3	[10011, 10027)	16	2	10013	64	0x271D	bs = 10013 = 0x271D (16 bits)
4	[32768, 65536)	2^{15}	64	32832	0	**0x8040 271D**	bs = concat(0x8040, 0x271D)

Decoding Algorithm - Receive A Bit Stream *bs*.

Initialization: set $V_{\text{size}} = 1$, $V_{\text{info}} = 0$. To decode each attribute value:

```
  1) If $V_{\text{size}} \geq \lambda = 2^{16}$, fetch 16 bits from $V_{\text{info}}$ and update $V_{\text{size}} = V_{\text{size}} \gg 16$;
     otherwise, fetch 16 bits from $bs$;
  2) Use **Inv-Translate** to get the symbol and its interval $[L, R)$. Update:
     $k = R - L$,   $a$ = 16bits - $L$,   $V_{\text{info}} = V_{\text{info}} \cdot k + a$,   $V_{\text{size}} = V_{\text{size}} \cdot k$.
```

Loop	16-bit	Hit Symbol	k	a	V_{info}	V_{size}	Info
0	-	-	-	-	0	1	Initialize
1	0x8040	**"b"** - [32768, 65536)	2^{15}	64	64	2^{15}	16-bit from *bs*
2	0x271D	**1** - [10011, 10027)	16	2	1026 (0x0402)	2^{19}	16-bit from *bs*
3	0x0402	**"@"** - [3, 32772)	32769	1023	1023 (0x03FF)	$2^3(2^{15}+1)$	16-bit from V_{info}
4	0x03FF	**3** - [1023, 1028)	5	0	0	20	16-bit from V_{info}

Figure 7: Delayed Coding - Given a tuple ("b", 1, "@", 3), the attribute encoder translates it into intervals: [32768, 65536), [10011, 10027), [3, 32772), [1023, 1028). These intervals are then encoded and decoded as shown above.

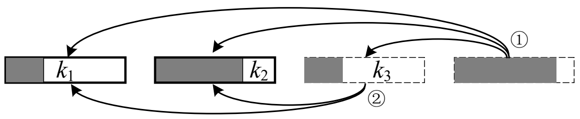


Figure 8: Recursive Encoding - First, the last interval is encoded by the numeral system with base (k_1, k_2, k_3). Then, the last-second interval is encoded by the numeral system with base (k_1, k_2).

5.5 Decoding Procedure

Conversely, the decoding procedure transforms the mixed radix numerals back into decimal numbers. There are two sources of bits to decode a tuple: the bit stream or the virtual input V_{info}. The decoding of each symbol has three steps: (1) retrieve a 16-bit code from V_{info} if the current option number V_{size} is larger than 2^{16}; otherwise from the bit stream; (2) obtain the desired symbol and its interval $[L, R)$ by calling the `Inv-Translate` function; (3) Update V_{info} and V_{size} accordingly.

The bottom part of Figure 7 shows the decoding process. We want to decode a tuple from the bit stream `0x8040 271D`, at first, $V_{\text{info}} = 0$, and $V_{\text{size}} = 1$. For the first attribute value, We fetch 16 bits from the bit stream, getting `0x8040`. The function `Inv-Translate` receives it and returns the symbol "b". We use the symbol-to-interval mapping to determine its interval, resulting in $[L, R) = [2^{15}, 2^{16})$. Next, we compute the digit base $k = R - L = 2^{15}$, and the digit number a = 16bits $- L = 64$. Using them, we update V_{info}, and V_{size}:

$$V_{\text{info}} = k \cdot V_{\text{info}} + a, \qquad V_{\text{size}} = k \cdot V_{\text{info}}.$$

This is just the inverse formula of Equation (3). It is necessary to use V_{size} to record the amount of information in V_{info}. For instance, with $V_{\text{info}} = 1$, a V_{size} of 4 results in two virtual bits 01_2, while a V_{size} of 8 yields three virtual bits 001_2. The second interval decodes to the symbol "1". When $V_{\text{size}} = 2^{19}$ ($\geq \lambda = 2^{16}$), we fetch the next 16-bit code from the virtual input, resulting in `0x0402` for the third symbol, as shown in the decoding table Loop 3 of Figure 7. This decoding process repeats for all symbols, getting ("b", 1, "@", "3").

5.6 Modification for Non-Continuous Intervals

Up to this point, we assume that each symbol is represented by a single continuous interval. In this section, we modify our algorithm by relaxing this constraint to allow a symbol to be represented by the union of multiple non-continuous intervals. A non-continuous interval example is shown in Figure 5, where the symbol "*b*" is assigned to the interval $[1/8, 1/3) \cup [7/12, 1)$, or its integer representation $[8192, 21845) \cup [38229, 65536)$.

Non-continuous intervals, offering the same number of options as continuous ones but with different option positions, require slight modifications in delayed coding. Take a non-continuous interval with two segments $[L^{(1)}, R^{(1)}) \cup [L^{(2)}, R^{(2)})$. It provides $k = (R^{(1)} - L^{(1)}) + (R^{(2)} - L^{(2)})$ options. To store a number $a \in [0, k]$ using these options, we modify the selection of the 16-bit code c as follows:

$$c = \begin{cases} L^{(1)} + a & \text{if } 0 \leq a \leq R^{(1)} - L^{(1)}, \\ L^{(2)} + a - (R^{(1)} - L^{(1)}) & \text{if } R^{(1)} - L^{(1)} \leq a \leq k. \end{cases}$$

In other words, we choose the a-th optional code of the symbol, regardless of its integer value. Decoding involves the reverse process. For a 16-bit code within this non-continuous interval, we retrieve the stored number a as follows: if 16bits $\in [L^{(1)}, R^{(1)})$, then a = 16bits $- L^{(1)}$. Otherwise, a = 16bits $- L^{(2)} + (R^{(1)} - L^{(1)})$. In other words, the 16-bit code is the a-th item in this non-continuous interval. This method can be extended to manage intervals with more than two segments, generating two piecewise linear functions for the computation of c and a. Importantly, this modification has no effect on the correctness and efficiency of delayed coding, as shown in our technical report [45].

5.7 Fine Granularity Compression Effectiveness

We show the effectiveness and optimality of delayed coding in this section. In Figure 7, there are 20 unused options (i.e., $k = 20$) after the encoding, resulting in a waste of $\log_2 20 = 4.32$ bits. The number of wasted bits can be bounded by $\log_2 \lambda$ (note that we mark

Table 1: Data sets - Unmarked data sets come from Public BI Benchmark [53] or earlier semantic compression research [25, 28].

Group	Data sets	#Rows	#Cols	Row Length
Numeric	Corel	68,040	93	820 byte
	Jena Climate [41]	420,551	14	138 byte
	Cars	344,287	155	393 byte
Categorical	Forest Cover	581,012	55	127 byte
	US Census 1990	2,458,285	69	145 byte
	Food	5,216,593	5	22 byte
	Bimbo	20,259,279	12	54 byte
String	Yale Languages	5,762,082	30	284 byte
	Medicare	8,645,072	26	229 byte
	Arade	9,888,775	11	88 byte

an interval once the number of options is larger than λ). Theorem 2 shows that as the number of intervals grows, the effectiveness of delayed coding improves, approaching the entropy.

Theorem 2. *Give a series of intervals* $[L_1, R_1), \cdots, [L_n, R_n)$, *where* $n \geq 1$, *and* L_i, R_i *are 16-bit integers less than or equal to* 2^{16} *for all* i. *Suppose delayed coding:*

(1) *Marks an interval if and only if the current option number is larger or equal to* λ, *where* $\lambda \geq 2^{16}$.
(2) *Encodes every* ζ *intervals as a bit stream, where* $0 < \zeta \leq n$.

Thus, the number of used bits $L^{n,\lambda,\zeta}$ *is bounded by*

$$L^{n,\lambda,\zeta} \leq n \cdot C + (\lfloor n/\zeta \rfloor + 1) \cdot \log_2 \lambda + n \cdot \log_2 (1 - 65535/\lambda)^{-1},$$

where $n \cdot C$ *is the entropy of all intervals. Further, for a sufficiently large* n, *by setting* $\lambda = \zeta = n$, *we have* $L^{n,\lambda,\zeta}/(n \cdot C) \longrightarrow 1$ *as* $n \longrightarrow +\infty$.

Proof. See Appendix D.2 in our technical report [45]. □

Summary: Delayed coding uses a fixed number of bits to encode each interval. It is based on the insight that altering the redundant information in an interval does not affect its symbol retrieval. Theorem 2 reveals that delayed coding has a near-entropy compression factor with fine compression granularity.

6 COMPRESSION MICROBENCHMARKS

We evaluate Blitzcrank in the next two sections. First, using 10 real tables, we compare Blitzcrank with modern compressors. This comparison focuses on compression factors and fast random tuple access from compressed storage (Section 6.1). Then, we provide a breakdown of the Blitzcrank structure learner (Section 6.2). Following this, we compare delayed coding with asymmetric numeral systems (Section 6.3). Finally, we optimize the random access performance by analyzing the compression block size (Section 6.4).

Baselines. We evaluate Blitzcrank against Zstandard [17] and Raman's approach [46]: (1) Zstandard is a real-time compression system. It has a training mode, designed for compressing many small files. This mode creates a "zstd-dictionary" from all files and uses it to compress each file independently. We use the open-source Zstandard (v1.5.1) in C++, setting the "zstd-dictionary" capacity to the recommended 110 KB and using the default compression level. (2) Raman's method [46] focuses on tuple compression. It considers correlations between columns and combines Huffman coding and delta encoding to achieve a high compression factor. We implemented Raman's approach in C++ using the default column ordering of an input table.

We exclude DeepSqueeze [28] because it does not support high cardinality columns and is not open-sourced. We do not include FSST [12] and other lightweight techniques [7, 8, 20], because they are not for row-stores. In our technical report [45], we also evaluate Blitzcrank against the open-sourced (in C++) Squish [25] and Gzip [62] for the table archive task. Our method is 20× faster than Squish and offers 2× higher compression factors compared to Gzip.

Blitzcrank Setting. Blitzcrank samples 2^{15} tuples for structure learning, with detailed sensitivity analysis provided in Section 6.2. For delayed coding, each tuple is individually encoded for the optimal access latency. We set $\lambda = 2^{16}$ to maximize the compression factor, as detailed in Theorem 2. We evaluate two Blitzcrank variants: one utilizes column correlation for compression (Blitzcrank w/ Correlation), while the other does not (Blitzcrank w/o Correlation).

Data Sets. Table 1 shows the data sets. Besides the data sets from previous studies [25, 28, 53], we also use *Jena Climate*, which consists of 14 time-series columns [41]. We classify each data set into categorical, numeric, or string types. Specifically, we calculate the proportion of each attribute type in the total data set size and select the type with the highest proportion as the representative group for that data set.

Experimental Setup. We use three metrics to measure compression performance: compression factor, throughput, and random access latency. The throughput represents the amount of data processed per unit of time for a given compression/decompression task. Random access latency is the time required to retrieve a random record. We conduct our experiments on a machine equipped with two Intel® Xeon 8375C (32 × 2 cores) and 512 GB RAM. The disk we use is an Intel® SSD D5-P5530 (1 TB). We use Debian GNU/Linux 11 and GCC 10.2 with `-O3` enabled. All microbenchmarks are conducted with a single thread.

6.1 Compression Evaluation

We evaluate Blitzcrank on in-memory tables (constructed using the data sets above) with each tuple compressed separately. The compressed tuples are organized using a primary-key index (implemented using a simple C++ vector) where the primary keys are monotonically increasing integers. We use YCSB (workload C) with a Zipf distribution to generate the random-access workloads [18]. Specifically, for each data set, we first compress and insert 5 million tuples into the in-memory table and then execute 1 million point queries, each involving decompressing a particular tuple. We report the average latencies for compression-insertion and random access separately. We also record the size of each in-memory table after insertion to calculate the compression factor. For each compressor, we first train its model over the corresponding data set if required by the algorithm.

Figure 9 shows the results, including compression factor, latency, and training time, across various data sets on the x-axis. Blitzcrank has the highest compression factor for 7/10 tables, and offers the lowest latency for 9/10 tables among all compressors. This is because Blitzcrank models columns in a semantic way and uses fixed-length code for encoding. Raman's approach has the

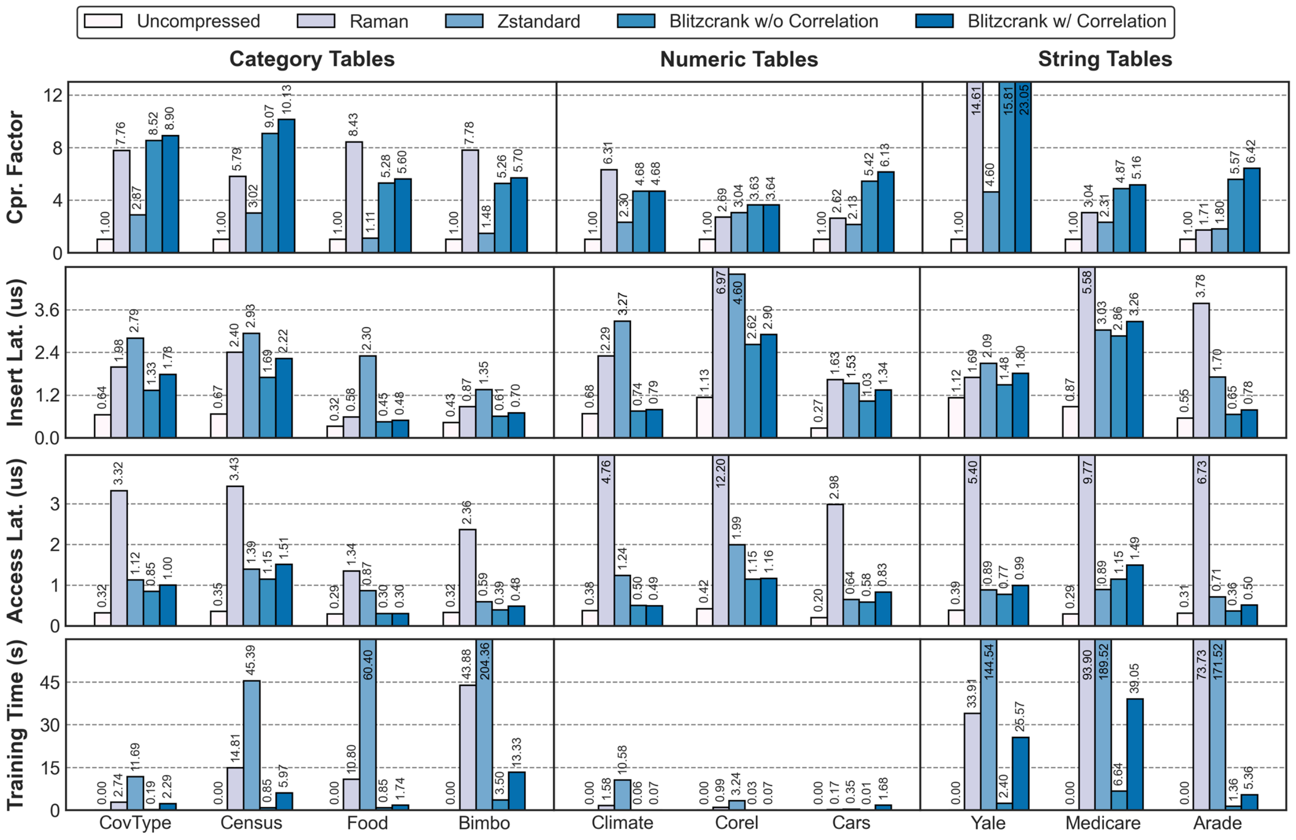


Figure 9: Compression Evaluation - We report the compression factor, insert/access latency, and training time of all compressors.

highest compression factor for the remaining 3/10 tables because tuples of these tables have low entropy; each tuple requires on average just a few bits for encoding (e.g., 2.6 bytes for a *Bimbo*'s tuple). In this case, using fixed-length codes is less efficient. However, Raman's approach is slow for accessing tuples, because its variable-length code for each attribute leads to additional checks when decoding. Zstandard falls short for both the compression factor and the latency, because it relies on long contexts, at least 4KB, to build an effective dictionary. However, the length of a single tuple is insufficient to meet this 4 KB requirement.

We advise using Blitzcrank w/o Correlation in most cases. Capturing column correlations improves the compression factor in the sacrifice of the access latency and the model training time. The training time is often exponential to the number of categorical columns, but a longer training time does not necessarily guarantee better performance. A detailed analysis of the semantic learner is given in Section 6.2.

6.2 Sensitivity to Sampling Number

Blitzcrank randomly selects a subset of samples for structure learning. We now investigate the sensitivity to the sampling number. We use Blitzcrank w/ Correlation to compress and decompress the whole data sets with the compression granularity being a single tuple (i.e., delayed coding encodes each tuple into a separate compressed block). We keep this tuple-level compression granularity or the remaining experiments unless specified otherwise. We select two representative data sets *Bimbo* and *Census* for the analysis. Because the structure learning influences the complexity of the model generated, which further affects the compression speed, we report the duration of each stage within Blitzcrank: structure learning (Structuring), model generation (Generation), compression, and decompression.

Figure 10 shows the results. We vary the #samples in structure learning on the x-axis (log-scaled) and record the compression factor and the running time of each stage. For *Bimbo* in Figure 10a, the sample number has little effect on the compression factor and running time – the learner cannot learn many dependencies. Structure learning time increases slightly with sample number; this is expected since more samples need scanning. *Census* in Figure 10b shows a different pattern: the compression factor increases with the #samples – the learner finds interesting dependencies and generates more complex models. Therefore, we need more time to generate models. The disparity between the two patterns is due to different column counts: *Census* has 69 columns compared to *Bimbo*'s 12. More columns typically indicate more complex dependencies, leading to higher access latency and longer training time. Considering the running time and performance, we set the default sample number to 2^{15} for Blitzcrank. R4.W3, R4.D5Although considering column correlation may improve the compression factor of Blitzcrank on a few data sets (e.g., *Yale*) with a small impact on the access latency, we opt to use Blitzcrank w/o Correlation for the remaining experiments unless otherwise specified.

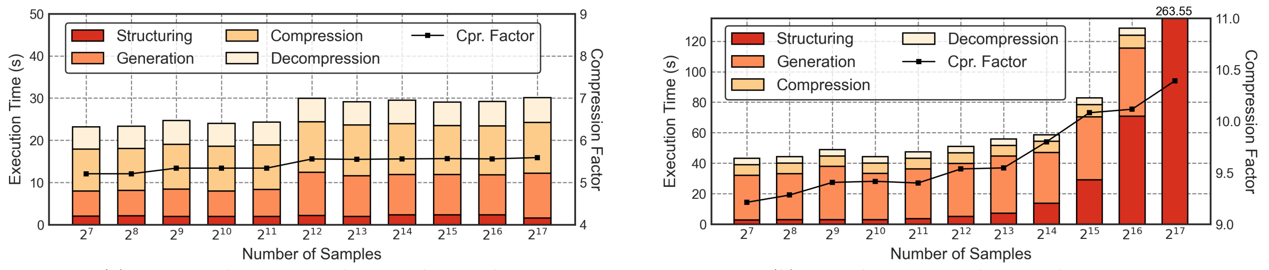

(a) No Correlation Found Example: *Bimbo* **(b)** Correlation Found Example: *Census*

Figure 10: Breakdown of Blitzcrank Distribution Learner - Vary the #samples and evaluate the performance.

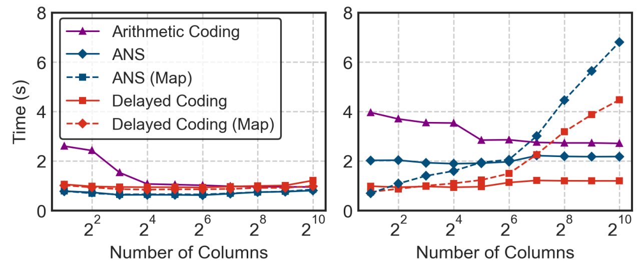

(a) Compression **(b)** Decompression

Figure 11: Entropy Coding Running Time - Vary the #column of tables and record the processing time.

6.3 Entropy Coding Running Time

We evaluate the delayed coding with arithmetic coding and asymmetric numeral systems (ANS) [22]. We implement arithmetic coding with the integer-based probability representation; and use finite-state entropy [16] as the implementation of the ANS. We integrate them into Blitzcrank and use the same models for distribution estimation. In this experiment, we create 64 MB relational data sets with different numbers of columns. Each column has a uniform distribution of cardinality 255 with values sampled from ASCII codes. We vary the column number from 2 to 1024 to record the compression and decompression times of all algorithms.

In Figure 11, we compare delayed coding, arithmetic coding, and ANS, represented by the solid lines. Delayed coding is 2× faster than ANS for the decompression speed, with arithmetic coding being the slowest. This is because delayed coding has a constant-time decoding complexity. Arithmetic coding, on the other hand, relies on binary search for code-to-symbol mapping, operating in $O(\log N)$. An improvement for ANS involves using an unordered map from codes to symbols to accelerate decoding [22]. We then implement such a decoding map for both ANS and delayed coding, denoted by the dotted lines. Figure 11 shows that the delayed coding is still faster than ANS. Decompression time for ANS is similar to delayed coding with few columns but slows as column numbers increase due to cache burden. Storing decoding maps increases the cache miss rate from 0.04% to 0.132% when columns exceed 108.

6.4 Sensitivity to Compression Granularity

We claim that delayed coding has near-entropy performance with fine compression granularity in Section 5.7. In this part, we investigate the effect of compression block size in practice. We present three data sets for this experiment: *Arade*, *Cover Type*, and *Yale Language*. We omit other data sets because they produce similar results. In each trial, we vary the block size (#tuple) to see how they affect the compression factor and the random access latency. We measure the access latency by repeating the process one million times. This experiment is conducted in memory.

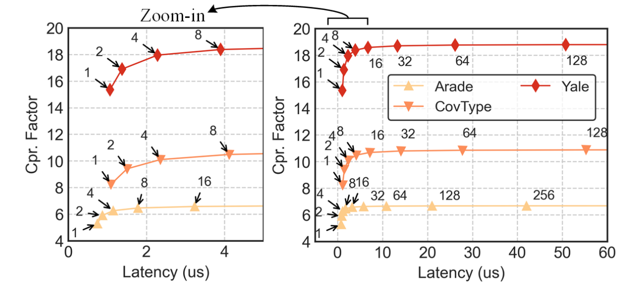

(a) Zoom-in **(b)** Latency vs. Cpr. Factor

Figure 12: Compression Granularity - Vary the block size from 1 to 128 tuples, and write it next to the marker for each trial.

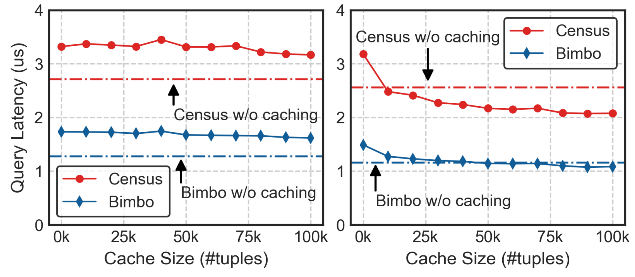

(a) Uniform Distribution **(b)** Zipfian Distribution

Figure 13: Effect of a Fast-Path LRU Cache - The workloads are based on YCSB Workload F (read-modify-write). Dashed lines are Blitzcrank without caching, while the solid lines are Blitzcrank with fast-path cache enabled.

Figure 12 shows that the delayed coding has a high compression factor with fine compression granularity. For each table, the compression factor reaches a plateau when the compression block size exceeds 8 tuples. This indicates a trade-off between compression factor and latency within the 0 to 8 tuple range, allowing the users to select their preferred block sizes. Since OLTP databases usually prioritize low latency and Blitzcrank has a high compression factor even at a block size of one tuple, we set this as the default.

6.5 Fast Path for Tuple Updates

We evaluate a fast path for tuple updates in this section. Specifically, we implemented an LRU write-back cache to buffer the most

Table 2: Data Generation Methods - Addresses are generated by ZIP code conventions [5]; Phone and district are produced by populating a predefined format with random numbers.

Column	Method	Source/Format
C_FIRST	Sampling	US Baby Names [4]
C_STREET	Sampling	Open Addresses [3]
C_DATA	Sampling	City Max Capita [53]
S_DATA	Sampling	Corporations [53]
C_STATE	Sampling	List of US States
C_CITY	Conditional	Cities within C_STATE
C_ZIP	Conditional	ZIP Codes within C_CITY
C_PHONE	Format-Based	"(XXX) XXX-XXXX"
S_DIST	Format-Based	"dist-str#XX#XX#XXXX"

recently accessed tuples in their decompressed form. Normally, a tuple update involves loading the compressed tuple, decompressing it, modifying the tuple, and re-compressing the updated tuple. With the cache, the workload flow starts by looking up the cache. If the target (decompressed) tuple is already in the cache, we modify the tuple directly. Otherwise, we first decompress the target tuple and insert it into the cache. We evaluate the cache using YCSB workload F (read-modify-write) under Uniform and Zipfian query distributions on data sets *Census* and *Bimbo*. The table initially contains five million tuples, and we execute one million read-modify-write queries on the table.

As shown in Figure 13, adding the fast-path cache slows down uniformly distributed queries because cache hits are rare, and cache lookup and maintenance bring overhead. For Zipf-distributed queries, having the fast-path cache improves the query performance because as the cache size increases, more and more tuple updates are performed directly in the cache without decompression. We conclude that a fast-path cache can benefit skewed workloads when using Blitzcrank for compression.

7 SYSTEM EVALUATION

We integrated Blitzcrank (w/o Correlation) into Silo [51] and measured the end-to-end performance using the TPC-C benchmark [19]. Silo is an OCC-based serializable database designed for excellent performance at scale on large multi-core machines. Silo uses the Masstree[39] for its underlying indexes, and has a very high transaction throughput, achieving more than 1 million txns/s on the standard TPC-C workload in our experiments.

Each record in a table is compressed separately. The compressors under test are Uncompressed, Zstandard, Raman, and Blitzcrank with the corresponding row-stores named Silo, ZstdDB, RamanDB, and BlitzDB, respectively. To access a record by primary key, Silo walks the index tree using that key to find the compressed record and then decompresses it into a list of attributes.

According to the TPC-C specification, some columns are filled with random bytes which are incompressible. We substitute these bytes with data that either follows real-world patterns or is sampled from the collected corpus. Table 2 details our data generation approach. The compression factors for the new Customer and Stock tables are 3.44 and 5.57, respectively.

7.1 In-Memory Workloads

In this section, we investigate the performance-space trade-offs of BlitzDB compared to the other baselines when the entire database fits in memory. We vary the number of warehouses in TPC-C from 64 to 896, in increments of 64. In each trial, we use 16 threads. Each database executes 16 million transactions and presents the average throughput results. The training time is measured before the transactions start, while the database size and model size are measured after the transactions. RamanDB uses the entire data set for training, while Blitzcrank and ZstdDB sample data from 16 warehouses. Because Raman's approach uses a static dictionary, it cannot compress new records. We, therefore, use a buffer (size = 64K tuples) to batch the newly inserted and updated records temporarily. When the buffer is full, we create a new dictionary to compress these buffered records. Before adding or using these dictionaries, each thread secures a mutex lock.

As shown in Figure 14, Blitzcrank compresses the data to 14.8% of the original (i.e., Silo) with a throughput decrease of around 21% due to the compression overhead. Such a performance-space trade-off is much more optimized compared to ZstdDB and RamanDB. Moreover, Figure 15 shows that Blitzcrank has the smallest model size and requires orders-of-magnitude shorter time for training compared to the baselines.

Figure 17 shows the TPC-C throughput as the number of threads grows, with each thread corresponding to a warehouse. Both BlitzDB and Silo demonstrate impressive thread scalability. However, throughput reaches a clear limit after 64 threads, which is particularly noticeable in Raman's approach. This limit can be ascribed to several factors: hyperthreading, the increasing size of the database, shared resources such as the L3 cache, and direct thread contention.

7.2 Larger-Than-Memory Workloads

We then evaluate Blitzcrank under the case when the working set does not fit in physical memory. The tuples are stored on disk with the memory acting as a cache. The memory (i.e., the buffer pool) adopts an LRU replacement policy, and we set the memory limit to 5 GB (excluding the memory occupied by indexes). We start with 16 warehouses (around 1 GB) and execute TPC-C transactions for 20 minutes using 16 threads.

Figure 16 shows the throughput and memory consumption for the experiments. Note that the x-axis represents the number of executed transactions as did in [56]. After 20 minutes of execution, BlitzDB completed 5× as many transactions as Silo. This is because Blitzcrank not only achieves an exceptional compression factor but also brings moderate compression/decompression overhead to the system. BlitzDB can sustain at high throughput for a longer time because the memory saved by Blitzcrank allows the database to keep a larger working set in memory.

As a comparison, neither ZstdDB nor RamanDB significantly improves transaction execution. Zstandard suffers from a low compression factor, especially on short tuples. For example, despite using the zstd-dictionary, Zstandard only achieves a compression factor of around 1.3 for the TPC-C table OrderLine. On the other hand, Raman's method is limited by its large dictionary size. It uses a buffer to temporarily hold tuples, compressing them once the buffer is full and then clearing them. This leads to the generation

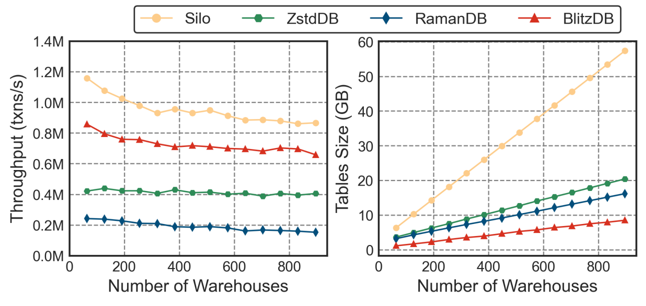

(a) Throughput **(b)** DB Size

Figure 14: TPC-C Workload - In each trial, we use 16 threads and each thread executes 1 million transactions.

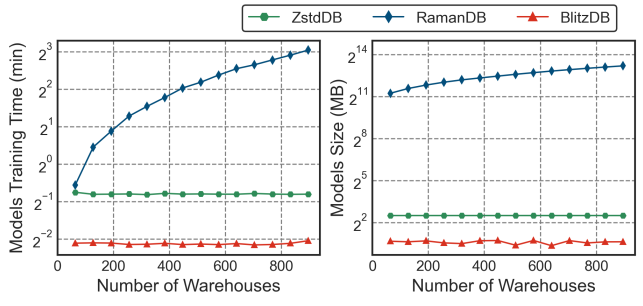

(a) Training Time **(b)** Models Size

Figure 15: Models in TPC-C - RamanDB uses full data for training, and the others only sample 16 warehouses for training.

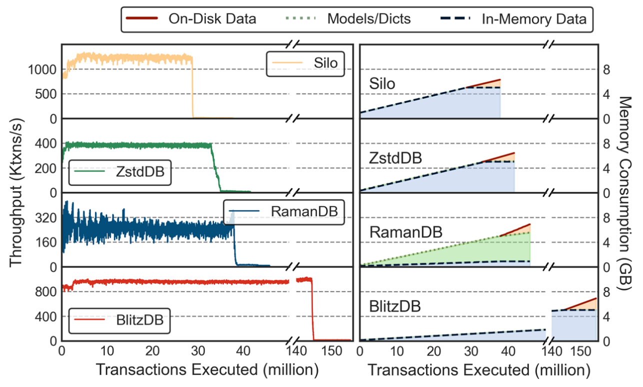

(a) Throughput **(b)** Memory Consumption

Figure 16: TPC-C Large-Than-Memory Workload - Start with 16 warehouses, and each trial runs 20 minutes using 16 threads.

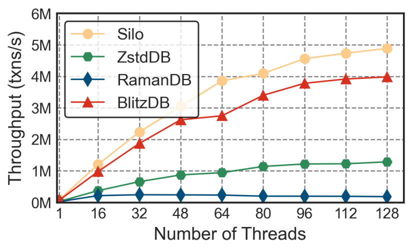


Figure 17: Scalability of Compression - Vary the thread number from 1 to 128, and each trail runs 1 minute.

of large compression dictionaries and unstable throughput, with a notable decrease in speed during compressing the buffer tuples.

8 RELATED WORK

Lightweight encoding, such as bit-packed, delta, run-length, dictionary, and bit vector encoding, is popular recently [1, 7–9, 12, 14, 37, 48]. These are used in column-store databases as they can quickly process large chunks of data using the SIMD technique [29]. However, delta and run-length encoding methods can be slow when we need to quickly grab just a single tuple/value, as they have to decode an entire data block. Therefore, these lightweight encoding methods are unsuitable for processing data in OLTP databases.

General-purpose block compression methods such as Gzip[62], Snappy [2], and Zstandard [17] effectively save disk space by using a sliding window technique to identify word repetitions, thus minimizing data transfer between disk and memory [1]. However, the static dictionary makes them *inflexible* for insert/update scenarios. Zstandard provides a special mode for small files. It improves compression by training on all small file, generating a dictionary. This dictionary is required to be loaded before compression and decompression. However, this mode is less effective at compressing files slightly different from the prior data.

Semantic compression needs to estimate probability distributions for each column in a relational table. Babu et al. proposed the first lossy semantic compression method, SPARTAN, for table compression [10]. Subsequently, Gao et al. introduced Squish, which uses a Bayesian network and arithmetic coding [25]. Later, DeepSqueeze was conceived, using auto-encoders [28]. However, being lossy, these techniques are suited for data archiving and less so for low-latency transaction processing. For example, Squish sorts each table column to make delta encoding more efficient, but slowing compression. DeepSqueeze does not support columns with high cardinality*. In contrast, BLITZCRANK supports all common column types in databases and provides a faster compression speed. Also, DeepSqueeze uses deep learning techniques [60] for structure learning. However, this approach lacks explainability and has a slow inference speed. BLITZCRANK, therefore, uses the Bayesian network to capture column correlations for its simplicity. The effectiveness of the Bayesian network approach has been proved in [25].

9 CONCLUSIONS

We introduce BLITZCRANK, a high-speed semantic compressor for OLTP databases. We first propose novel semantic models that support fast inferences and dynamic value sets for both discrete and continuous data types; we then introduce a new entropy encoding algorithm, called delayed coding, that achieves significant improvement in the decoding speed compared to modern arithmetic coding implementations. BLITZCRANK has high compression factors and fast decompression speed. We integrate BLITZCRANK into an in-memory OLTP database, Silo. The TPC-C benchmark shows that, for data sets larger than the available physical memory, BLITZCRANK can help the database sustain a high throughput and execute four times more transactions before the I/O overhead dominates.

*DeepSqueeze uses one-hot encoding for each column in its network architecture, and high cardinality introduces numerous parameters in the fully connected layer [28].

REFERENCES

[1] 2013. parquet. https://parquet.apache.org/

[2] 2019. Snappy. https://github.com/google/snappy

[3] 2023. The free and open global address collection. https://openaddresses.io/

[4] 2023. Popularity of Names by US State from the Social Security Website. https://www.ssa.gov/oact/babynames/limits.html

[5] 2023. US Zip Codes Database. https://simplemaps.com/data/us-zips

[6] 2023. Zstandard - Real-time data compression algorithm. http://facebook.github.io/zstd/

[7] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* (2013), 197–280.

[8] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of SIGMOD'06*. 671–682.

[9] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of SIGMOD'16*. ACM, 671–682.

[10] Shivnath Babu, Minos N. Garofalakis, and Rajeev Rastogi. 2001. SPARTAN: A Model-Based Semantic Compression System for Massive Data Tables. In *Proceedings of SIGMOD'01*. ACM, 283–294.

[11] Bruno Barbarioli, Gabriel Mersy, Stavros Sintos, and Sanjay Krishnan. 2023. Hierarchical Residual Encoding for Multiresolution Time Series Compression. *Proceedings of SIGMOD'23* (2023), 1–26.

[12] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of VLDB'20* (2020), 2649–2661.

[13] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. 2015. *Time series analysis: forecasting and control*. John Wiley & Sons.

[14] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. 2001. Query optimization in compressed database systems. In *Proceedings of SIGMOD'01*. 271–282.

[15] John G. Cleary and Ian H. Witten. 1984. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Trans. Commun.* 32, 4 (1984), 396–402.

[16] Yann Collet. 2022. Finite State Entropy. https://github.com/Cyan4973/FiniteStateEntropy

[17] Yann Collet and Murray S. Kucherawy. 2021. Zstandard Compression and the "application/zstd" Media Type. *RFC* 8878 (2021), 1–45.

[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of SOCC'10*. 143–154.

[19] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0). https://www.tpc.org/tpcc/

[20] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses).. In *Proceedings of EDBT'17*. 72–83.

[21] Scott Davies and Andrew W. Moore. 1999. Bayesian Networks for Lossless Dataset Compression. In *Proceedings of SIGKDD'99*. ACM, 387–391.

[22] Jarek Duda. 2021. Encoding of probability distributions for Asymmetric Numeral Systems. *CoRR* abs/2106.06438 (2021).

[23] Yannis Foufoulas, Lefteris Sidirourgos, Elefterios Stamatogiannakis, and Yannis E. Ioannidis. 2021. Adaptive Compression for Fast Scans on String Columns. In *Proceedings of SIGMOD'21*. ACM, 554–562.

[24] Aviezri S Fraenkel. 1985. Systems of numeration. *The American Mathematical Monthly* 92, 2 (1985), 105–114.

[25] Yihan Gao and Aditya G. Parameswaran. 2016. Squish: Near-Optimal Compression for Archival of Relational Datasets. In *Proceedings of SIGKDD'16*. ACM, 1575–1584.

[26] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS.. In *CIDR*.

[27] Christoph Heger, André van Hoorn, Mario Mann, and Dusan Okanovic. 2017. Application Performance Management: State of the Art and Challenges for the Future. In *Proceedings of ICPE'17*. ACM, 429–432.

[28] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Çetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proceedings of SIGMOD'20*. ACM, 1733–1746.

[29] Hao Jiang and Aaron J Elmore. 2018. Boosting data filtering on columnar encoding with simd. In *Workshop on Data Management on New Hardware, Proceedings of SIGMOD'18*. 1–10.

[30] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press.

[31] Richard A Kronmal and Arthur V Peterson Jr. 1979. On the alias method for generating random variables from a discrete distribution. *The American Statistician* 33, 4 (1979), 214–218.

[32] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proceedings of SIGMOD'23* (2023), 1–26.

[33] Glen G Langdon. 1984. An introduction to arithmetic coding. *IBM Journal of Research and Development* 28, 2 (1984), 135–149.

[34] 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. *Proceedings of VLDB'20* 29, 6 (2020), 1263–1285.

[35] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling Low Tail Latency on Multicore Key-Value Stores. *Proceedings of VLDB'20* 13, 7 (2020), 1091–1104.

[36] Jiguo Li, Chuanmin Jia, Xinfeng Zhang, Siwei Ma, and Wen Gao. 2021. Cross Modal Compression: Towards Human-comprehensible Semantic Compression. In *Proceedings of ACM MM'21*. ACM, 4230–4238.

[37] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proceedings of SIGMOD'24* 2, 1 (2024), 65:1–65:28.

[38] David J. C. MacKay. 2003. *Information Theory, Inference, and Learning Algorithms*. Copyright Cambridge University Press.

[39] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[40] V Mahoney Matthew. 2005. Adaptive weighing of context models for lossless data compression. *Florida Institute of Technology CS Dept, Technical Report* (2005).

[41] Baligh Mnassri. 2020. Jena Climate Dataset. https://www.kaggle.com/datasets/mnassrib/jena-climate

[42] Alistair Moffat. 2019. Huffman coding. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–35.

[43] Orestis Polychroniou and Kenneth A Ross. 2015. Efficient lightweight compression alongside fast scans. In *Proceedings of DaMoN@SIGMOD'15*. 1–6.

[44] Meikel Pöss and Dmitry Potapov. 2003. Data Compression in Oracle. In *Proceedings of VLDB'03*. VLDB Endowment, 937–947.

[45] Yiming Qiao, Yihan Gao, and Huanchen Zhang. 2024. Blitzcrank: Fast Semantic Compression for In-memory Online Transaction Processing. arXiv:2406.13107

[46] Vijayshankar Raman and Garret Swart. 2006. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *Proceedings of VLDB'06*. VLDB Endowment, 858–869.

[47] Amir Said. 2004. Comparative Analysis of Arithmetic Coding Computational Complexity.. In *Data compression conference*. Citeseer, 562.

[48] Jia Shi. 2020. Column partition and permutation for run length encoding in columnar databases. In *Proceedings of SIGMOD'20*. 2873–2874.

[49] Tanmay Sinha. 2021. OLAP vs. OLTP: What's the Difference? https://www.ibm.com/blog/olap-vs-oltp/

[50] Sivaprasad Sudhir, Michael Cafarella, and Samuel Madden. 2021. Replicated layout for in-memory database systems. *Proceedings of VLDB'21* (2021), 984–997.

[51] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of SOSP'13*. 18–32.

[52] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of SIGMOD'18*. 1541–1555.

[53] 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 *Proceedings of DBTest@SIGMOD'18*. 1:1–1:6.

[54] Ian H Witten, Radford M Neal, and John G Cleary. 1987. Arithmetic coding for data compression. *Commun. ACM* 30, 6 (1987), 520–540.

[55] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of SIGCOM'18*. 561–575.

[56] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of SIGMOD'16*. ACM, 1567–1581.

[57] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of SIGMOD'20*. ACM, 1601–1615.

[58] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proc. VLDB Endow.* 15, 11 (2022), 2679–2691.

[59] Junyi Zhao, Huanchen Zhang, and Yihan Gao. 2023. Efficient Query Re-optimization with Judicious Subquery Selections. *Proc. ACM Manag. Data* 1, 2 (2023), 185:1–185:26.

[60] Hu Zhu, Yiming Qiao, Guoxia Xu, Lizhen Deng, and Yu-Feng Yu. 2020. DSPNet: A Lightweight Dilated Convolution Neural Networks for Spectral Deconvolution With Self-Paced Learning. *IEEE Trans. Ind. Informatics* 16, 12 (2020), 7392–7401.

[61] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *Proceedings of SIGMOD'22*. 685–699.

[62] Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343.

[63] Jacob Ziv and Abraham Lempel. 1978. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Inf. Theory* 24, 5 (1978), 530–536.