Optimizing Data Compression via Data Reordering

Qinxin Du
York University
qinxindu98@gmail.com
Toronto, ON, Canada

Xiaohui Yu
York University
xhyu@yorku.ca
Toronto, ON, Canada

Aijun An
York University
aan@yorku.ca
Toronto, ON, Canada

Dariusz Jania
IBM Cloud and Cognitive Software
dariusz.jania@pl.ibm.com
Krakowiaków 32, Warszawa, Poland

***Abstract*—Data compression is critical for efficient data operations such as migration, backup, and analytical processing. While there has been significant research into compression techniques for multimedia like videos and images, relational data has garnered less attention. Current methods for compressing relational data, such as gzip and LZ4, do not optimize record ordering within tables, potentially missing opportunities for improved compression. It has been shown that grouping similar records together can enhance compression performance, as measured by the compression ratio. As such, we propose two novel record reordering methods: (1) clustering and (2) solving a Travelling Salesman Problem (TSP) to reorder records. Additionally, we apply locality-sensitive hashing (LSH) on textual columns to convert text into vector representations, aiding in clustering and TSP reordering. For columns with a fewer number of distinct values, using run-length encoding (RLE) rather than traditional dictionary-based compression may be more effective. We therefore extend our approach to partition the set of columns in a table into two groups based on the distinct value count and then use appropriate compression algorithms for each, aiming to maximize the overall compression ratio. We conducted extensive experiments with one synthetic and three real datasets to evaluate our proposed methods. The results confirm that our proposed approaches significantly outperform existing baselines, demonstrating their efficacy in relational data compression.**

***Index Terms*—data compression, data migration, classification**

I. Introduction

A. Motivation

Recent advancements in information technology have led to the generation of massive amounts of data every second [1]. This surge in data creation has heightened the need for efficient methods to access, store, and update data from various databases and applications. Concurrently, the proliferation of computer communication networks has resulted in an increased flow of data over communication links [2]. Consequently, there is an urgent need for data compression techniques to reduce redundancy in stored or communicated data, thereby increasing effective data density [2].

Data compression can be categorized into two types: lossless (exact) and lossy (inexact) compression [3]. Lossless compression techniques allow the original data to be perfectly reconstructed from the compressed data, ensuring data integrity. On the other hand, lossy compression techniques achieve higher compression ratios by sacrificing some data precision or quality. Despite extensive research on lossy compression techniques for images and videos, relatively little attention has been given to compressing relational data, which remains the predominant form of data organization in many enterprises.

This paper is concerned with lossless compression of relational data that take a tabular form consisting of rows (records) and columns (attributes). Standard lossless compression techniques may not always yield optimal results for tabular data as these methods might not efficiently capture patterns, repetitions, and localized structures. To address this, researchers have introduced additional step before applying standard compression methods, where rows are first reordered to make them more amenable to the compression algorithm used. Prominently, Lemire et al. introduce a row-reordering approach and utilize it with run-length encoding (RLE) [4] as the compression algorithm. Another study describes an incremental heuristic for selecting columns to compress and determining row order to improve compression ratios for in-memory columnar databases using RLE [5]. Both approaches exclusively use RLE for compression after data reordering, and thus restricting themselves to scenarios where repeated values in a column are prevalent for RLE to be effective. They tend to perform best when applied to columns with a limited number of unique values.

However, for some columns, their specific repetitive patterns, which involve only partial matches between different values, might not benefit from RLE. For instance, while the words “apply” and “apple” are similar and contain the same segment “appl”, they are not considered the same when RLE is applied, and thus placing these two values through reordering does not bring any benefit in terms of compression ratio. It is therefore highly desirable to identify suitable combinations of compression algorithms and reordering methods to be applied to different columns in a tabular dataset, in order to optimize compression ratios. As such, this study focuses on preprocessing the data before reordering it to uncover similar patterns among data values, group them more effectively, and select suitable compression strategies for each column based on their characteristics.

B. Proposed Approach

Standard dictionary-based compression methods belonging to the LZ77 family, such as gzip (Deflate) and LZ4, are widely used in current approaches to compress tabular data due to their versatility across different data types (e.g., numerical, textual, categorical) and broad support across various computing platforms. However, these methods typically can only exploit

repeating patterns within a sliding window of fixed-size. Our approach enhances these standard methods by initially reordering data to cluster similar or frequently occurring sequences together. This reordering allows dictionary-based compression algorithms to detect and compactly encode longer repetitive sequences that appear consecutively.

We implement two data reordering methodologies: k-modes clustering and a heuristic-based solver for the Traveling Salesman Problem (TSP). These techniques optimize the order of records in a way that maximizes the effectiveness of the compression by placing similar records near each other.

To address the challenge of exploiting textual similarities between entries that are similar but not identical (e.g., "12345" and "12346"), we incorporate an advanced data encoding method called locality-sensitive hashing (LSH). LSH transforms raw tabular data into low-dimensional vector representations, assisting in the reordering process by uncovering hidden textual similarities. This method enhances the ability of reordering algorithms to group data items with high textual similarity, making it possible for dictionary-based compression to efficiently compress these groups.

Moreover, we observe that columns with a high number of unique values gain substantial benefits from the LSH encoding process. Conversely, for columns with fewer distinct values, pre-reordering encoding using LSH shows less impact on compression outcomes. On the other hand, in such cases, run-length encoding (RLE) with heuristic-based TSP reordering as proposed by Lemire et al., has shown to be highly effective, as this method is particularly suited for sequences with limited variability.

To optimize the overall compression ratio of a table, we have developed an innovative strategy to partition the columns in a table into two groups based on the distinct value count of each column. This partitioning allows us to apply the most suitable compression techniques to each group. Columns with fewer distinct values are processed using TSP reordering followed by run-length encoding, whereas columns with a larger number of distinct values are subject to LSH encoding followed by TSP reordering and subsequent dictionary-based compression. We establish a threshold for partitioning based on the percentage of unique values in each column, which dictates the appropriate compression strategy for each group. This partitioning approach provides a comprehensive compression solution that adapts to the unique characteristics of each column within the table, significantly enhancing the overall compression ratio.

C. Contributions

In summary, our contributions are as follows:

- We explore reordering of records as a way to improve compression ratio and propose two new methods for this purpose, based on clustering and TSP heuristics respectively.
- We employ Locality-Sensitive Hashing (LSH) to transform data values of various data types into low-dimensional vectors and thus achieve a representation that is amenable to similarity computation for clustering or TSP reordering.
- We devise a data partitioning method to enhance the generality of our approach by considering the unique value percentage in each column, which allows distinct compression strategies to be adopted for different groups of columns.
- We experimentally evaluate the effectiveness of our proposal and present results from a thorough evaluation, testing the impact of different dataset parameters using both synthetic and real datasets, demonstrating impressive performance improvements.

II. Related Work

A. Data Compression Algorithms

Data compression algorithms can be divided into two primary categories based on compression quality: lossy and lossless. Our focus in this paper is on lossless compression. Unlike lossy compression, which permanently discards some information to reduce data size, lossless compression ensures that no data is lost, allowing the original data to be fully restored. Lossless compression methods can be broadly categorized into three types: dictionary-based, entropy-based, and hybrid compression algorithms.

Dictionary-based compression reduces file sizes by identifying repeating patterns or sequences and replacing them with references to a dictionary. Prominent algorithms include LZ77, LZ4, and Zstandard (Zstd). LZ77, created by Ziv and Lempel [6], uses a sliding window to find and replace recurring sequences, making it suitable for compressing repetitive data [7]. This adaptability has led to applications in fields such as computational biology and DNA sequence compression [8], [9], and its evolution into LZ78 has led to hybrid algorithms and precise parameter estimation using convolutional neural networks [10], [6]. LZ4, derived from LZ77, is known for its high-speed compression and decompression, and has been optimized for applications such as FPGA implementations and cloud computing to reduce data transmission costs [11]. Zstandard, known for its efficient compression and adjustable levels, is used in nanopore sequencing and IoT healthcare applications to reduce data broadcast by sensors, e.g., [12].

Entropy-based compression, or entropy coding, uses statistical properties of data to reduce storage space. RLE, a simple lossless algorithm in this category, replaces consecutive repeated symbols with a count and the symbol itself [13], and is effective for compressing data with long sequences of identical symbols in image processing, signal processing, and test data compression [14]. Enhanced RLE methods include Golomb and frequency-directed encoding, as well as adaptive 2D RLE, improving compression ratios when combined with other techniques like Huffman encoding [15].

Hybrid compression combines multiple algorithms for optimal results, with Deflate and gzip integrating LZ77 and Huffman coding. Deflate locates and replaces recurrent sequences using LZ77, followed by Huffman coding for further compression [16], and is widely used in formats like ZIP,

HTTP, and PNG for faster data transmission and better user experience [17]. Gzip, based on Deflate, includes additional features like checksums and file packaging, serving as a benchmark for comparing compression methods and has been improved for real-time compression [18]. Despite its prevalence, gzip has limitations, such as the inability to decompress at arbitrary locations due to the nature of Deflate [19].

B. Data Reordering for Data Compression

Multiple compression techniques used in databases can be affected by data reordering, particularly when it comes to rearranging rows [4].

Poss and Potapov proposed an innovation compression technique in Oracle [20]. Their work highlights that reordering data before loading can enhance the compression ratio of a database table, as columns arranged sequentially may possess identical content. Sorting the data on fields with fewer unique values in a database object will likely enhance the compression factor.

Data compression in the column-oriented database systems is improved through the process of data reordering, where data is stored by column rather than by row. Stonebraker et al. discovered that in C-Store, a column-oriented database, sorting a column results in grouping the same values together [21]. This grouping facilitates the compression of these values using techniques like run-length encoding.

Lemire et al. evaluate several row reordering heuristics, including the Lexicographic Order heuristic, the Vertex heuristic, the Multiple Lists heuristic, and the TSP heuristic to improve compression rate for database tables [4]. We employ this method as part of our proposed DVP-partitioned table compression strategies for cetain columns.

Pan et al. [22] explore the use of online clustering of records in a table to achieve a higher compression ratio in the particular setting of data migration from one site to another, where the main focus is the trade-off between clustering overhead and the improvement in compression ratio. In contrast, we deal with the general setting of data compression with the objective of optimizing the compression ratio.

III. Problem Description

Let $\mathbf{T}$ denote a relational table comprising multiple columns $\mathcal{C}$, where each specific column c_i represents an attribute. We use r_j to denote the j-th record in T as per the physical order of records in T. The data entry at i-th column and j-th row in $\mathbf{T}$ is represented as d_{ij}. Our research goal is to optimize the compression ratio of the relational table $\mathbf{T}$ and improve its storage efficiency.

As c_i represents a single attribute in $\mathbf{T}$, such as the gender or age of a person, it contains data of uniform type with repeating values. Thus, instead of compressing data row by row, we compress c_i independently to better exploit the similarities between values in the same c_i to achieve a higher compression ratio. Specifically, we apply either a dictionary-based compression algorithm or RLE to c_i stored in $\mathbf{T}$ to implement column-wise compression.

TABLE I
Notations Used in the Problem Description

Notation	Description
$\mathbf{T}$	A relational table
$\mathcal{C}$	The set of columns within $\mathbf{T}$
c_i	The *i*th column in original $\mathbf{T}$
r_j	The *j*th record in original $\mathbf{T}$
d_{ij}	The data entry at *i*th column and *j*th row in $\mathbf{T}$

IV. Methodologies

The primary idea of our methodologies is to reorder the data records based on their similarity so that the compression algorithms can better exploit the repeating sequences and compress the data into smaller sizes. We propose two data reordering methods: Clustering-based and Traveling Salesman Problem (TSP)-based. Since the dictionary-based compression algorithms treat all values as strings, we consider all column values as strings by default, and a data representation algorithm, Locality Sensitive Hashing (LSH), is applied to the data values before data reordering. It aims to transform different types of data (e.g., categorical, numerical, textual, or mixed) into a uniform format (low-dimensional vector representations) and expose hidden textual similarity to data reordering algorithms.

Yet for some columns, it may be the case that there are only a very limited number of distinct values being repeated across records, and thus it is more sensible to apply the RLE compression algorithm after reordering as done in [4]. To handle such cases, a table-partitioned compression strategy based on the number of distinct values within an individual column is proposed to further improve the overall compression ratio for a whole table. The table is partitioned into two groups via a configurable threshold: columns with a few distinct values and columns with many distinct values. Each group of columns will then undergo suitable recording strategies respectively. Finally, the compression algorithm compresses tabular data column by column.

Data reordering methods are discussed in Section IV-B (Clustering-based) and Section IV-C (TSP-based) respectively. Before discussing these methods, we present the data representation algorithm in Section IV-A. In Section IV-D, we describe our table-partitioned compression strategies.

A. Data Representation Algorithm

Repetitive patterns often appear among values in a column. Some of them are the same, while others are just similar. For instance, A column representing attribute “Gender” has many repeating values of “Male” and “Female”. Another column represents the attribute “ID” with values like “ID-1110”, “ID-1111”, and “ID-1112”. Even though they share the same substring “ID-111”, they are different data entries which data reordering algorithms will not group them closer.

To account for overlapping substrings in similarity computations, we utilize Locality Sensitive Hashing (LSH) [23] to represent the values in the data table. With LSH, each value in a column is first sliced into pieces (called shingles), which

are stored in a vocabulary. Then each shingle in a value is represented by the index of the shingle in the vocabulary. Such indexes for all shingles of a data value d_{ij} are then converted into a dense vector of a small, fixed size using the minHashing technique [24]. This dense vector is called the *signature* of d_{ij}. The LSH method consists of the following two steps.

(1) **Shingling** is the process of transforming a string into a set of $shingles$. It's akin to sliding a window of length Q down the data string and capturing a snapshot at each step. These snapshots are saved to create a set of $shingles$. The vocabulary $vocab$ comprises all the $shingles$ spanning all d_{ij}s. The vector representation of each d_{ij} is formed using this $vocab$.

For example, for the string "Hello", when we set Q to 2, the $vocab$ contains $shingles$: "He", "el", "ll", "lo".

(2) **MinHashing**: After the shingles are obtained, we further apply MinHashing to generate lower-dimensional signatures for computing similarity between values d_{ij}. For each set of shingles corresponding to a d_{ij}, M hash functions are applied to produce hash values, and the minimum hash value for each hash function is recorded. This process generates a MinHash signature, a compact representation of the set. The similarity between two sets can then be estimated by comparing their MinHash signatures, specifically by counting the number of matching hash values and dividing by the total number of hash functions used. Compared to main competing methods like exact Jaccard similarity, MinHash offers significant benefits: it is highly scalable, reducing computational complexity from quadratic to linear; it is space-efficient, storing small signatures instead of large sets; it provides fast, approximate similarity estimations, which are often sufficient for practical applications; it is simple to implement and can be easily parallelized for distributed computing environments, making it ideal for handling large-scale data.

The procedure to implement locality sensitive hashing for single value d_{ij} is described in Algorithm 1. Initially, we extract $shingles$ of sliding window size Q from a data entry d_{ij}, where each $shingle$ is added to the set $shingles$ (lines 6-10). Next, M hash functions are generated using random integers a and b within the range of the vocabulary size $|vocab|$, defining each hash function as $h(x) = (a \cdot x + b) \mod |vocab|$ (lines 11-17). For each hash function, the algorithm iterates over all shingles of d_{ij} to compute hash values and determine the minimum hash value for each hash function (lines 18-28). Finally, the computed MinHashing signature vector $signature$ is returned as the output which used to represent a d_{ij} (line 26). Without MinHashing process, the dimensionality of the data representation vector for each data entry depends on the size of $vocab$ which is typically high for columns with many distinct values. In contrast, the dimensionality can be significantly reduced after applying hash functions while preserves the similarity between $signatures$.

Figure 1 illustrates an example of LSH. Each data entry in the column *Car Name* is transformed into a four-dimensional vector using sliding window size Q=2 for shingling and M=4 for MinHashing. For instance, the $shingles$ for the string "Hyundai Grand i10 CRDi Asta" include "Hy", "yu", "un", "nd" ..., "ta". After applying shingling to all the d_{ij} in the column, the $vocab$ consists of 1,369 $shingles$. We apply four hash functions to produce a 4-dimensional $signature$, such as [74, 27, 454, 1246], for this long string. We can observe that similar strings share similar signatures.

Note that the dictionary-based compression also utilizes the concept of sliding window to identify repeated data patterns. Therefore, the segmentation of d_{ij} via the shingling process shares similarity with dictionary-based compression. Thus, grouping data based on the LSH data representation can benefit dictionary-based compression by detecting repetitive subsequences in data values.

Car Name	D1	D2	D3	D4
Hyundai Grand i10 CRDi Asta	74	27	454	1246
Hyundai Grand i10 1.2 CRDi Sportz	74	1101	454	1246
Mahindra TUV 300 T4	860	1180	454	1246
Mahindra Bolero SLX 4WD BSIII	860	1180	454	1095
Mahindra XUV300 W8 Option Diesel BSIV	860	1180	454	790

Fig. 1. An LSH Representation Example for Column *Car Name*

Algorithm 1 Locality Sensitive Hashing

1: **Input:** d_{ij}: data entry at *i*th column and *j*th row,
2: Q: shingling window size,
3: $vocab$: vocabulary of shingles,
4: M: dimension of signature
5: **Output:** $signature$ of d_{ij}
6: $shingles \leftarrow \emptyset$
7: **for** $k \leftarrow 1$ to $|d_{ij}| - Q + 1$ **do**
8: $shingle \leftarrow d_{ij}[k : k + Q]$
9: $shingles \leftarrow shingles \cup \{shingle\}$
10: **end for**
11: $hashFunctions \leftarrow \emptyset$
12: **for** $i \leftarrow 1$ to M **do**
13: $a \leftarrow$ random.randint$(1, |vocab|)$
14: $b \leftarrow$ random.randint$(1, |vocab|)$
15: $hashFunction \leftarrow \lambda x : (a \cdot x + b) \mod |vocab|$
16: $hashFunctions \leftarrow hashFunctions \cup \{hashFunction\}$
17: **end for**
18: $m \leftarrow 1$
19: **for** $hashFunc$ **in** $hashFunctions$ **do**
20: $minValue \leftarrow +\infty$
21: **for** $shingle$ **in** $shingles$ **do**
22: $i \leftarrow$ index of $shingle$ in $vocab$
23: $hashValue \leftarrow$ applyHashFunction$(i, hashFunc)$
24: $minValue \leftarrow \min(minValue, hashValue)$
25: **end for**
26: $signature[m] \leftarrow minValue$
27: $m \leftarrow m + 1$
28: **end for**
29: **return** $signature$

B. Clustering-based Methods

Data compression entails encoding information using fewer bits than its original representation. However, similar or iden-

tical values dispersed throughout a table may not be efficiently exploited by conventional compression algorithms. Clustering is the process of grouping object attributes and features such that the data objects in one group are more similar than data objects in another group with no prior knowledge of data structure [25]. It can enhance data compression by identifying patterns or similarities within the data and subsequently grouping similar elements together, thereby improving the efficiency of data compression.

K-modes clustering is an unsupervised machine learning algorithm that is analogous to the k-means algorithm but designed to handle only categorical variables [26]. In our proposed technique, clustering is performed on the concatenation of $signatures$ obtained from LSH, which represent d_{ij}s in each c_i and belong to the same r_j. Each signature consists of hash values, which are by definition un-ordered and thus should be treated as categorical values as opposed to numerical (hence the choice of k-modes over k-means). The number of clusters, K, is set as a parameter for the experiments before the clustering process. After k-modes clustering, **T** is partitioned into K groups. As shown in Figure 2, all rows r_j are grouped into three clusters Each cluster exhibits a significant degree of internal similarity. These three clusters are then sequentially merged to form the reordered **T**. Subsequently, dictionary-based compression techniques are expected to be applied to each reordered c_i.

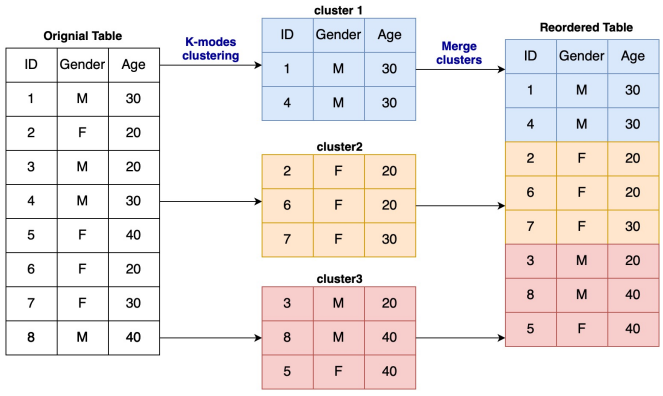

Fig. 2. Clustering-based Table Reordering Diagram

While k-modes clustering is effective in improving compression ratios, its application to large datasets presents a computational challenge. In real-life scenarios, the amount of time required for reordering data using clustering methodologies renders it impractical. Hence, there is an urgent need to explore alternative, more time-effective approaches for rearranging data and identifying similar patterns suitable for dictionary-based compression.

C. TSP-based Methods

In our methodology, TSP solutions serve a distinct purpose: they function as a data reordering technique, aiming to optimize the arrangement of data items based on their similarity. Unlike the traditional TSP formulation, which conceptualizes cities as physical entities with spatial relationships, data elements are abstract entities typically representing data points or vectors in data reordering. As a result, the distance calculation is adjusted to assess the similarity between data points rather than computing physical spatial distances. In contrast to graph-based reordering algorithms, which heavily rely on clear hierarchical or relational structures present in the given data, TSP solutions offer greater adaptability to general cases, as they do not necessarily require prior knowledge of the data structure.

We consider the open-loop Traveling Salesman Problem (TSP) where the salesperson does not return to the starting city and find the optimal order for data items using a greedy local-search heuristic, which is referred to as a heuristic-based TSP solver in our approach [27]. To initiate the process of applying this heuristic-based TSP solver to data reordering, we first construct a square matrix representing the distances between data points, denoted as d_{ij}. For clarity, let's consider an example illustrated in Figure 3 featuring three data points: X, Y, and Z. The distance from point X to itself is 0, the distance from X to Y is 2, and the distance from X to Z is 3. As a result, the first row of the square matrix displays the distances originating from X, denoted as (0, 2, 3). Following the same procedure for Y and Z, we can derive the complete square matrix encompassing all three data points.

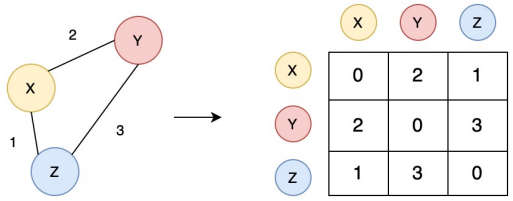

Fig. 3. Example of Square Matrix Formation

The distance between d_{ij} is measured by Jaccard distance in our work. The Jaccard index, also referred to as the Jaccard similarity coefficient, assesses the similarity and dissimilarity of sample sets [28]. This coefficient quantifies the similarity between sample sets by evaluating the ratio of the intersection to the union of the sets. Conversely, the Jaccard distance, representing dissimilarity between sample sets, complements the Jaccard coefficient. Equation 1 illustrates the calculation of the Jaccard distance:

$$J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

where A and B are two sets, $|A \cap B|$ denotes the cardinality of the intersection of A and B, and $|A \cup B|$ denotes the cardinality of the union of A and B.

Utilizing the distance matrix generated using the Jaccard distance, the heuristic-based TSP solver starts to construct an initial solution greedily aiming to minimize the total distance traveled. Subsequently, the local search heuristic is applied to optimize this initial solution. This heuristic iterates through

pairs of edges in the solution, attempting to replace them with more optimal pairs whenever possible. This iterative process continues until no further improvement can be achieved. Finally, the rows (records) are reordered based on the optimal path derived from the heuristic-based TSP solver.

TSP is a well-known NP-hard problem [29]. For an efficient (yet approximate) solution, our approach involves partitioning the $\mathbf{T}$ into chunks of records, solving TSP within each individual chunk. Based on resolved tours for TSP, we reorder the tuples within each chunk. As illustrated in Figure 4, all the d_{ij} of $\mathbf{T}$ are reordered by chunk. Finally, we concatenate all the chucks of $\mathbf{T}$ with the optimal order together.

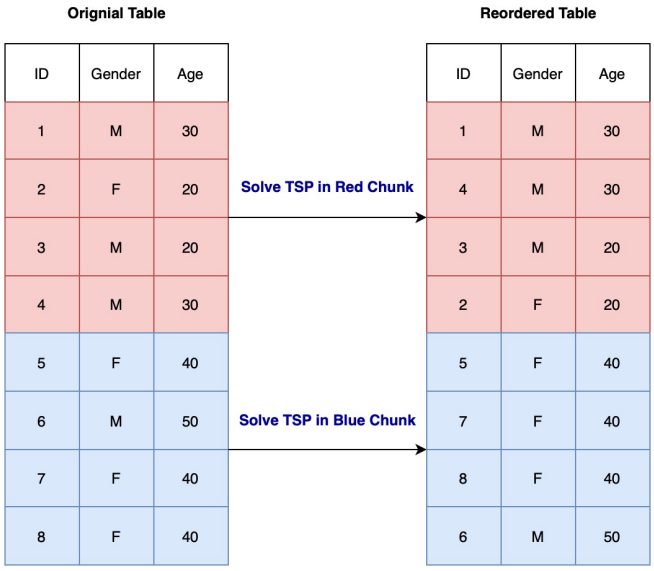

Fig. 4. Example of Chunk-wise TSP Solution

Reordering the data through a heuristic-based TSP solver based on similarity facilitates the detection of repetitive patterns, particularly through the application of dictionary-based compression techniques. Compared to the previously mentioned k-modes clustering method, the heuristic-based TSP solver reorders data by chunks, proving to be more time-efficient while providing excellent reordering capability.

For the compression process, column-wise compression will be implemented on individual c_i of the reordered table to attain a higher compression ratio, given that the data within c_i typically shares similar traits in tabular data.

D. DVP-Partitioned Table Compression

The work by Lemire et al. has been shown to be effective for compressing columns with a smaller number of distinct values [4]. Therefore, to integrate it with the reordering methods introduced in Sections IV-B and IV-C, we introduce a novel table-partitioned compression strategy based on the distinct value percentages in individual columns. We call it DVP (Distinct Value Percentage)-Partitioned table compression.

The DVP in a column is calculated as follows:

$$\text{DVP} = \frac{\text{the number of unique values within the column}}{\text{total number of records}} \tag{2}$$

DVP-Partitioned table compression aims to categorize $\mathcal{C}$ into different groups based on DVP. The same compression strategy is applied to the $\mathcal{C}$ within the same group. Additionally, each group is reordered by the heuristic-based TSP solver together, and a corresponding row ID for each r_j is assigned before reordering. This is to ensure the recovery of the original order after decompression. If $\mathbf{T}$ contains only a few columns and rows, dividing it into too many groups will lead to a large extra storage cost for row IDs, which is not favorable for calculating the overall compression ratio.

As a result, we decide to partition use a single DVP threshold to partition $\mathcal{C}$ into two groups only:
1. DVP $>$ DVP threshold
2. DVP $\leq$ DVP threshold

The DVP threshold here is adjustable, and based on the results of numerous experiments, we typically set its value in the range of $[0.01, 0.1]$.

The procedures for DVP-Partitioned table compression are illustrated in Figure 5. DVP-Partitioned table compression works as follows. For a table $\mathbf{T}$ to be compressed, the DVP for every c_i in $\mathbf{T}$ is calculated to facilitate the following column grouping process. After comparing each column's DVP with the DVP threshold, all the columns are divided into two groups.

The LSH algorithm preprocesses the group of c_i with DVP larger than the DVP threshold: each c_i undergoes LSH once and is transformed into high-dimensional vectors. The reordering via the heuristic-based TSP solver is then performed utilizing the vectors generated from all c_i within the group. Finally, the dictionary-based compression algorithm is applied to each reordered column.

The other group of c_i with DVP smaller than the DVP threshold undergoes a simple data preprocessing stage. The distinct values in each c_i are assigned ordinal values to aid the following data reordering and run-length encoding procedures. The compression process for this group involves three compression algorithms performing different functions. First, run-length encoding is applied to compress each reordered c_i as it is used to handle repetitive sequences efficiently. Then, the RLE result is further compressed and converted into compact binary files using the messagepack algorithm, which aims to optimize serialization to reduce overhead and represent data compactly. Finally, Zstandard compression, which provides adaptive compression for different data types, is applied to compact binary files to further reduce the compressed files' size and enhance compression performance. As the three compression algorithms work differently but corporately to exploit the duplicate patterns in smaller DVP columns, there is no existing single compression technique that can achieve similar functions and compression performance.

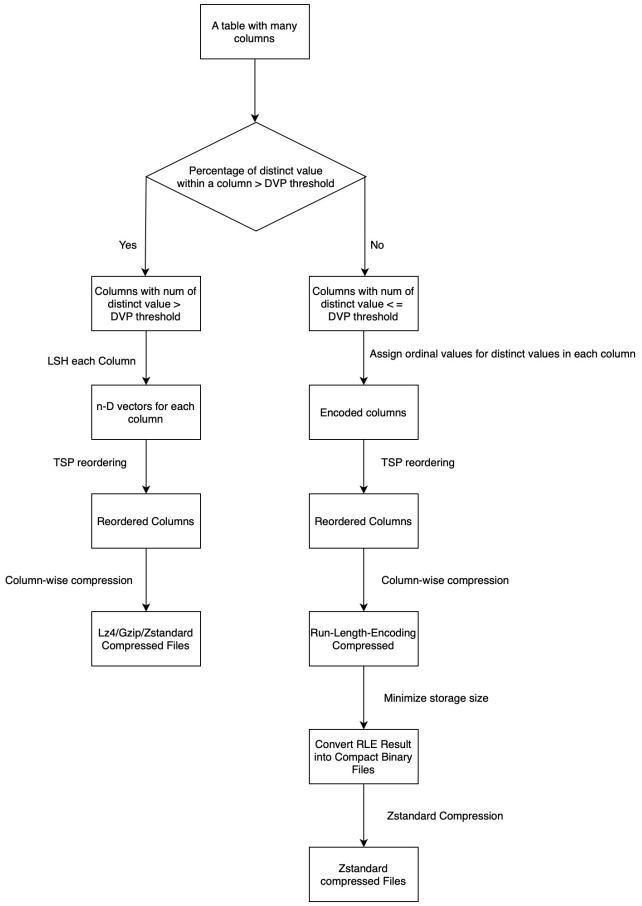

Fig. 5. DVP-Partitioned Table Compression

V. Experiments

A. Settings

The experiments are conducted on a server with an AMD Ryzen 24-Core Processor and 128GB RAM. The codes for all the experiments were written in Python 3.9. The primary libraries utilized include Pandas, NumPy, and SciPy.

B. Datasets

The proposed methodologies are assessed using one synthetic and three real datasets. Both numerical and categorical data are present throughout every dataset. DS001 dataset is a proprietary dataset derived from the TPC-H benchmark [30] that is used internally at IBM. The Bank Marketing and Census Income datasets are sourced from the UCI Machine Learning Repository [31], [32]. The Vehicle dataset is obtained from Kaggle [33]. These datasets are chosen to represent small, medium, and large datasets based on the number of columns and tuples, ensuring the versatility of DVP-partitioned table compression. An overview of the datasets' information is presented in Table II. The *Cols* represents the total number of columns in datasets. In addition, we present the number of columns with numerical (Num.), categorical (Cat.), and mixed (numerical and categorical) data for each dataset.

TABLE II
Overview of Dataset Information

Name	#Rows	#Cols	Num.	Cat.	Mix.	Size (MB)
DS001	386,671	7	4	3	0	63.7
Bank Marketing	45,212	17	7	10	0	3.8
Census Income	32,462	15	6	7	2	3.6
Vehicle	8,129	13	4	5	4	1.1

C. Evaluation

The data compression ratio is used to evaluate the result. It is calculated as the ratio of the uncompressed file size to the compressed file size:

$$\text{Compression Ratio} = \frac{\text{Uncompressed Files Size}}{\text{Compressed Files Size}} \quad (3)$$

The compression ratio of the random arrangement of columns serves as the benchmark for the compression ratio of individual columns.

For the evaluation of DVP-Partitioned table compression, the overall compression ratio (OCR) after reordering is calculated as indicated in Equation 4. As the columns are reordered by groups, individual group obtains a sequence of row IDs which is used to recover the original order after decompression. The uncompressed files are reordered columns and two sets of row IDs as the table is divided into two groups based on each column's DVP. The compressed files are compressed columns and compressed sets of row IDs.

$$\text{OCR} = \frac{\sum(\text{Uncompressed Files Size} + \text{Row IDs Size})}{\sum(\text{Compressed Files Size} + \text{Row IDs Size})} \quad (4)$$

To ensure a fair comparison, we compress columns separately in random order. The baseline is calculated as the sum of all the compressed random-order column sizes divided by the sum of all the uncompressed random-order column sizes.

D. LSH and K-modes clustering

To investigate the impacts of two LSH parameters and the number of clusters on the column-wise compression ratio when reordering via k-modes clustering. We introduce three major parameters employed in the LSH and k-modes clustering experiments:

- Q is the size of sliding windows for shingling
- M is the dimension level for $signiture$
- K is the number of clusters for k-modes clustering

The ultimate reordered file is formed through the concatenation of all clusters. Specifically, cluster 2 is appended to cluster 1, and this process extends sequentially to include cluster 3 and beyond. The comparison is between one compressed original file and one compressed reordered file.

We showcase the `mileage` column from the Vehicle dataset for LSH parameters' experiments due to its inclusion of multiple recurring patterns, a characteristic that is easily discernible for humans but poses a challenge for computational analysis. Furthermore, the tuples within `mileage` consist of

mixed types of data (categorical and numerical data) instead of solely one type of data (categorical or numerical data) in other columns, reflecting a scenario akin to those encountered in real-life situations.

Regarding the clustering parameter K, the column `Customerkey` from the DS001 dataset is employed, given its substantial number of tuples, which can significantly influence the determination of the optimal number of clusters.

1) Sliding Window Size Q for LSH: In this experiment, we aim to investigate the impact of the LSH parameter Q on compression results. We select a column `mileage` to perform the experiments as it contains numerical and textual data. To maintain consistency, we employ k-modes clustering as the method for data reordering, with a fixed cluster number of 10. Another crucial LSH parameter, M, is set to a constant value of 2 or 3. Subsequently, we vary the size of Q to obtain reorganized columns and apply three commonly used dictionary-based compression algorithms to assess the compression outcomes.

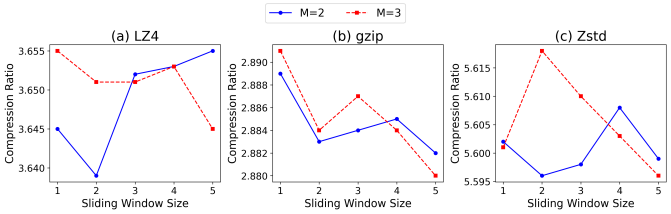

Fig. 6. Compression Ratio for `mileage` – Varying Q (M=2/3, K=10)

As depicted in Figure 6, we observe that, under the influence of three different compression algorithms, the tendencies of compression ratios for the reordered column are different when increasing Q.

Figure 6(a) shows that the LZ4 compression ratio increases when Q increases from 2 to 5, with M=2 and K=10. However, when M is set to 3, the tendency is distinct when Q ranges from 2 to 5. The highest compression ratio is 3.655, achieved when Q=1 and M=3, or when Q=5 and M=2. This indicates that the parameters M and Q are highly relevant for the LZ4 compression method.

Figure 6(b) illustrates that increasing Q leads to a generally decreasing compression ratio for M=2 and M=3 for gzip. The highest compression ratio is obtained when Q=1.

Figure 6(c) demonstrates a distinct tendency when M=2 and M=3. The settings of Q and D are highly relevant for Zstd. The highest compression ratio is achieved when Q=2 and M=3.

In summary, optimal compression performance is achieved for all three compression methods when Q is set to a relatively small value. A smaller sliding window size Q usually leads to a smaller vocabulary ($vocab$) and better string matching when the column contains many repetitive patterns. However, the effect of Q might vary when the setting of M changes. Notably, each column possesses distinct properties and features, resulting in varying impacts of Q across different columns.

2) Dimensional Level M for LSH: Similar to the experimentation with the parameter Q mentioned above, the experiment on the parameter M employed the method of controlling variables to investigate its impact on compression effectiveness. The cluster number K for k-modes clustering is fixed at 10, and the value of parameter Q is set to 2 or 3. By varying the value of M, we aimed to observe its influence on the compression of the reordered column "mileage". The compression algorithms utilized in this experiment are LZ4, gzip, and Zstd.

Figure 7 shows that different values of Q lead to nearly opposite tendencies for all three compression methods, especially for LZ4 and Zstd. This emphasizes that parameter M is significantly related to parameter Q when it comes to the compression ratio of the reordered column.

The highest compression ratio for LZ4 is achieved when M=4, as shown in Figure 7(a). Similarly, in the case of gzip, the compression ratio in Figure 7(b) reaches its optimal when the value of M is 4. Figure 7(c) indicates that the peak point of the Zstd compression ratio is obtained when M=3.

In conclusion, the impact of parameter M on the reordered column's compression ratio is sensitive to the setting of parameter Q. The optimal compression ratio is more likely to be achieved when M is a relatively large value. M represents the dimensional level of the signature. A higher dimensional level of the signature can typically reveal more details and patterns of data than a lower dimensional level of the signature. It is noticed that the impact of parameter M on the reordered column's compression ratio may vary across columns due to differences in data types and lengths for each column.

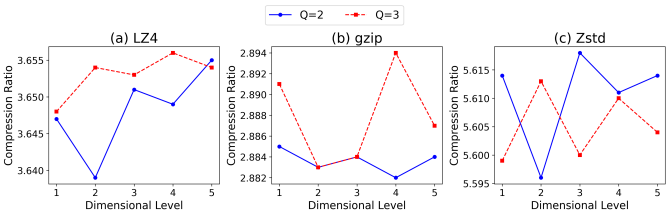

Fig. 7. Compression Ratio for `mileage` – Varying M (Q=2/3, K=10)

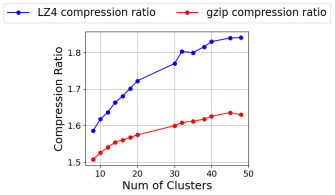

Fig. 8. Compression Ratio for `Customerkey` – Varying K (Q=3, M=5)

3) Number of Clusters K for K-modes Clustering: After conducting numerous experiments with various parameter settings, the optimal LSH parameters for the "CustomerKey" column were determined to be Q = 3 and M = 5. Consequently, to investigate the impact of the cluster number K, we set Q and M as 3 and 5, respectively. The rearranged column is compressed using the LZ4 and gzip algorithms. As illustrated in Figure 8, with K varying from 10 to 48, the compression ratio exhibits a continuous increase. On average, the compression ratio increases by 17.01% for LZ4 and 12.80% for gzip as K varies. When K is set to 30,

the compression ratio obtains a 25.6% improvement for LZ4 and a 16.1% improvement for gzip compared to baselines of 1.408 and 1.377, respectively, which were generated from the random split of the column for LZ4 and gzip.

Also, considering the diverse data types, lengths, and record counts of distinct columns, the influence of parameter K on the overall compression ratio may vary for different columns.

E. TSP-based vs Clustering-based

Both the heuristic-based TSP solver and k-modes clustering serve as data reordering methods in our approach. Experiments on whole table compression using the Bank Marketing dataset are conducted to assess their respective effects on enhancing overall compression performance. In real-life scenarios, the execution time of data reordering algorithms may be a limiting factor. Therefore, we record not only the overall compression ratio of both methods but also their processing time.

To guarantee an unbiased comparison, the heuristic-based TSP solver and k-modes clustering undergo identical data preprocessing stages. For each record, the values from all columns are concatenated into a single column, and LSH is applied to this consolidated column. Both LSH parameters, Q and M, are set to 6 to ensure the revelation of more detailed patterns within the consolidated column.

The heuristic-based TSP solver works chunk-by-chunk to expedite the reordering process. The chunk size is consistently set to 5,000 for all experiments.

TABLE III
COMPRESSION RATIOS AND EXECUTION TIMES FOR BANK MARKETING DATASET USING HEURISTIC TSP SOLVER / K-MODES CLUSTERING

Method	LZ4 ratio	gzip ratio	Time (hh:mm:ss)
TSP Solver	3.798	5.863	00:11:30
K-modes (K=5)	3.613	5.963	00:28:00
K-modes (K=10)	3.696	6.048	00:54:25
K-modes (K=15)	3.765	6.122	01:17:16

In Table III, it is evident that the execution time for all k-modes clustering experiments is considerably longer than that for the heuristic-based TSP solver. The compression ratio for k-modes clustering shows a slight increase with the rise of parameter K. The heuristic-based TSP solver achieves a higher compression ratio than all clustering methods when using LZ4. However, it attains the lowest compression ratio when using gzip. Overall, the compression ratios for the heuristic-based TSP solver and k-modes clustering show minimal differences, indicating that they can achieve comparable performance for the Bank Marketing dataset. The shorter execution time for the heuristic-based TSP solver stands out as a significant advantage over clustering methods. Consequently, the heuristic-based TSP solver is employed for the data reordering procedure in the subsequent DVP-partitioned table compression.

F. DVP-Partitioned table compression with heuristic-based TSP solver

To ensure the universality of this strategy, we conducted evaluations using three datasets derived from real-world scenarios.

Initially, we conducted experiments on the Vehicle dataset. After determining the DVP for each column, we iteratively grouped columns ten times based on DVP thresholds ranging from 0.01 to 0.1. The values in high DVP columns (columns with fewer unique values) were transformed into signatures through LSH.

The impact of parameters Q and M on the compression ratio depends on the unique properties of individual columns, as explained previously. As a rule-of-thumb, we selected Q=2 and M=4 as the LSH parameters to generate signatures for all columns in the high DVP group. The other group of columns with low DVP underwent the corresponding process.

The calculation of the overall compression for the reordered table accounted for the inclusion of row IDs, ensuring accurate recovery of the original sequences after decompression, as illustrated in Equation 4.

In Figure 9, the blue line depicts the overall compression ratio for the reordered table across DVP thresholds ranging from 0.01 to 0.1. The red line serves as the baseline, calculated as the sum of all compressed random column sizes divided by the sum of all uncompressed random column sizes. Both the blue and red lines are compressed using Zstd, which was chosen for its superior performance compared to LZ4 and gzip.

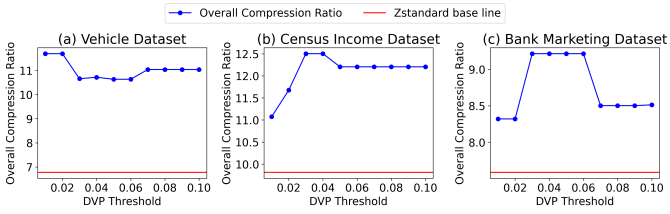

Fig. 9. Overall Compression Ratios for Three Datasets

The peak point of the blue line, reaching an overall compression ratio of 11.692 in Figure 9(a), corresponds to the table partitioned by DVP thresholds of 0.01 and 0.02. This represents a notable improvement compared to the red line, which registers a compression ratio of 6.774. The compression ratio for the Vehicle dataset experiences a substantial increase of 72.6%.

Similar procedures involving appropriate LSH parameter selections are applied to both the Census Income dataset and the Bank Marketing dataset. Specifically, for the Census Income dataset, the parameters Q and M are set to 2 and 3, respectively, while for the Bank Marketing dataset, they are set to 1 and 2.

Figure 9(b) represents the experiment results for the Census Income dataset using DVP-Partitioned table compression. The value of the red line is 9.811. When the DVP threshold is set to 0.03 and 0.04, the optimal table partition is achieved, contributing to an overall compression ratio of 12.503. Compared to the baseline represented by the red line, the reordered table consistently demonstrates a significant improvement in the overall compression ratio, achieving an increase of 27.4%.

The baseline compression ratio is recorded in the Bank Marketing dataset at 7.586, as illustrated in Figure 9(c). The

maximum overall compression ratio is observed at 9.215, achieved when the DVP threshold ranges from 0.03 to 0.06. With the implementation of DVP-Partitioned table compression, the overall compression ratio significantly increases, notably by 21.5%.

VI. Conclusions

In conclusion, this study addresses the gap in compression techniques for relational data by introducing innovative record reordering methods that enhance compression performance. Unlike traditional methods that do not consider record ordering, our approaches—clustering and solving a Travelling Salesman Problem (TSP)—significantly improve the compression ratio by grouping similar records together. Additionally, we leverage locality-sensitive hashing (LSH) to convert textual data into vector representations, facilitating effective clustering and TSP reordering. For columns with fewer distinct values, we utilize run-length encoding (RLE) to further optimize compression. By partitioning table columns based on distinct value counts and applying the most suitable compression algorithms, our methods achieve superior results. Extensive experiments with both synthetic and real datasets validate the efficacy of our approaches, demonstrating significant improvements over existing compression techniques for relational data.

Acknowledgement

Xiaohui Yu and Aijun An are the corresponding authors. This work was supported by an IBM CAS Fellowship and NSERC Discovery Grants.

References

[1] U. Jayasankar, V. Thirumal, and D. Ponnurangam, "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications," *Journal of King Saud University-Computer and Information Sciences*, vol. 33, no. 2, pp. 119–140, 2021.

[2] D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Computing Surveys (CSUR)*, vol. 19, no. 3, pp. 261–296, 1987.

[3] A. Gopinath and M. Ravisankar, "Comparison of lossless data compression techniques," in *2020 International Conference on Inventive Computation Technologies (ICICT)*. IEEE, 2020, pp. 628–633.

[4] D. Lemire, O. Kaser, and E. Gutarra, "Reordering rows for better compression: Beyond the lexicographic order," *ACM Transactions on Database Systems (TODS)*, vol. 37, no. 3, pp. 1–29, 2012.

[5] J. Shi, "Column partition and permutation for run length encoding in columnar databases," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2873–2874.

[6] B. Kwon, H. C. Song, and S. H. Lee, "Accurate blind lempel-ziv-77 parameter estimation via 1-d to 2-d data conversion over convolutional neural network," *IEEE Access*, vol. 8, pp. 43 965–43 979, 2020.

[7] S. Shu and Y. Shu, "A two-stage data compression method for real-time database," *2012 3rd International Conference on System Science, Engineering Design and Manufacturing Informatization*, 2012.

[8] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, "Storage and retrieval of highly repetitive sequence collections," *Journal of Computational Biology*, vol. 17, pp. 281–308, 2010.

[9] A. I. Maarala, O. Arasalo, D. Valenzuela, V. Mäkinen, and K. Heljanko, "Distributed hybrid-indexing of compressed pan-genomes for scalable and fast sequence alignment," *Plos One*, vol. 16, p. e0255260, 2021.

[10] Y. Li, B. Yu, D. Su, and X. Dai, "A nearly loss-less compression technology based on ctp and partial sample points calibration for ofdm signal," *Proceedings of the 2015 International Conference on Computer Science and Intelligent Communication*, 2015.

[11] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja, "Test prioritization in continuous integration environments," *Journal of Systems and Software*, vol. 146, pp. 80–98, 2018.

[12] H. Gamaarachchi, H. Samarakoon, S. P. Jenner, J. M. Ferguson, T. G. Amos, J. M. Hammond, H. Saadat, M. A. Smith, S. Parameswaran, and I. W. Deveson, "Fast nanopore sequencing data analysis with slow5," *Nature biotechnology*, vol. 40, no. 7, pp. 1026–1029, 2022.

[13] B. Madoš, Z. Bilanova, and J. Hurtuk, "rle," *Journal of Information and Organizational Sciences*, vol. 45, pp. 329–349, 2021.

[14] Y. Shan, Y. Ren, Z. Guo-yong, and K. Wang, "An enhanced run-length encoding compression method for telemetry data," *Metrology and Measurement Systems*, vol. 24, pp. 551–562, 2017.

[15] P. Wu, S. Zhou, B. wan, F. Fang, and S. Zhou, "An improved two-dimensional run-length encoding scheme and its application," *International Journal of Signal Processing, Image Processing and Pattern Recognition*, vol. 9, pp. 339–346, 2016.

[16] M. Alsenwi, M. Saeed, T. Ismail, H. Mostafa, and S. Gabran, "Hybrid compression technique with data segmentation for electroencephalography data," in *2017 29th International Conference on Microelectronics (ICM)*. IEEE, 2017, pp. 1–4.

[17] M. Weinberger, G. Seroussi, and G. Sapiro, "The loco-i lossless image compression algorithm: principles and standardization into jpeg-ls," *IEEE Transactions on Image Processing*, vol. 9, pp. 1309–1324, 2000.

[18] A. S. Shah and M. A. J. Sethi, "The improvised gzip, a technique for real time lossless data compression," *EAI Endorsed Transactions on Context-aware Systems and Applications*, vol. 6, no. 17, pp. e5–e5, 2019.

[19] M. Kerbiriou and R. Chikhi, "Parallel decompression of gzip-compressed files and random access to dna sequences," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 209–217.

[20] M. Pöss and D. Potapov, "Data compression in oracle," in *Proceedings 2003 VLDB Conference*. Elsevier, 2003, pp. 937–947.

[21] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-store: a column-oriented dbms," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, 2018, pp. 491–518.

[22] J. Pan, Y. Peng, K. Li, A. An, X. Yu, and D. Jania, "Optimizing data migration using online clustering," in *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '23, 2023, p. 173–178.

[23] O. Jafari, P. Maurya, P. Nagarkar, K. M. Islam, and C. Crushev, "A survey on locality sensitive hashing algorithms and their applications," *arXiv preprint arXiv:2102.08942*, 2021.

[24] W. Wu, B. Li, L. Chen, J. Gao, and C. Zhang, "A review for weighted minhash algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 6, pp. 2553–2573, 2020.

[25] M. Goyal and S. Aggarwal, "A review on k-mode clustering algorithm." *International Journal of Advanced Research in Computer Science*, vol. 8, no. 7, 2017.

[26] A. Chaturvedi, P. E. Green, and J. D. Caroll, "K-modes clustering," *Journal of classification*, vol. 18, pp. 35–55, 2001.

[27] M. H. Rashid and M. A. Mosteiro, "A greedy-genetic local-search heuristic for the traveling salesman problem," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, 2017, pp. 868–872.

[28] M.-U.-S. Shameem and R. Ferdous, "An efficient k-means algorithm integrated with jaccard distance measure for document clustering," in *2009 First Asian Himalayas International Conference on Internet*, 2009, pp. 1–6.

[29] S. Deb, S. Fong, Z. Tian, R. K. Wong, S. Mohammed, and J. Fiaidhi, "Finding approximate solutions of np-hard optimization and tsp problems using elephant search algorithm," *The Journal of Supercomputing*, vol. 72, pp. 3960–3992, 2016.

[30] "Tpc-h benchmark," http://www.tpc.org/tpch/, accessed: May 2nd, 2022.

[31] S. Moro, P. Rita, and P. Cortez, "Bank marketing," UCI Machine Learning Repository, 2012, dOI: https://doi.org/10.24432/C5K306.

[32] B. Becker and R. Kohavi, "Adult," UCI Machine Learning Repository, 1996, DOI: https://doi.org/10.24432/C5XW20.

[33] N. V. Nehal Birla and N. Kushwaha, "Vehicle dataset," Kaggle, 2023, dOI: https://www.kaggle.com/datasets/nehalbirla/vehicle-dataset-from-cardekho.