Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Low overhead self-optimizing storage for compression in DuckDB

Author: Jim Stam (2669580)

1st supervisor: Prof. Dr. Peter Boncz
daily supervisor: Dr. Pedro Holanda (DuckDB Labs)
2nd reader: Dr. Hannes Mühleisen

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 25, 2022

Abstract

Embeddable database management systems can handle large amounts of data in limited environments, such as laptops. This data is often stored in the cloud, where disk usage determines the cost, meaning efficient storage is essential. Therefore, the data is compressed using lightweight compression methods like run-length encoding (RLE). In this thesis, we explore methods to make RLE more effective by reordering the data automatically.

We first evaluate several methods, including sorting columns from low to high cardinality and rows lexicographically, exploiting the combined cardinality of columns and two novel methods called Vortex and Multiple Lists. Our evaluation shows that sorting columns from low to high cardinality is an acceptable trade-off between time performance and compression ratio. We then implement this reordering method in the open-source database system DuckDB by modifying the checkpoint process, which is the moment data is written to disk.

Since we focus on real-life usage, we evaluate the implementation using the Public BI benchmark, which contains 46 datasets of real user data. At most, the compression ratio increased by 15%. On average, the compression ratio is improved by 3.23% while increasing the checkpoint time by 98%.

Contents

List of Figures iii

List of Tables v

1 Introduction 1

1.1 Research questions . . . 2

1.2 Thesis outline . . . 3

2 Background 5

2.1 Row-Store vs Column-Store . . . 5

2.2 Bitmap Index . . . 5

2.3 Common compression algorithms in database systems . . . 6

2.4 Partitioning . . . 8

2.4.1 Z-Ordering . . . 8

2.5 Column-oriented file formats . . . 8

2.6 HyperLogLog . . . 10

2.7 DuckDB . . . 11

3 Related Work 13

3.1 Reordering strategies . . . 13

3.1.1 Reordering in practice . . . 19

3.2 Self-organizing data & machine learning . . . 20

4 Comparing data reordering methods 27

4.1 Method . . . 27

4.1.1 Public BI . . . 27

4.1.2 Cluster machine . . . 28

4.1.3 Experimental setup . . . 28

4.2 Experiments . . . 28

4.3 Results . . . 31
4.4 Summary . . . 36

5 Design & implementation in DuckDB 39
5.1 Design . . . 39
5.2 Implementation . . . 41
5.2.1 Enabling and testing . . . 41
5.2.2 Checkpointer . . . 41
5.2.3 RowGroup Scanning . . . 42
5.2.4 Modules . . . 42
5.2.5 Rewriting persistent blocks . . . 43

6 Evaluation in DuckDB 45
6.1 Experimental setup . . . 45
6.1.1 Automatic checkpoints . . . 46
6.1.2 Reordering strategies . . . 46
6.2 Evaluation . . . 46
6.2.1 Baseline performance . . . 46
6.2.2 Real-life performance . . . 49
6.2.3 Time increase per table . . . 50
6.2.4 Profiling . . . 51
6.3 Benefit analyser . . . 53
6.3.1 Evaluation of benefit analyser . . . 53
6.3.2 Using all columns . . . 54
6.4 Summary . . . 55

7 Conclusion 59
7.1 Research questions . . . 59
7.2 Future work . . . 61

References 63

A Exploratory results 69

B DuckDB implementation - pre-optimised 71

List of Figures

2.1 Row-store vs. Column-store (1) 5

2.2 PFOR-DELTA and Bit-packing combined (2) 7

2.3 Z-curve used for Z-Ordering (3) 9

2.4 Parquet file format (4) 10

2.5 Avro file format (5) 10

2.6 ORC file format (6) 11

2.7 DuckDB file format - columns are separated in blocks, with pointers to the specific blocks kept in the Metadata (adapted from (1)). 11

3.1 Multiple Lists algorithm. First sort the table by a column order, then connect each row with the row above and below by Hamming distance (only 3 connections are shown for clarity). The final sort order is created by shortest path of Hamming distances. 15

3.2 Vortex algorithm. Column A is sorted based on its frequent component 1. Column B is sorted with its frequent component 2; the order is reversed in the top partition, which connects the 2s in the middle. This is repeated for column C but with two extra partitions. 16

3.3 Mapping table with movement dropping. Swaps are kept in a mapping table with format «start, newStart». To recreate the original table, this is traversed backwards. Redundant movement information is dropped (indicated in red) 20

3.4 SDC Architecture (7) 22

3.5 DeepSqueeze compression pipeline (8) 25

LIST OF FIGURES

4.1 Combined cardinalities of multiple columns with sorted order. Columns A and B have a combined cardinality of 6. Columns A, B, and C have a combined cardinality of 9. We search for a sort order before the combined cardinality sharply increases (e.g. A & B). 29
4.2 Public BI compression ratios. All reordering strategies, sorting the entire table or per row group (granular) on 11 datasets. "E" for "entire table" and "g" for granular sorting 31
4.3 Public BI reordering times. All reordering strategies (except Vortex), sorting the entire table or per row group (granular) on 45 datasets. "E" for "entire table" and "g" for granular sorting 32
4.4 Public BI compression ratios. All reordering strategies, sorting the entire table or per row group (granular) on 45 datasets. "E" for "entire table" and "g" for granular sorting 33
4.5 Public BI data types 35

5.1 Architecture of the checkpoint process in DuckDB 40
5.2 Architecture of the checkpoint process in DuckDB after implementing the self-optimising storage 40

6.1 Baseline - compression ratio change 47
6.2 Baseline - checkpoint time change 49
6.3 Real-life optimised - compression ratio change 50
6.4 Real-life optimised - checkpoint time change 51
6.5 Checkpointing multiple tables 52
6.6 Architecture of the checkpoint process with benefit analysis module 53
6.7 Benefit analyser - compression ratio change 54
6.8 Benefit analyser - checkpoint time change - average of 5 runs 55
6.9 Reordering only with RLE compressable columns as "key" columns or with all columns - compression ratio change 56
6.10 Reordering only with RLE compressable columns as "key" columns or with all columns - checkpoint time change - averaged over 5 runs 57

B.1 Real-life - compression ratio change 71
B.2 Real-life - checkpoint time change 72

List of Tables

3.1 Full dataset with equality and range encoding. Full data means all possible combinations of tuples are in the dataset. 13

4.1 Average cardinalities of tables 35

6.1 Average cardinalities of tables 48

6.2 Profiling result of the storage-optimiser 52

A.1 Exploratory results of all datasets and strategies in percentage. Positive number means file got smaller. S1 is strategy 1, "e" or "g" is for "entire table" or "granular" application level 70

LIST OF TABLES

1

Introduction

Database systems apply automatic optimisations to help users efficiently manage data. An important optimisation is compression. It benefits query performance if operated on in compressed form (9) but also saves disk space and thus money if data is stored in the cloud.

Compression algorithms, like run-length encoding (RLE), FOR and PFOR-DELTA, perform better on data that has been sorted (10, 11). They have their performance impacted by two major data layout decisions. The first decision is deciding how the rows and columns are sorted. The second decision is the granularity on which the rows and columns are partitioned. Finding the optimal order of rows or columns for maximum compression is NP-hard (12), and only heuristics that approach the optimal solution have been proposed.

In this thesis, we explore methods for automatically applying reordering techniques to improve compression performance in database systems without severely impacting performance. We then implement the method into the open-source embedded column-oriented database system DuckDB. Since DuckDB uses bit-packing, RLE and dictionary compression, we focus on reordering methods that benefit RLE.

The methods include sorting columns from low to high cardinality (13) and using the combined cardinalities of columns. We also explore more novel techniques such as Vortex and Multiple Lists (14). This exploration is done with the Public BI benchmark (15), which contains user-created data, unlike a synthetic benchmark. Therefore, it is a good benchmark to evaluate the real-life performance of the reordering methods.

Out of these, a method is chosen to implement in DuckDB. Many users do not have the time or knowledge to order the data to optimise for compression. Therefore, the major challenge is applying the reordering method automatically while having a low impact on DuckDB's data writing performance. Finding a method that works for most or all

datasets is the goal. We sort RLE compressable columns before the data is compressed and written to disk (checkpoint). The columns are sorted from low to high cardinality, and rows are sorted lexicographically. We lay a foundation for further research into optimised compression in DuckDB and provide insights into future directions and optimisations.

The main contributions of this thesis are:

- We perform a literature study on the different proposed reordering methods for improving RLE compression and evaluate them with the Public BI benchmark.
- We provide an evaluation of a reordering method in DuckDB and show how it increases RLE compression ratios on real-life data.
- We contribute to the open-source database system DuckDB by writing an extensible foundation to implement reordering methods.
- We give guidance for future optimisations and research directions for optimising compression in DuckDB.

1.1 Research questions

Our main research question is: "How can we apply low overhead automatic reordering techniques to improve compression ratios and query performance of an analytical database system?"

1. How can we best reorder data to optimise RLE compression ratios?
    - How do we automatically identify columns of interest? (e.g. cardinality, column correlation)
2. At what granularity should these optimisations be applied? Moreover, what is their impact on query performance?
3. How do we apply these optimisations automatically, and what is the optimal moment to do this?
4. How do we apply these optimisations in DuckDB?

1.2 Thesis outline

In chapter 1, we have introduced the problem and research questions. Chapter 2 provides the background required to understand the remainder of the thesis. Chapter 3 discusses and comments on related work. In chapter 4, we compare various reordering methods on the Public BI dataset using an evaluation framework made in Python. Chapter 5 outlines the design of the reordering heuristic, discusses the final implementation in DuckDB and any challenges encountered during the engineering work. In chapter 6, we evaluate our implementation in DuckDB and explore optimisation opportunities. Finally, chapter 7 provides a conclusion and guidance for future work.

2

Background

2.1 Row-Store vs Column-Store

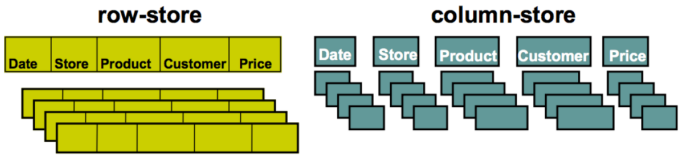

Figure 2.1: Row-store vs. Column-store (1)

Relational database systems have two main ways of storing data, row-wise and column-wise. Traditional databases (e.g. SQLite, PostgreSQL) use a row-store model, in which rows are stored contiguously. This contiguous storage means individual rows are cheap to fetch, but all columns must always be brought into memory (16). Having rows in memory is preferred for transactional workloads since they typically update a few rows. Analytical workloads commonly require a subset of columns but many rows. Columnar stores store the columns contiguously, which means only a subset of columns can be retrieved, saving on disk access and memory bandwidth (16). Since compression algorithms typically work better on data with lower information entropy, column-wise compression can be more efficient (9, 16). The difference can be seen in Figure 2.1.

2.2 Bitmap Index

Bitmap indexes are a storage format for indexes which enable fast querying of large data set (17). It works through bins, in which each attribute is stored in a bin with all its possibilities. An attribute "machine" can have "on" and "off" in a table. When the

machine is on, the attribute "on" will be set to 1 and "off" to 0. If an attribute is a range of values, bins of ranges of values will be created. For example, age will have bins of 0 to 18, 18 to 65 and 65 to 100. A query is then a bitwise OR, and results can quickly be retrieved through this index.

Several compression methods exist specifically because these bitmap indexes can become very large. Byte-aligned Bitmap Compression (BBC) and Word Aligned Hybrid Code (WAH) are two of these methods. Both are similar to RLE but have minor differences to make them more suitable for the compression of bitmap indexes.

2.3 Common compression algorithms in database systems

Compression algorithms can be split into two categories (18). Lossless compression means the exact data can be reconstructed when reversing the compression, also known as decompression. Lossy compression, on the other hand, approximates the original data, possibly losing some information.

Both lossless and lossy compression algorithms can be split as well into lightweight compression and heavyweight compression techniques (11). Heavyweight compression techniques minimise the storage size at the expense of I/O usage and time. Lightweight compression techniques compromise between I/O usage, time and storage size. This compromise makes them more suitable for situations where compression cannot take too long, like in a database management system.

In this section, we discuss several lightweight compression techniques.

Run-Length Encoding

Run-Length encoding (RLE) is a compression method that compresses "runs", or repeating instances, of identical values. For example, 4444222 becomes 4-4, 2-3. First comes the value, and secondly, the amount of times it is repeated. The longer the runs, the better the compression (14).

Bit-Packing

Bit-packing is a technique to represent decimal values in the minimal number of bits required. For example, if we have a list of 10 values, it might be possible to represent 9 of those in 16 bits (i.e. they are lower than 65,536). The last value might need 32 bits to be represented. We can then designate a "bit-header" at the start of each bit. In the simplest form of bit-packing, the bit-header might be 0 or 1. If it is 0, the next sequence

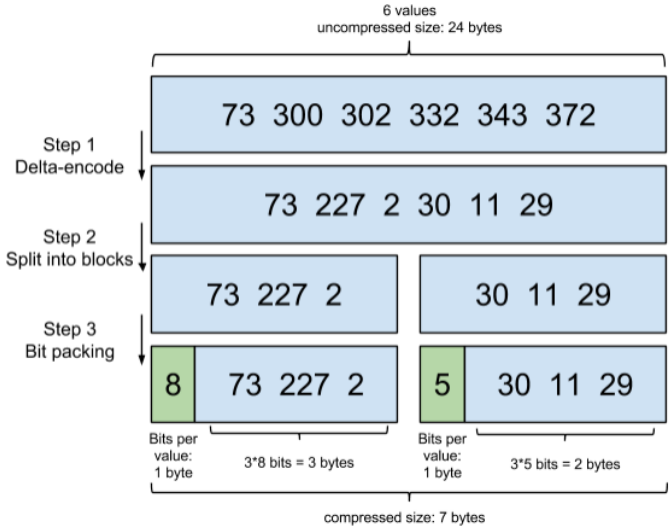

Figure 2.2: PFOR-DELTA and Bit-packing combined (2)

is 16 bits; if it is 1, the next sequence is 32 bits. We therefore save $15 * 9 = 135$ bits by using bit-packing.

Frame of Reference (FOR)

FOR retains a minimum value m (base value) and maximum value M for each numeric column C in a table (19). It subtracts m from each value in the column, which results in a maximum of $[log_2(M + 1 - m)]$ bits per value. For example, if the frame of reference were the value 72, the value 300 would be encoded as 227. This example can also be seen in steps 1 and 2 of Figure 2.2.

Patched Frame Of Reference (PFOR)

PFOR is similar to FOR in that it uses a base value (11). However, the base value must not be the column's minimum. Values lower than the minimum, outliers, are stored as *exceptions*. The difference between the minimum and maximum can therefore be reduced.

PFOR-DELTA

PFOR-DELTA encodes the differences between subsequent values (11). For example, if the current value is 205 and the next is 207, it would be encoded as 2. It also includes the *exception* mechanism as with PFOR.

Figure 2.2 shows an example of combining PFOR-DELTA and bit-packing. In this case, lists that have been compressed with PFOR-DELTA are split into groups. Each group has a bit-header that encodes the maximum number of bits required to represent all decimal values in the group.

Bucketing scheme

The bucketing scheme is a lossy compression method in which data values are placed in buckets. Each bucket has a *minVal* and a *maxVal*, along with a length. A value is added to a bucket in case the difference between *minVal* and *maxVal* is less than 2ϵ. A new bucket is created if the difference is larger than 2ϵ. The values in the bucket can then be approximated (20).

2.4 Partitioning

Partitioning generally means splitting up a larger logical table into physical partitions or files (21). The most basic method is partitioning a table into arbitrary blocks of rows and dividing those over several files. This method is called horizontal partitioning. It helps with dividing the load across the system (22).

2.4.1 Z-Ordering

Z-ordering is a data clustering algorithm that uses a space-filling curve (23). It takes the binary representations of values and interleaves them, following a Z-pattern. The resulting binary representations can then be sorted again. They can then be placed in a binary search tree which can be used for efficient searching as data locality is preserved. It can be further extended to use more than two dimensions. The resulting files/partitions have lower min-max ranges (the minimum and maximum values of that partition) and can be used for data skipping. An example is shown in Figure 2.3 where each block represents a file.

2.5 Column-oriented file formats

There are several data formats designed to store columnar data. The specific implementations differ, but they overlap because columns are stored separately. The rows are partitioned into groups called row groups. Each row group contains a certain amount of rows,

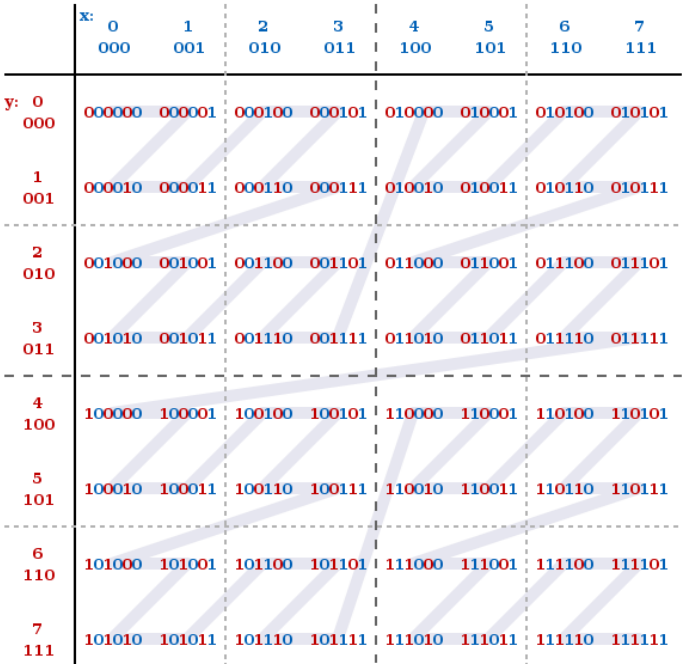

Figure 2.3: Z-curve used for Z-Ordering (3)

and statistics are calculated per row group. The statistics, such as the minimum and maximum values, can be used to "skip" over parts of the data. This skipping increases the efficiency when finding specific data.

We will now discuss several column-oriented file formats, specifically what sets them apart from other formats.

Parquet: The parquet file format is built to allow for efficient compression (24). It supports various compression methods, including Snappy, GZIP, deflate and BZIP2. Figure 2.4 shows a schematic of the file format. It is a column-oriented, binary file format. Data is thus stored adjacently by column, and the rows are separated into row groups. Statistics for the files are kept in the form of metadata. Metadata is kept for each file, each column and the page header. Readers first check the file metadata and then read the column chunks required.

Avro: Avro is a data format designed to use schemas defined using JSON. The schema is present constantly with the data file, which means the data file is self-describing and does not require any outside information (25). The data is stored in a row-oriented format, separated into blocks.

ORC: Optimised Row Columnar (ORC) is a column-oriented file format which is also self-describing. A file is divided into "stripes", which contain index data, row data and a

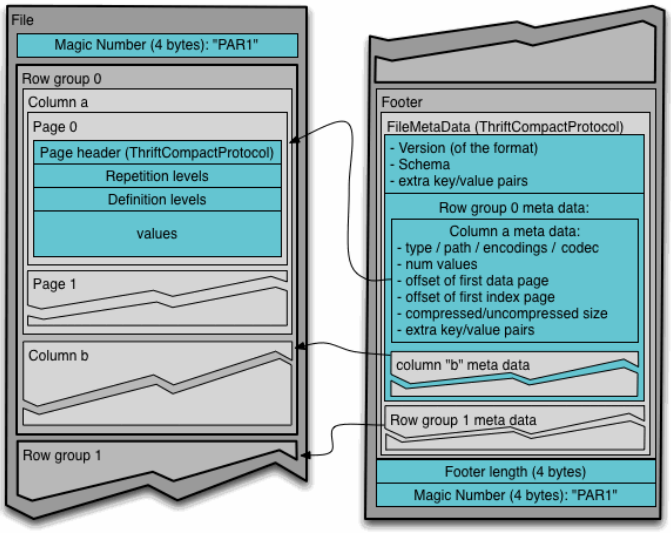

Figure 2.4: Parquet file format (4)

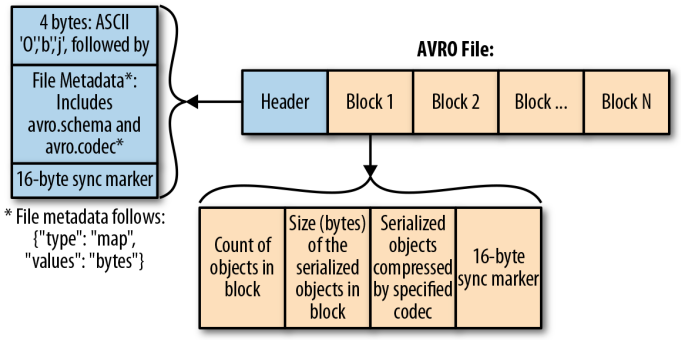

Figure 2.5: Avro file format (5)

footer. The indexes contain the minimum and maximum values for each column (6).

2.6 HyperLogLog

The "cardinality" of a column is the number of unique values. Determining the exact cardinality of large columns can be expensive. HyperLogLog++ is an algorithm which estimates cardinalities up to 1 billion using 6×2^p bits in memory, where p is the precision of the calculation. (26, 27). This calculation is achieved by using randomisation in the form of hash functions applied to each element of a multiset. The hash functions generate a series of 0s and 1s, each having a 50% chance of occurring. The cardinality can then be estimated by finding the hash value with the most leading 0s. With x number of leading

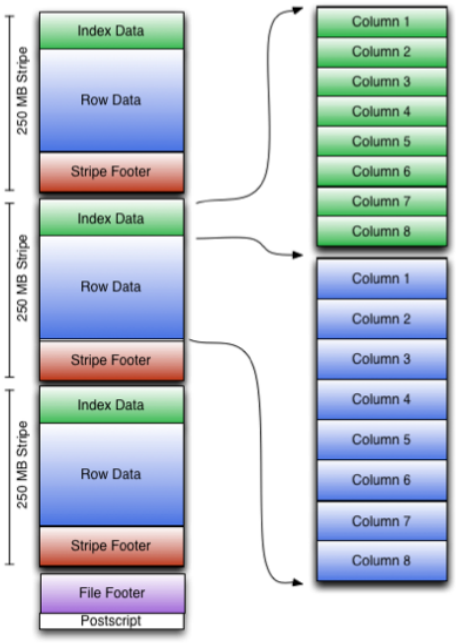

Figure 2.6: ORC file format (6)

0s, the cardinality is estimated as 2^{x-1}. The algorithm divides the hash values into M multiple sets using the first y bits to not depend on a single observation. Each of these sets will have a different number of leading 0s. The final cardinality is given by calculating the harmonic mean of these M outcomes. Finally, the algorithm has a bias to overestimate the cardinality but multiplying the final answer by 0.72134 corrects this bias.

2.7 DuckDB

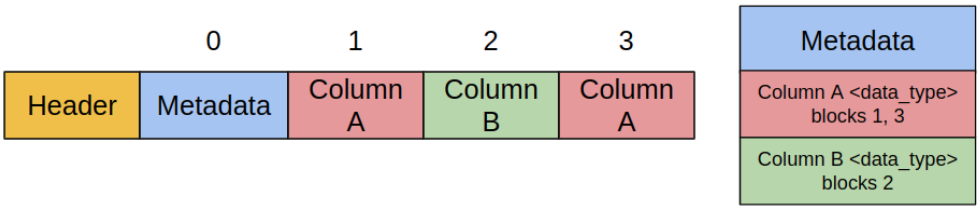

Figure 2.7: DuckDB file format - columns are separated in blocks, with pointers to the specific blocks kept in the Metadata (adapted from (1)).

DuckDB is an embedded analytical column-oriented database system (28). It has its own file format, as seen in Figure 2.7. Each column is partitioned into a block, a slice of

a column of 256KB. The blocks store the minimum and maximum value for that column, making data skipping possible. These blocks are stored in a single file with a 4KB header followed by metadata. The metadata contains information on which block contains what data. The data is partitioned into row groups, consisting of approximately 100.000 rows.

Checkpoints and compression

A database management system separates its work into individual, logical units of work called "transactions". A transaction can represent multiple operations, such as reading and writing, as one operation. The essential properties of transactions are ACID, which stands for atomicity, consistency, isolation and durability (22). Atomicity means that a transaction is performed successfully as a whole (commit) or disregarded as a whole (abort). Consistency means the transaction moves the database from one valid state to another. Isolation means the database system can execute transactions concurrently without interfering with each other. The level of isolation is dependent on the database system. Finally, durability means when a transaction has been committed, its changes are persistent and survive system failures.

Commits are recorded in the "write-ahead log" (WAL), an append-only file on disk, to ensure durability (22). If the database system crashes, previous commits are replayed from the WAL to return the database system to the state before the crash. The WAL is flushed (i.e. emptied) when the in-memory blocks are converted to on-disk blocks.

In-memory blocks are created by appends to the table and are converted to on-disk blocks during a process called "checkpoint". A checkpoint is done on a row group level, which means a change to a row will only require a rewrite of the segment within its row group. The user can manually trigger a checkpoint, or when DuckDB is gracefully stopped, it is performed automatically. A checkpoint will leave empty blocks on the disk in case of deletions and will fill these again during the next checkpoint. The total size of the database file is the sum of used and free blocks times the block size.

DuckDB currently supports RLE, bit-packing and dictionary compression. At checkpoint time, an analysis is performed to determine which compression method is optimal for each column. A score is assigned to each compression method, and the method with the best score is chosen. This assignment is done per column, per row group.

3

Related Work

3.1 Reordering strategies

In this section, we first describe reordering strategies which use Gray codes and lexicographical sorting and compare these strategies. We then discuss more advanced reordering strategies, such as Multiple Lists, Vortex and Genetic algorithms. Then, we describe two papers which discuss considerations when using reordering methods and RLE in practice.

Gray codes & lexicographical sorting

Several papers have studied the effects of using ordered Gray codes or lexicographical sorting on the effectiveness of RLE (12, 13, 29, 30). This work was done in the context of bitmap indexes, of which an example is shown in table 3.1. Bitmap indexes encode tuples in a table as 0s or 1s, depending on the type of encoding used. For example, if a machine is "on", the associated value is 1; otherwise, it is 0. With equality encoding, if a value falls

Tuple	Equality Encoding				Range encoding			
	Machine status		Machine age		Machine status		Machine age	
	on	off	0-5	>5	on	off	0-5	>5
t_1 = (on, 10)	1	0	0	1	1	1	0	1
t_2 = (off, 10)	0	1	0	1	0	1	0	1
t_3 = (on, 0)	1	0	1	0	1	1	1	1
t_4 = (off, 3)	0	1	1	0	0	1	1	1

Table 3.1: Full dataset with equality and range encoding. Full data means all possible combinations of tuples are in the dataset.

into a bin, only that bin is marked with a 1. In range encoding, if a value falls into a lower bin, all bins above are also marked with a 1. Since bitmap indexes produce runs of equal values, they were a prime candidate for studying reordering methods to create longer runs and thus better compression.

Pinar et al. (29) propose to use sorted Gray codes to compress bitmap indexes. A Gray code encodes a sequence of numbers where the adjacent numbers have only a single digit differing by one. The Gray code algorithm takes a Gray code, for example (0,1). It then writes it forward and the same code backwards, making (0,1,1,0). It then appends 0 to the beginning of the first n numbers and 1 at the beginning of the last n numbers. This makes (00,01,11,10). This appending can be done again to create binary Gray codes: (000, 001, 011, 010, 110, 111, 101, 100). Generating the Gray codes in this manner is called the reflection technique because codes are "reflected" in the middle. Tuples in the table are converted to Gray codes and then placed in ascending order according to their Gray code. For example, the Gray code sequence (0001, 0010, 0101, 1100, 1110, 1011) is Gray code sorted as the Gray codes correspond to the values: (2, 4, 7, 9, 12, 14).

Apaydin et al. (30) compare the Gray code ordering against lexicographical ordering of data for RLE in the context of bitmap indexes. They use a dataset which contains all possible combinations of tuples and call it "full data". An example of "full data" can also be seen in table 3.1, where a "full" dataset (all combinations of tuples are present) is shown for equality and range encoding. From their experiments, they conclude that lexicographical sorting and Gray code ordering achieve the same compression ratio for equality encoding. If range encoding is used, Gray code ordering results in better compression than lexicographical sorting.

In 2003, Oracle engineers proposed sorting before loading a table to achieve better compression (10). They state that reordering columns with a medium cardinality and many rows results in better compression than sorting on low or high cardinality columns. Lemire et al. (12, 13) leverage this idea and show, through experiments on real-life datasets, that sorting rows lexicographically and ordering the columns from low to high cardinality improves the compression ratio of these datasets. They compare this to the Gray code ordering and conclude that lexicographical sorting results in better compression ratios. Following up on this, Pourabbas et al. (31) find that if columns have equal cardinality, sorting them from high to low skewness results in an increase in compression ratio as opposed to the natural order.

Multiple Lists

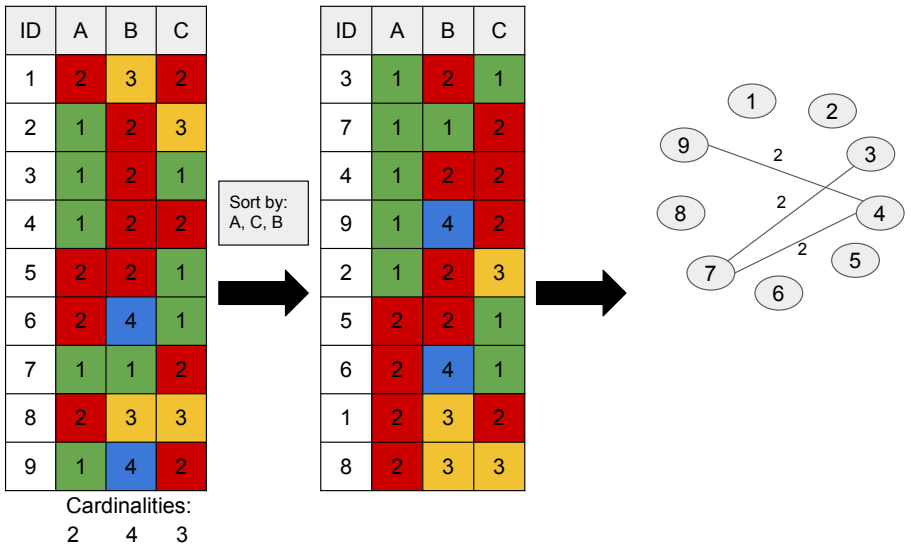

Figure 3.1: Multiple Lists algorithm. First sort the table by a column order, then connect each row with the row above and below by Hamming distance (only 3 connections are shown for clarity). The final sort order is created by shortest path of Hamming distances.

Multiple Lists (14) uses a sparse graph to keep track of all possible ways of reordering a table. The multiple lists algorithm starts with a table with several columns. For example: A, B and C. The columns are then rotated cyclically to create in this case three lists, A, B, C / B, C, A / C, A, B. Each time a new list is created, the table is sorted according to this order. For each list, an edge between nodes is created between the rows above and below in the lexicographical order. An example is shown in Figure 3.1, where row 3 is connected to 7, and then 7 is connected to 4, which is connected to 9. The value of the connection is the Hamming distance (i.e. the number of values that differ between rows). Once these connections have been made for all column orders, the final tuple order will be generated by traversing the graph from a random node using the nearest neighbour algorithm and removing the nodes that have been traversed. A downside of this method is that the tables have to be sorted several times. For larger tables, the authors propose using partitioning to run the above algorithm parallelised.

Vortex

The Vortex algorithm (14) is based on the frequent-component, where a table is sorted by frequency triples. Vortex is inspired by the Gray code order and *bit interleaving*. Bit

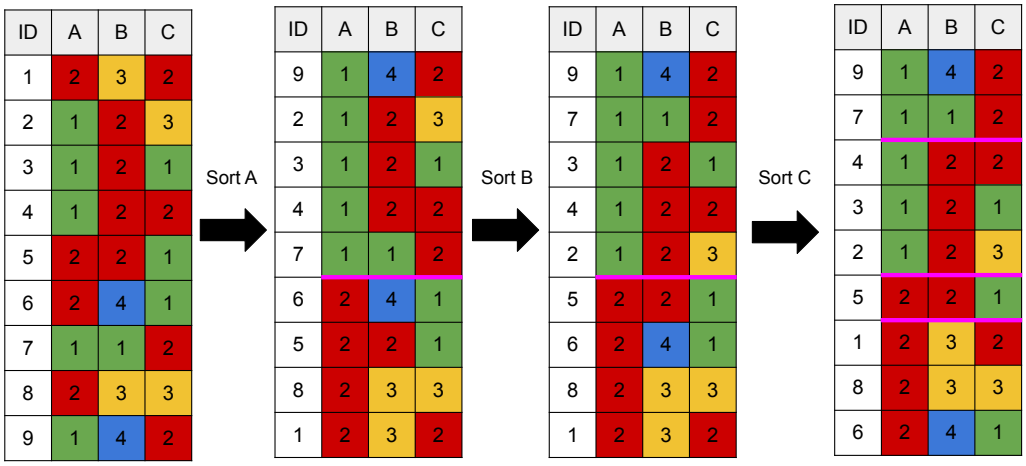

Figure 3.2: Vortex algorithm. Column A is sorted based on its frequent component 1. Column B is sorted with its frequent component 2; the order is reversed in the top partition, which connects the 2s in the middle. This is repeated for column C but with two extra partitions.

interleaving means that first, the most significant bits of two tuples are compared before comparing the less significant bits. The algorithm starts with a list of tuples. First, the most frequent value from the first column is computed as x^1, and all tuples starting with this value are selected. They are put first. Then the most frequent value from the second column is computed as y^1. Among tuples with x^1 as their first value, all tuples with y^1 as their second value are placed last. The remaining tuples are placed first if they have y^1 as their second component. This sequence is repeated until the whole table is sorted.

An example is shown in Figure 3.2. The most frequent component in column A is 1. This 1 is placed first in the order. In column B the most frequent component is 2. In the first partition, it is placed last. In the second partition it is placed first, connecting the 2s in the middle.

Experiments are performed on six real-life datasets and six compression algorithms. Vortex is three times slower than lexicographical sorting and Multiple Lists four times. This slower time is also because each algorithm requires the data to be sorted. Vortex was similar in performance to lexicographical, especially if RLE compression is considered. In some cases, it was 10 to 20 percent better, other times worse. Multiple Lists had similar results, although it generally did not perform worse than lexicographical sorting. Some guidance is offered on when to use the algorithms in real-life situations. Two values, ω and p can be calculated to determine whether compression will benefit from using Vortex or Multiple Lists.

In some experiments, Vortex and Multiple Lists improve RLE compression but can also worsen the compression. Furthermore, they are 3 to 4 times slower than lexicographical sorting. Vortex improved RLE compression the most on average and could be useful for our work. The ω and p metrics could be used to determine when it is worth spending extra time reordering the data with Vortex or Multiple Lists, as not every table's compression benefits from these algorithms.

Genetic and meta-heuristic algorithms

Jovanovski et al. (32) explore a genetic algorithm based on natural evolution and chromosomes. The optimal solution is found by crossing over chromosomes and mutation upon crossover. Only the best individuals in the population are retained when crossover happens, a concept called "elitism".

Experiments were performed using this algorithm. The initial population was created randomly and with a heuristic for sorting the columns by cardinality. It is compared against sorting the columns by increasing cardinality.

The genetic algorithm outperforms sorting by increasing cardinality on synthetic and realistic datasets in all cases. However, the genetic algorithm is more computationally expensive and thus slower than sorting by increasing cardinality. The authors claim this is not an issue in a data warehouse since the compression is typically done once.

As a follow-up to their previous work (32), Jovanovski et al. (33) explore other meta-heuristic methods to maximise the rate of RLE compression. These meta-heuristic methods are six algorithms inspired by natural processes. The goal of the authors is twofold. They build an RLE compression tool which they use to implement the algorithms, and they use this to evaluate which algorithm works best for their use case. Each algorithm has a solution assigned through a fitness function and represented by an abstract class. The initial solution is randomly generated. The objective is to minimise the total number of columnar runs in a table.

The authors then describe each of the six algorithms in detail.

1. Genetic algorithm (GA): based on natural evolution. The solution is represented through a chromosome. Selection, crossover and mutation are used to determine new solutions. Additionally, replacement is used, which means that only the best individuals of a population are retained in a new population.

2. Cuckoo search (CS): combines cuckoo birds' parasitic brooding behaviour and Levy flight behaviour. The authors also employ an optimisation on the original algorithm, which uses three types of cuckoos, which has faster convergence to the minimum.

3. Bat algorithm (BA): inspired by the echolocation behaviour of bats. If "bats" are close to the solution, they make less aggressive moves than if they are far away from the best-found solution.

4. Particle swarm optimisation (PSO): based on particles "flying" through a solution space to get close to the best solutions found. Each particle has momentum and cognitive and social components, which are the parameters to optimise.

5. Simulated annealing (SA): based on atoms and thermodynamics. It starts from a single solution and visits the current neighbours to evaluate. The algorithm has a probability of accepting a worse solution in an attempt to escape local optima. As the algorithm runs, this probability decreases to settle on a solution.

6. Tabu search (TS): escapes local optima by keeping a list of forbidden moves. The algorithm visits k neighbours and chooses the solution which performs best on the fitness function and contains no forbidden moves.

The authors then perform experiments on synthetic datasets. The three best-performing algorithms are chosen: SA, GA and CS. A sensitivity analysis is performed to find optimal parameter values, after which the algorithms are tested on real-life datasets. Along with that, the authors make changes to the way random initialisation is performed. They prioritise low cardinality columns for sorting; low cardinality columns are randomly sorted in ascending or descending order, and high cardinality columns are not sorted. The columns are then sorted by the number of increasing cardinality. The authors note an increase in compression ratio and speed with these optimisations.

The authors find that using the SA, GA and CS algorithm and presorting can positively affect RLE compression. Unfortunately, they do not compare against naive compression without sorting. From the paper, it can be concluded that using algorithms inspired by nature to optimise RLE compression is a suitable method if the time to optimise is available. The question is also whether the proposed methods are applicable in database systems. The experiments typically lasted over 500 seconds. The algorithms would only be practical if data is to be stored for a more extended period, it would be hard to use it regularly while a database is in use.

3.1.1 Reordering in practice

Column partitioning

Shi (34) proposes a method of partitioning columns into two sets to find a balance between compression time and ratio after reordering. The primary set and the secondary set. The primary set will be compressed by RLE and will only contain columns which benefit from this compression. The secondary set contains columns where RLE will have the opposite effect. Thus the algorithm attempts not to consider all columns but only a portion. The lexicographical sort, as previously proposed, is used. The author compresses the columns by increasing cardinality until the size benefits of compressing high cardinality columns begin to decrease (i.e. RLE compression increases the size). The author calls this the "turning point".

The author shows that compressing only part of the columns can be beneficial. However, the comparison to other algorithms in the field is lacking. In the experiments, the algorithm is only compared against compressing none or all columns. The experiments could have been extended to include different methods or ideas instead of one algorithm.

Recovering original positions

Lee and Chung (20) propose an improvement to RLE by reordering data in systems where the original position of the data needs to be recovered. Time series data is an example of this. To reconstruct the original position of the data, one must use a mapping table. The mapping table stores movement information but takes up much memory. The authors propose a method to store this movement information compactly.

While the data is being reordered, each movement is recorded in «start, newStart» format. To recreate the original list, the data is traversed reversely. The algorithm checks the newStart position for the required values and moves it to the old start position. This check is done for each movement. Figure 3.3 shows an example. The top part of the figure shows elements being swapped to create a sorted table and their movement information recorded. Notice that, to save space, movement information is dropped. For example, if position 2 is swapped with 3, position 3 automatically moves to 2; thus, this information does not have to be recorded.

For RLE, the compression improved by approximately 30%. Movement dropping added another 3%. However, the experiments have a caveat. The method does not consider multiple columns, and the reordering technique focuses on sorting one column.

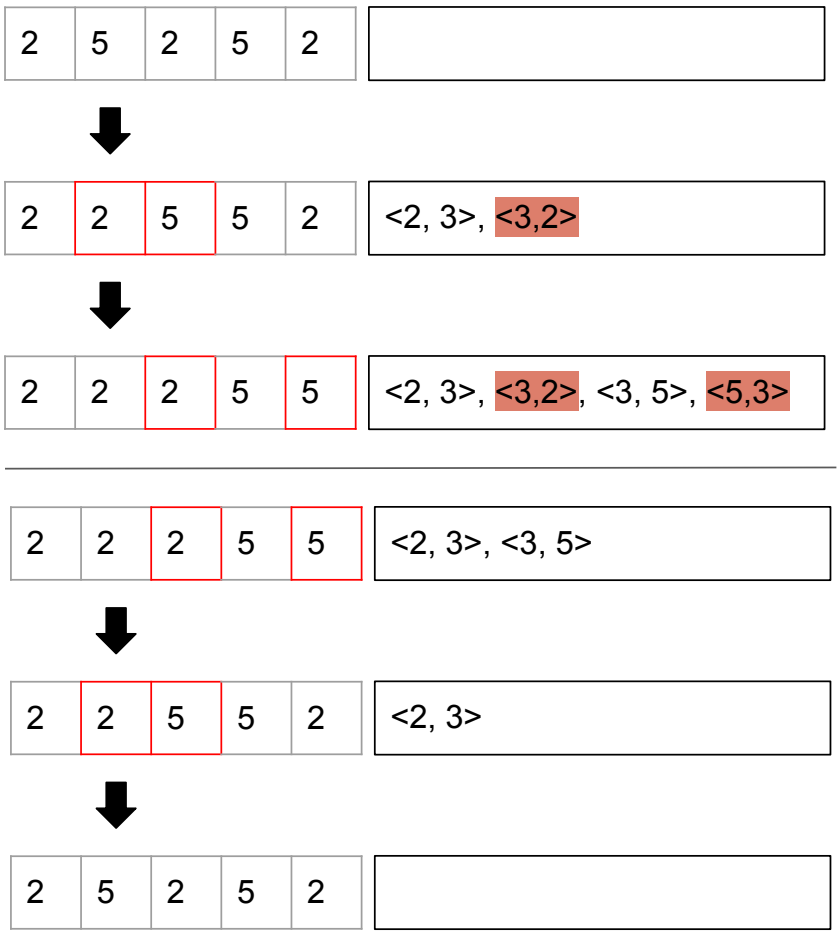

Figure 3.3: Mapping table with movement dropping. Swaps are kept in a mapping table with format «start, newStart». To recreate the original table, this is traversed backwards. Redundant movement information is dropped (indicated in red)

When implementing these algorithms, the movement dropping and neighbour merging could be left out. They do not result in significant gains in their experiments, especially in large datasets, but add complexity. A mapping table to reconstruct certain data types could be helpful in future versions of our work.

3.2 Self-organizing data & machine learning

Self-Organising Data Layouts

In this thesis, Tufa (35) outlines an implementation of self-organised data in Databricks Delta to improve query efficiency on real-life data.

Databricks is a product which brings Spark to the cloud and decouples compute from storage. It also contains several optimisations for query performance and storage space. Databricks Delta introduces an additional storage layer which provides ACID transactions and is stored in a Parquet file. The implementation focuses on using the file statistics of these Parquet files to automatically optimise data layouts by providing more opportunities for data skipping. The optimisation process was already implemented in Delta through Z-Ordering, but it was still a manual process requiring a human to start it.

The primary implementation enriches the existing file statistics for Z-Ordering and SOP. SOP is used to cluster data on a given workload using predicates that evaluate to a boolean (36). The statistics are computed after clustering upon writing the files to storage and are extracted from the current workload. Possible features to cluster on are collected for both Z-Order and SOP and pruned according to several metrics. Finally, correlation in the data is considered. When sorting one column, for example year, another column could be automatically sorted along with it. For example, volume sold in that particular year. In this case, the columns are treated as one. These correlations between columns are calculated using CORDS (37). Once the features have been extracted, they are evaluated on suitability by calculating the number of queries they cover and their degree of correlation with other features.

The solution was implemented on a single compute cluster, where there is no need to consider concurrent operations. The automatic optimisation is considered after each insert query when writing to storage and is only performed on new and unclustered data. Before performing the optimisation, a cost-benefit analysis is done. The cost function estimates the cost based on data size in GB, which is compared against the benefit of the reading costs after optimisation. If the result is positive, the optimisation is performed. The benefit estimation is efficiently done due to only using a sample of the data for the computation.

The final benchmark results show that the implementation significantly improves query time compared to a manual Z-Ordering optimisation. However, the insert time is higher than before due to the overhead. The author indicates that the cost-benefit analysis is the key to the increase in performance.

Self-Organising Data Containers

The increasing movement toward the cloud has resulted in data services which are no longer monolithic but disaggregated. Most services have a storage layer (e.g. Amazon S3) and systems to interface with the data (e.g. SQL). The disaggregation means the components can be managed separately but can harm performance. File formats like

Parquet do not retain performance-related metadata. The authors claim if they retain more of this metadata, such as histograms and data access patterns, query performance can be increased by optimising the file layout. That is why Madden et al (7) propose a self-organising storage format called "self-organising data containers" (SDC). It stores various physical representations of data (i.e. summaries, aggregates, columnar and row-oriented layouts) in a single data object. Furthermore, the data object transforms over time based on the user's access patterns to find the "optimal layout", which increases query performance.

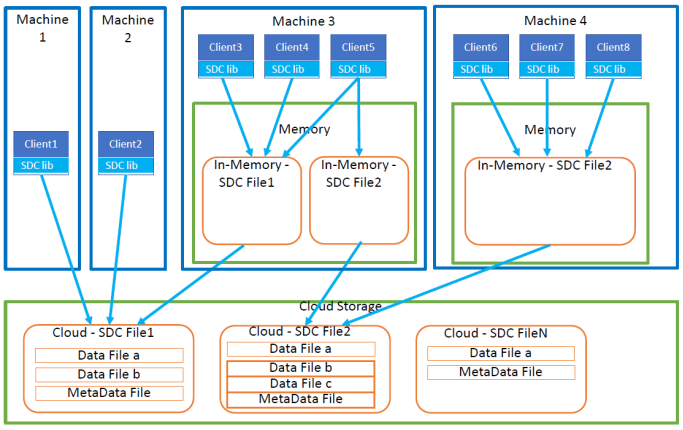

Figure 3.4: SDC Architecture (7)

The SDC architecture is shown in Figure 3.4. It can exist as in-memory storage on a local machine or as storage in the cloud. It can be used by multiple users simultaneously and can exist as one mutable file or a collection of files. The basic structure is as follows:

- Data blocks: read-only and variable-sized
- Index blocks: references to data blocks
- Meta-data blocks: summaries of access patterns and data statistics of blocks

As the SDC is accessed, users write their access patterns to the meta-data blocks. This meta-data can then be used for self-organisation. The SDC can create new indexes or modify existing ones to extract more performance from queries. It can also be the other way around, where an index is deleted to reclaim space.

The paper is highly conceptual, and the prototype used for the evaluation is implemented using Parquet files. It records the meta-data about read operations. The layout optimisation layout has to be called explicitly, using range partitioning or qd-tree blocks. The

prototype is evaluated against Delta Lake's OPTIMIZE ZORDER BY command, which uses Z-ordering. The authors find that their implementation outperforms Delta Lake.

While the concept of SDCs is interesting and the results promising, there is no clear description of the optimisations performed. The range partitioning and qd-trees are not explained, so it is difficult to grasp why these optimisations work so well. Furthermore, there is little consideration for how the ideas would work in a production system. The authors regard this as future work.

Machine Learning in databases

Towards instance-optimised data systems

Kraska (38) provides an overview of the state of applying machine learning (ML) to improve database systems. The main reason for applying ML is to have the system self-adjust to a particular use case and workload to provide near-optimal performance. However, this is not feasible to do manually because of time constraints. A basic form is optimising the parameters of a system for a particular workload. It often requires training an algorithm on a workload and then using a reinforcement learning algorithm to make continuous adjustments. The author calls this "knob and design tuning", whereas an instance-optimised system goes far beyond only modifying parameters.

Instance-optimized optimised database components can be designed in three ways. *Design continuum* uses a machine learning model to configure a traditional algorithm. *ML-enhanced algorithms* use an oracle to make a prediction, which can make an algorithm more efficient. *Full-model replacement* swaps an algorithm out with a model, for example, to estimate cardinality.

The author then provides a progress report on their "learned indexes", a form of ML-enhanced algorithms that can optimise B-trees, hash-maps and Bloom filters. New research into these learned indexes reveals that the smaller index sizes can improve query performance on disk-based systems but is not always better than newer index variants such as the radix-based binary search and the Compact HisTree. The author highlights recent research into learned Bloom filters/hash-maps, multi-dimensional indexes, and storage layouts.

Another large area is learned query optimisation. Improving cardinality estimation has been a focus of several papers. However, these papers do not provide evidence that improved cardinality estimation leads to better query plans and can even cause queries to be slower. "Bao" is an optimiser which takes an existing query optimiser and uses reinforce-

ment learning to activate/deactivate features on a query-by-query basis. According to a study done with Microsoft, it can improve runtime latency for complex queries by 90%.

Finally, an overview of SageDB is given, a prototype instance-optimised database system that intends to combine the previously described techniques. It is developed as an accelerator for PostgreSQL and, as of July 2021, works on a single table. It automatically performs block-layout optimisation, automatic replication, encoding and partial materialisation. The system had a latency improvement of 2x up to 8x as compared to PostgreSQL and a cloud vendor's database, tuned by an expert in those systems. However, there is a caveat. Cloud vendors often prefer consistent performance. With SageDB, changes in the data structure can result in a different configuration; thus, the same query has a different performance. A possible solution could be to optimise in such a way that performance degressions are minimised, using a weighted loss function.

While using machine learning to optimise database system components can significantly increase performance, robustness remains an issue. Furthermore, creating models can be a time-consuming process. The area is still developing but provides many research opportunities for optimising database systems.

LEA

LEA (39) is a machine learning component for database systems which can predict the optimal encoding (i.e. compression method) for a given column. It can be optimised for size after compression or for query latency. LEA uses three models. One model to predict the encoded size and two models to predict the scan time of a column. Training is done by generating slices of data (one column of a block of data). The range, cardinality and the first three moments of the distance between adjacent values in the slice are used as features. 1255 synthetic datasets of 1 million values each are used, where a contiguous sample of 1% is selected for training. Loading the data took three hours, but training was done within a minute. A random forest regression and linear regression model were used.

LEA was compared to a heuristic-based encoding advisor, C-Heuristic, of the anonymous database system "System C". C-Heuristic was not explained further, except that it was optimised for encoding size. Testing on a dataset with 12.5 million rows and 24 columns showed that LEA is better than C-Heuristic at optimising encoding size and query latency. The same was found with TPC-H. The authors report that applying LEA to TPC-H takes 10 minutes for the entire process, including gathering statistics and loading the data. However, they do not report the performance of the C-Heuristic.

The authors have shown that using machine learning models to optimise query performance and encoding size can positively affect non-machine learning heuristics such as C-Heuristic. With LEA, determining the encoding takes 10 minutes, while brute-forcing it takes 30 minutes. However, the performance of C-Heuristic is not given.

DeepSqueeze

DeepSqueeze (8) is a machine learning model which can capture relationships between tabular data in a column-store to compress them, also known as semantic compression. In general, semantic compression is limited to finding associations between two columns. DeepSequeeze aims to capture relationships between multiple columns. Its primary focus is minimising the overall data size for long-term data archives at the cost of the overall runtime.

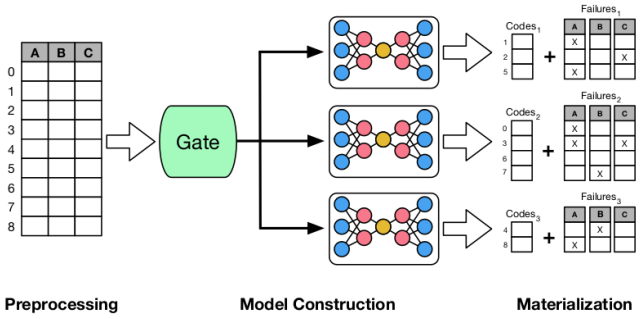

Figure 3.5: DeepSqueeze compression pipeline (8)

Figure 3.5 shows DeepSqueeze's compression pipeline. The first step is preprocessing. It takes tabular data and applies transformations depending on the column type. Categorical columns with strings are transformed into integers, in which a value represents each category (i.e. dictionary encoding). Numerical columns with integers or floating-point values are normalised to a range between 0 and 1. They are compressed in a lossy way by predicting the values upon decompression with a specified error threshold. To achieve this, the model is trained on the midpoints of "buckets" of data.

The model is a neural network in the form of an autoencoder. It maps a dataset to a lower-dimensional representation using an encoder and can reverse this transformation using a decoder. Numerical columns consist of one node in the network, whereas categorical columns have a node per distinct value. A shared output layer is used for all categorical columns to avoid an explosion of model size. The size is bounded by the column with the

highest number of distinct values. The model architecture consists of two hidden layers, with twice as many nodes as the columns in the input data.

To keep the size of the model small, a mixture of experts is used. Several smaller models are trained for specific tasks, known as a "sparsely gated mixture of experts", where a "gate" assigns tuples to the expert with the highest accuracy. During training, Bayesian optimisation searches for the best combination of hyperparameters. After each trial, the best combination of hyperparameters is predicted. The goal is to minimise the size of the compressed output.

The final step is to materialise all components required for decompression. The decoder, the codes (i.e. the compressed input), failures (i.e. input that was not compressed) and the expert mappings are materialised. A few techniques are applied to reduce the size of these components.

The decoder is compressed with gzip and stored with a width of 64 bits, often larger than required. The codes are truncated in increments of one byte until the number of failures increases. Failures are the largest contributor to size and are compressed by writing them to a Parquet file. Categorical columns are also compressed using more conventional techniques, like Huffman coding or RLE. Numerical columns are compressed by encoding them as the difference between predicted and actual values. The expert mappings are stored together and compressed with RLE.

The model is first evaluated against gzip and Parquet. It outperforms these in compression ratio on six datasets by between 5% to 20%. However, it should be noted that DeepSqueeze uses lossy compression, as opposed to gzip and Parquet, which are lossless. The authors state that the runtime of the compression is within 2x of Parquet and gzip. This claim is valid without considering that hyperparameter tuning takes up most of the time and must be done for each dataset.

DeepSqueeze is then compared against Squish, another semantic compressor. The improvement in compression ratio is around 5% over Squish, on average. This result is for an error threshold of 0.5%, and the compression ratio improves as the error threshold is raised.

DeepSqueeze improves the state-of-the-art semantic compressor Squish. It will be interesting to see whether the runtime performance of these semantic compressors prevents more general usage of them. They are useful for long-term data archival since they focus on achieving the best compression but are less relevant when the runtime is essential or when lossy compression is unacceptable.

4

Comparing data reordering methods

Since implementing data reordering functionality in DuckDB will be a significant engineering effort, we first assess the viability of column and row reordering strategies on real-life data through Python experiments. We are focused on improvements to compression ratio and comparing the relative performance of reordering strategies. Specifically, we want to answer the following questions:

1. Which strategies improve the RLE compression ratio compared to the natural order of the data, and how do their processing speeds compare?
2. How does the compression ratio change when strategies are applied granularly (i.e. to a group of rows)?
3. Are there metrics to predict the performance of a strategy?

4.1 Method

4.1.1 Public BI

We performed experiments on the Public BI benchmark (15). It contains data extracted from 46 Tableau workbooks saved in CSV files and compressed with BZIP2. It is real-life data created by Tableau users, making it more suitable for evaluating real scenarios than synthetic benchmarks. Furthermore, it is very diverse in its content and data types, making it a good candidate for evaluating compression strategies.

We had to make preparations to use the Public BI benchmark. First, since DuckDB only supports reading files compressed with GZIP, we converted all BZIP2 files to GZIP. Second, the original Public BI benchmark did not contain any results, only the data and

the queries. Furthermore, many queries did not contain an ORDER BY clause, which meant we could not evaluate the correctness of the results, nor could we reproduce results as they would be in a different order each time. We enriched the benchmark by adding the ORDER BY keyword to each query. We have made the benchmark available on Zenodo [1].

4.1.2 Cluster machine

The experiments were performed on a cluster machine with an Intel Xeon Gold 5115 CPU running at 2.4GHz. It has a Turbo Boost to 3.2GHz with 20 cores and 40 threads. The CPU has an L1d and L1i cache of 32KB per core. The L2 cache is 1024KB per core, and the L3 cache is 1.375MB per core. The machine has 384GB of RAM and is running Fedora Linux. The data is stored on a 1.8TB NVMe drive.

4.1.3 Experimental setup

The implementation was done using Python 3.8.6, Pandas 0.25.3 and the DuckDB Python package (0.3.2). All columns, except those with VARCHARs, will be compressed with RLE since DuckDB does not support RLE compression with VARCHARs. We created an extendable AbstractOracle class used as the basis for implementing all strategies. Its *compress* method takes a table and creates a temporary DuckDB database. Then sorts by the "key" columns, which differ per strategy. Each strategy is a class, inheriting from the AbstractOracle class. The reordering is performed using the ORDER BY clause when data is inserted into the table. The data is loaded into a Pandas dataframe and then "registered" as a view in DuckDB. It then closes the connection to the DuckDB database, causing DuckDB to compress all columns of the table. A .db file is produced, which is compared against the size of the .db file before reordering.

Finally, we created an evaluation script to run multiple techniques at once. Since the Public BI benchmark consists of multiple datasets, we loop through them and perform the desired oracle on each one. The results are then saved in .csv format for later analysis.

4.2 Experiments

The evaluated reordering strategies are shown below. They were inspired by, and chosen from, the related work. Various strategies exist, as covered in section 3.1. Since our goal is to implement the algorithms in a production database system with strict requirements

[1] `https://zenodo.org/record/6277287` & `https://zenodo.org/record/6344717`

for time performance, we have disregarded strategies with known high overheads, such as Multiple Lists and genetic and meta-heuristic algorithms. These methods were at least four times slower than lexicographical sorting, making them unsuitable for real-time reordering.

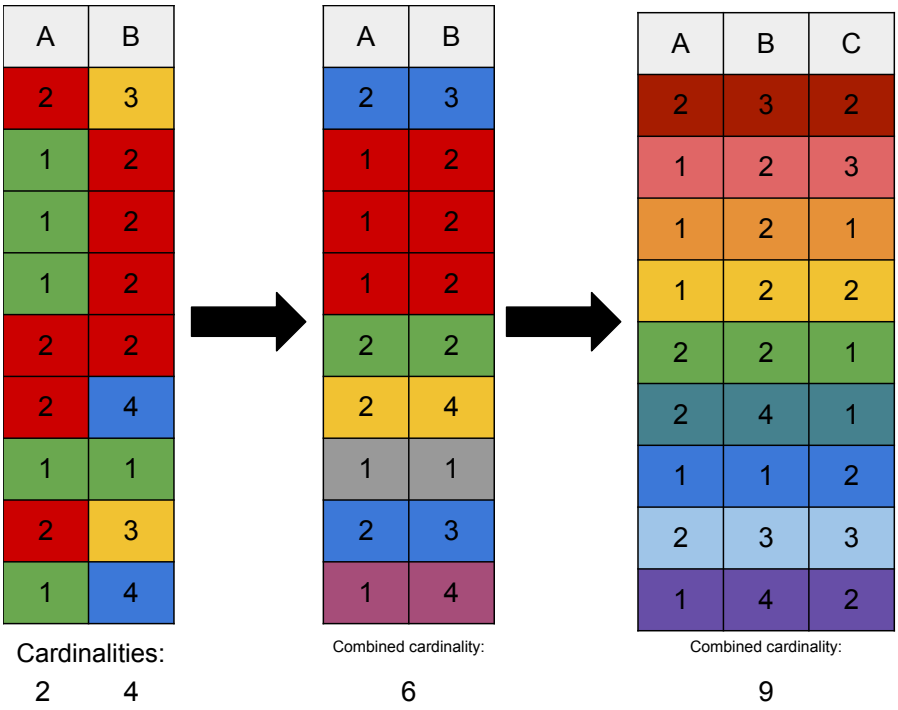

Figure 4.1: Combined cardinalities of multiple columns with sorted order. Columns A and B have a combined cardinality of 6. Columns A, B, and C have a combined cardinality of 9. We search for a sort order before the combined cardinality sharply increases (e.g. A & B).

We have also evaluated a novel strategy by calculating the combined cardinalities. Correlation between columns has been used to improve data skipping using CORDS (35). However, we wish to find a more efficient way to determine if columns are related. If column x has ten unique values and column y has ten unique values, then they are closely related if the cardinality of column x and column y combined is ten as well. The cardinality remaining the same means each row had the same value in both columns. Sorting these columns together could therefore result in better compression in both columns. We use this intuition to find a combination of columns with the lowest "combined cardinality".

Figure 4.1 shows an example of this strategy. Column A and B have a combined cardinality of 6, meaning there are 6 distinct rows. Column A, B and C have a combined cardinality of 9. Therefore, we might choose to sort the table using column A and B which is before the combined cardinalities increased by a large amount.

We will now list the evaluated reordering strategies with a brief explanation.

Reordering strategies

Strategy 1 - Cardinality: Sort the rows lexicographically from the lowest to highest cardinality column.

Strategy 2 - Cardinality and skewness: Sort the rows lexicographically from lowest to highest cardinality column. If columns have the same cardinality, sort by high to low skewness.

Strategy 3 - Combined sorted order: Determine the cardinality of all columns individually. Then determine the cardinality of the lowest cardinality column and the second-lowest cardinality column combined. Keep adding columns to the sort order, in order, until the result consists of more than 30% of the rows in the database. 30% was chosen as an arbitrary value. Intuitively, the cardinality should not be high when all columns are closely related.

For example, if the respective cardinalities of A, B and C are: 2, 4, 3. Then we first order this list from lowest to highest cardinality: A, C and B. Thus, we determine the combined cardinalities of A and C. Finally, we determine the cardinality of A, C and B combined.

Strategy 4 - Combined natural order: Do not determine the cardinality of the individual columns. Instead, iterate through the columns in their insertion order and determine the cardinality of column combinations. Stop adding the columns to search when the combined columns consist of more than 30% of the rows in the database.

Figure 4.1 shows an example of this. First, calculate the combined cardinalities of A and B. Then, the combined cardinalities of A, B and C. Notice that we do not calculate the individual cardinalities first, as we did with strategy 3.

Strategy 5 - Vortex: Sort columns from low to high cardinality. Select the tuples with the highest frequency and add them to the final order. Repeat for the second column, and connect them with previously added tuples by reversing the order. (Refer to section 3.1 for an in-depth explanation)

Since all strategies require knowing the cardinality, or combined cardinalities, of columns, we must have a reliable and fast way of calculating them. As detailed in section 2.6 the

HyperLogLog algorithm is used. We applied it to a sample of 30% of the data. Cardinality estimations based on a sample of high cardinality columns cannot guarantee a low error (40). However, we are interested in ordering low cardinality columns, which have a reasonable error. Therefore, we have chosen to use a sample to speed up the estimations.

4.3 Results

This chapter contains figures with summaries of the results. The full results can be found in Appendix A.

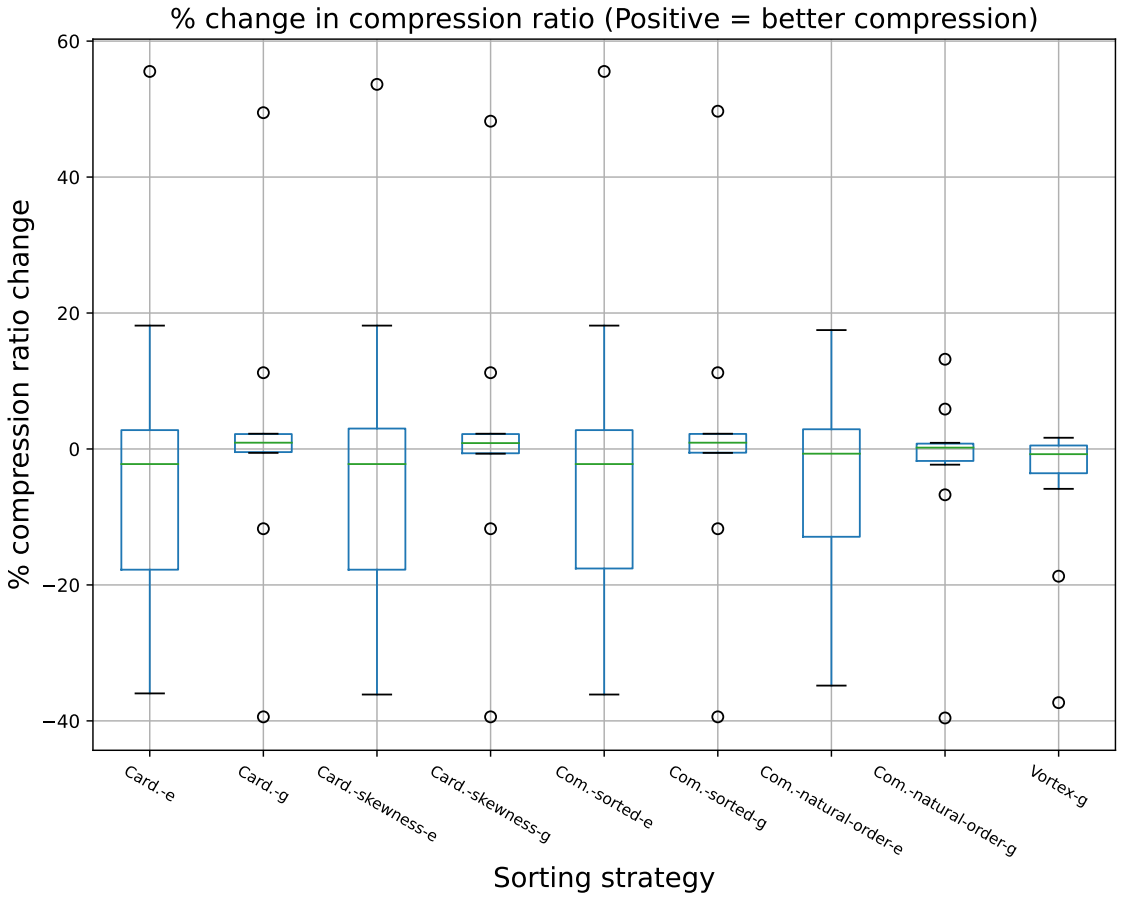

Figure 4.2: Public BI compression ratios. All reordering strategies, sorting the entire table or per row group (granular) on 11 datasets. "E" for "entire table" and "g" for granular sorting

Figure 4.2 shows the compression ratio compared to the data's natural order for all strategies. The granular experiments apply the reordering to 100.000 rows at a time, the approximate row group size in DuckDB. Figure 4.3 shows the reordering times. Both

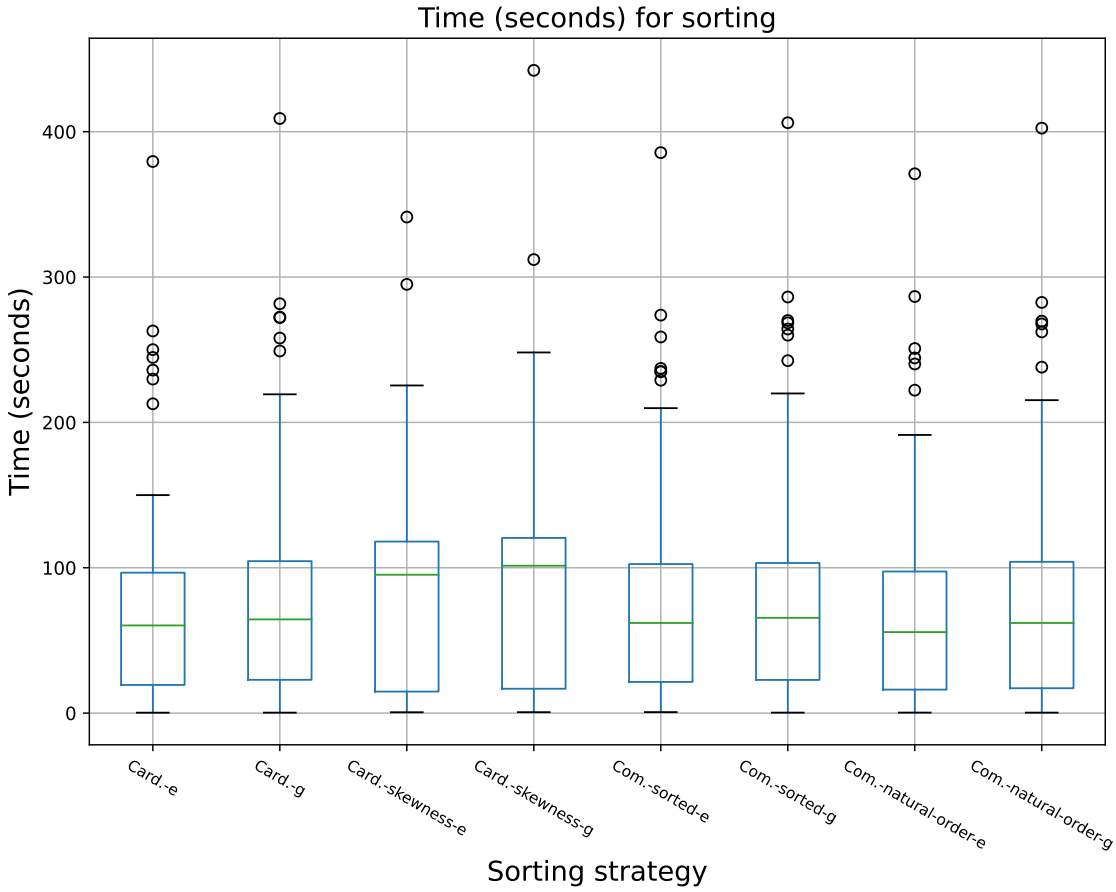

Figure 4.3: Public BI reordering times. All reordering strategies (except Vortex), sorting the entire table or per row group (granular) on 45 datasets. "E" for "entire table" and "g" for granular sorting

figures were generated using 11 of 46 datasets due to the memory and time requirements of the Vortex algorithm. Using only those datasets ensures a fair comparison.

We first discuss strategies 1 to 4, as they are similar in their reordering method. We can observe that reordering on the entire table results in a greater spread of compression ratios than granular reordering, with almost 75% of the results being negative, meaning the file size increased. Reordering on a granular level shows a tighter spread of results but with more outliers. The median is just above 0.

Comparing reordering methods, we notice that strategies 1, 2 and 3 were similar, while strategy 4 performed slightly worse. Especially during granular reordering, strategy 4 had more results below 0%. For strategy 5, we can observe that it increased the compression ratio on less than 25% of the datasets, but with more outliers toward the negative.

If we compare the reordering and compression running times in Figure 4.3, there are

few differences between strategies 1 to 4. We can see that reordering the table granularly means a higher median. This increase is expected as the table is loaded in portions, adding additional overhead. We observe that reordering by fewer columns, which strategies 3 and 4 attempt to achieve, does not result in a significant difference in running time. Strategy 5, Vortex, is not contained in this figure. The average time to run the benchmarks for strategies 1 to 4 is approximately 40 seconds, whereas strategy 5 has an average time of 16568 seconds.

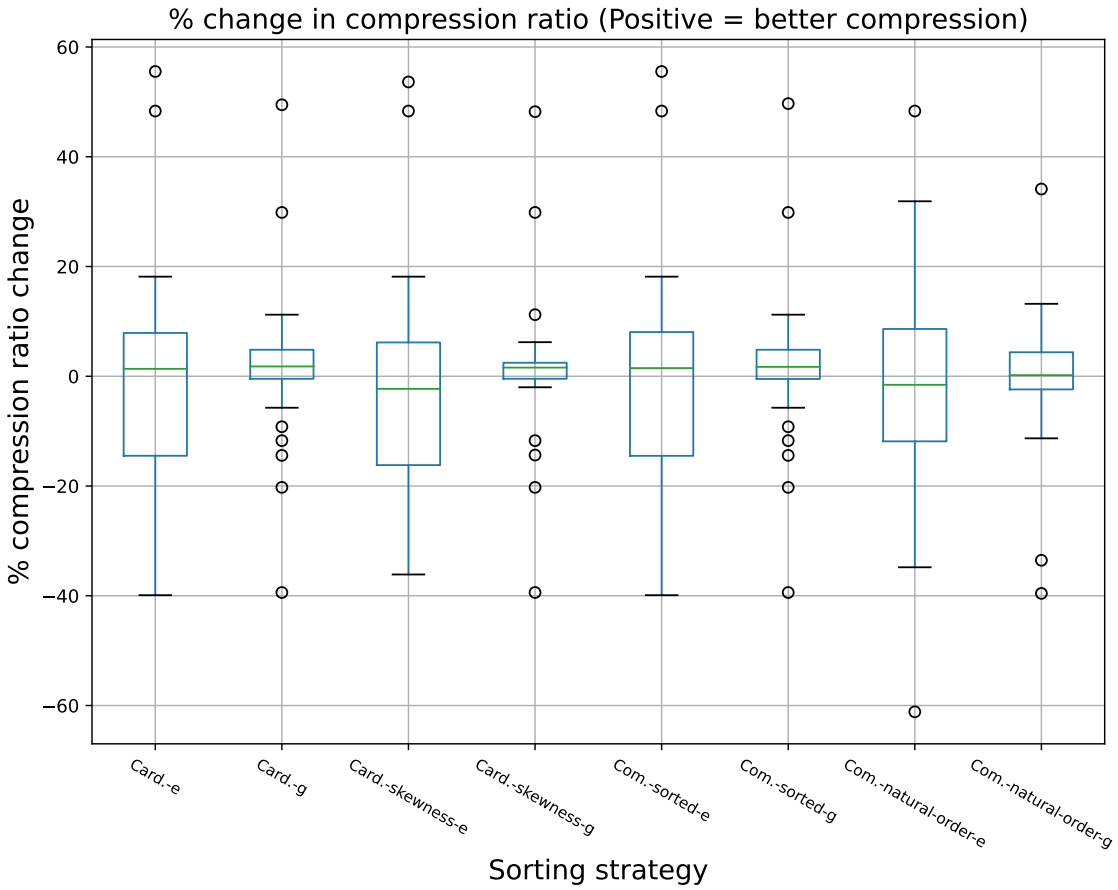

Figure 4.4: Public BI compression ratios. All reordering strategies, sorting the entire table or per row group (granular) on 45 datasets. "E" for "entire table" and "g" for granular sorting

Since Vortex ran only on a subset of the data, we also compare strategies 1 and 4 on 45 out of 46 datasets. The results can be seen in Figure 4.4. The difference between reordering the entire table and granularly remain the same as with 11 datasets. However, we can see that the median is further above 0 for all strategies except the combined natural order. For granular sorting with the cardinality, approximately 75% of the results are positive, meaning it increased the compression ratio (i.e. reduced the file size).

Based on this analysis, we can answer the questions posed at the start of this chapter.

Question 1

Which strategies improve the RLE compression ratio compared to the natural order, and how do their processing speeds compare?

Applying reordering, on average, improves the compression ratio when combined with RLE compression. Using the combined cardinalities, as seen in strategies 3 and 4, does not result in a significant difference in running time or compression ratio in our experiments. Strategy 5, Vortex, has a worse compression ratio than strategies 1 to 4 while also being slower in our experiments. While this worse time performance is due to the overhead our Python implementation introduces, it was also found by Lemire et al. (14). Strategies 2 to 4, while attempting to use fewer columns, are not significantly faster than strategy 1 but more complex in their implementation. For now, strategy 1 is preferred as it has a good trade-off between compression ratio improvement, time and implementation complexity.

Question 2

How does the compression ratio change when strategies are applied granularly (i.e. to a group of rows)?

The experiments were performed on a group of 100.000 rows and the entire table. Applying the strategies to 100.000 rows at a time results in a tighter boxplot, meaning the results are clustered closer together and more consistent. We do not observe that one strategy has significantly better compression over the other. It does impact the time, but this could be due to the implementation in Pandas.

Question 3

Are there metrics to predict the performance of a strategy?
We want to predict the impact of reordering on the compression ratio before performing the reorder. This prediction could especially save unnecessary work on larger datasets. Since we will use granular reordering in DuckDB, we analyse the worst and best-performing datasets of strategy 1, cardinality reordering, at a granular level.

First, we examined the data types of the three worst and top-performing datasets from our experiments, as seen in Figure 4.5. The three datasets from the left of the figure

Table name	Average cardinality
Taxpayer_1	809941.68
Provider_1	809941.68
TrainsUK2_1	272190.27
TableroSistemaPenal_1	213.96
Generico_1	168653.65
Redfin3_1	536048.57

Table 4.1: Average cardinalities of tables

increased in file size, while the three on the right had a decrease in file size. Studying the figure, we do not notice a pattern in the data types. The "bad" and "good" datasets both contain a mix of DOUBLE, INTEGER and VARCHAR types. While the best performing dataset, Redfin3, is 61% DOUBLE's, Generico only contains 5%.

Next, we examined the average cardinality of the same datasets. These are shown in Table 4.1. We notice that the top-performing datasets have a lower average column cardinality. It would have to be investigated whether this is generalisable enough to make it a consistent metric.

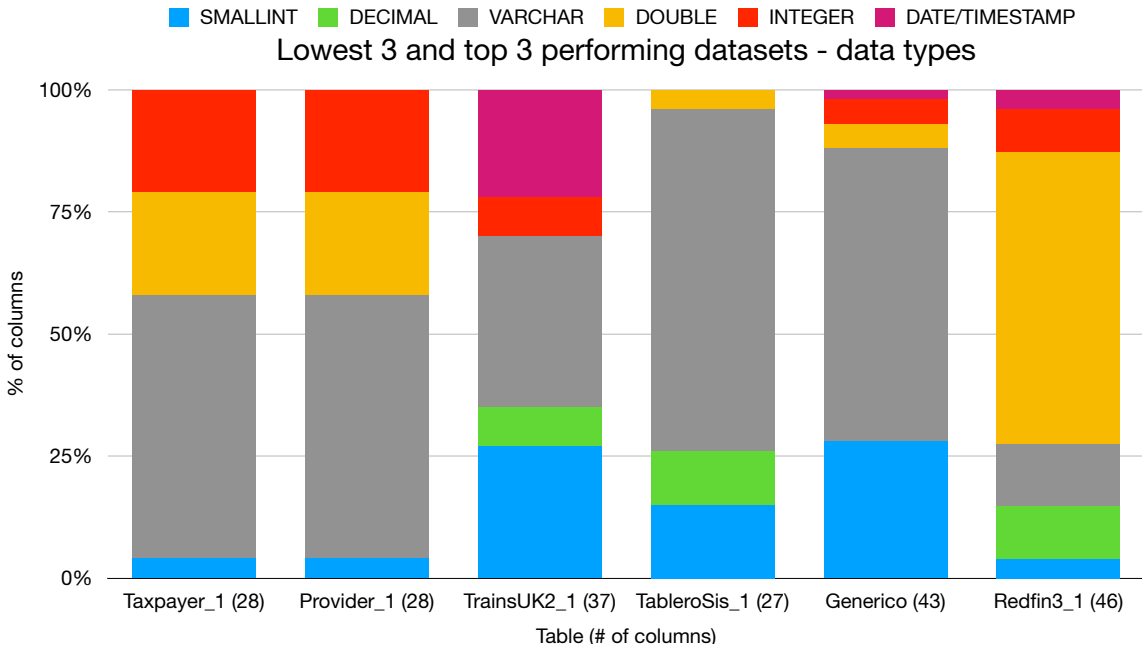

Figure 4.5: Public BI data types

$$P0 = \sum_{c}^{i=1} f(v_i)/nc \tag{4.1}$$

$$\omega = \frac{\sum_{c}^{i=1} n_{1,i}}{n + c - -1} \tag{4.2}$$

Two other possible metrics are equation 4.1, $P0$, and equation 4.2, ω, as described by Lemire et al. (14). $P0$ calculates the statistical dispersion of a table, where $f(v_i)$ is the number of times v_i occurs in a column, and v_i is the most frequent value of the column. ω is a metric for determining when a lexicographical sort is optimal. c is the number of columns, n is the number of distinct rows and $n_{1,j}$ is the number of columns when considering only the first j columns. Lemire et al. recommend applying the Vortex algorithm when ω is higher than 3 and $P0$ is higher than 0.3.

We found that calculating these metrics is expensive, especially on larger datasets. All datasets, except one, had an ω value larger than 3. Only two datasets had a $P0$ larger than 0.3. The P0 and ω values do not seem to correlate with their results, as they had a worse compression ratio after running Vortex. Furthermore, it is unlikely that we will use Vortex due to its high computational cost, as previously mentioned.

Conclusion: Related work has not shown us any reliable metrics in predicting the RLE performance after reordering. However, our exploration has shown a difference in average cardinality on the worst and top-performing tables in our experiments. It remains to be seen whether this is significant enough to be a reliable metric.

4.4 Summary

From the exploratory results, we can observe that reordering rows lexicographically by increasing column cardinality can result in better RLE compression than the data's natural order. Furthermore, we can conclude that we do not require a list of exact cardinalities for all columns in order for the strategies to be successful. The Vortex algorithm, on average, did not improve the Public BI dataset's compression ratio and is very slow.

We also attempted to find a metric by which we could judge the compression improvement before reordering. There is a possibility of using the average cardinality to determine whether reordering a table is beneficial. It should be further investigated whether this is a reliable metric.

The next phase of this thesis will focus on implementing strategy 1, sorting by increasing cardinality, in DuckDB. This method has a balance between compression ratio, running

time and engineering complexity. Strategies 3 and 4, which used the combined cardinalities, were also considered. However, their results were not significantly different from strategy 1 and are more complex to implement due to calculating combined cardinalities. Strategy 4, combined cardinalities in the natural order, had a decrease in compression ratio on average. Vortex has a too high computational cost for our use case and will not be used.

5

Design & implementation in DuckDB

This chapter will discuss the design of self-optimising storage. We then discuss implementation challenges and complexities associated with each module and where we had to deviate from the original design.

5.1 Design

We aimed to create a self-optimising storage that optimises data for compression automatically each time data, in the form of row groups, is written to disk. The reordering must be done quickly and with a low impact on DuckDB's performance. Our storage-optimiser consists of the sort order module, sorting module and compression module. We discuss each module separately and then detail how these modules were brought together to create the storage-optimiser.

Sort order module: The sort order module determines the cardinality of the required columns. It is also responsible for determining which columns should have their cardinality estimated, how to calculate this cardinality and then returning the columns which should be used for sorting. The design is generic to allow alternative ways of generating sort orders in the future.

Sorting module: The sorting module runs after determining the order of columns based on the "key" columns selected by the sort order module and creates a new row group from the sorted data.

Compression module: The compression module will be responsible for compressing the sorted data and determining which compression method to use.

Figure 5.1 shows the architecture of the storage-optimiser as a whole. The starting point is query processing because the user must have executed queries to create any checkpoints.

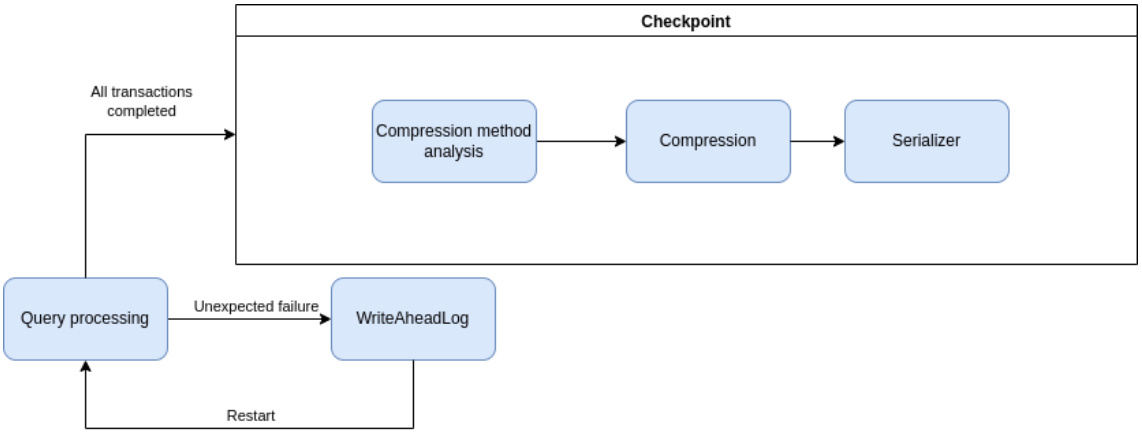

Figure 5.1: Architecture of the checkpoint process in DuckDB

Once all transactions are completed and the user gracefully shuts down the database, the checkpoint module will be called. A checkpoint is performed per row group, which means the checkpoint module will also be called for each row group. First, an analysis is performed on each column to determine what compression method to use. A compression method is chosen based on a score, which is the approximate amount of bytes of the column on disk. Second, each column is compressed with its respective method. Finally, the data is written to disk by the serialiser.

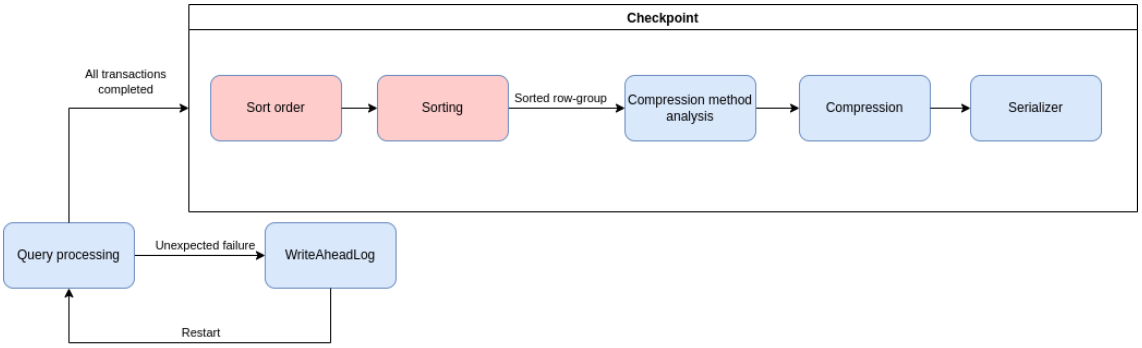

Figure 5.2: Architecture of the checkpoint process in DuckDB after implementing the self-optimising storage

Figure 5.2 shows the DuckDB checkpoint architecture after implementing our self-optimiser. The two modules have been added at the start of the checkpoint process before the compression method analysis is performed. The advantage of this design is that we can turn the self-optimiser on and off without impacting other parts of the checkpoint process.

5.2 Implementation

5.2.1 Enabling and testing

In DuckDB, features can be enabled or disabled through PRAGMAs. Some are used for debugging, like the PRAGMA *enable_profiling*, which helps with analysing query performance or PRAGMA *database_size*, which shows information about the block count and size. They can also turn features on and off, such as PRAGMA $force\_compression = false$, which disables compression when writing data to disk. Our first step was creating the $force\_compression\_sorting$ PRAGMA to enable/disable the storage-optimiser. It is implemented in the database configuration settings and enabled by default.

We have chosen to always attempt sorting, as we expect the average compression ratio to improve. Furthermore, the order in which the data is stored on disk does not impact the end-user as there are no guarantees about the data order without an ORDER BY clause (41).

The PRAGMA was useful when testing. Upon every commit, DuckDB runs a set of approximately 1900 tests. These tests are made up of SQL queries and verify if the results are correct (42). Almost all tests are written using Sqllogictest (43), a framework to test SQL databases. We used these to verify the correctness of our implementation.

Since our sorting is enabled by default, it also meant changing existing unit tests to fit the new default behaviour. For example, many query results were assumed to be returned in insertion order which was no longer the case. These issues were resolved by adding an ORDER BY clause or manually changing the result to the new order after sorting.

5.2.2 Checkpointer

The storage-optimiser runs before the data is written to disk during the checkpoint process. We have chosen this moment because all transactions are finished at this point, and there will be no further changes to the data.

Before calling the sorting method, we first determine which key columns to use. Thus, we analyse all the columns and determine their compression method. Then, an *RLESort* object is created, which holds a vector of the candidate key columns. Either we only add columns that will be compressed with RLE to this vector, or we start with all columns. During our experiments, we investigate the differences between these approaches. DuckDB supports booleans, integers, unsigned integers, floats and doubles for RLE compression. Our implementation also supports these column types.

5.2.3 RowGroup Scanning

In both the sort order module and the sorting module, we sequentially scan the RowGroup to convert the data from a RowGroup object to a DataChunk object. The DataChunk has been initialised with the same types as the RowGroup.

To facilitate rollbacks after a transaction has been committed, the RowGroup uses a VersionInfo array to keep track of deleted tuples. For example, if tuple 2 was deleted in transaction 3, it is added to the VersionInfo. Therefore we cannot scan all the tuples naively, as this will include tuples marked as deleted. When scanning, we first check the VersionInfo and compare the transaction ID of the deletion against the current transaction ID. If we are in a higher transaction, we add the tuple to a SelectionVector. This Selectionvector is used after scanning to select non-deleted tuples and add them to the DataChunk. The scanning process is performed in steps of size $STANDARD\_VECTOR\_SIZE$, which is 1024 tuples.

At first, we had issues scanning RowGroups which had any deleted entries. After a thorough investigation, we discovered the transaction ID was already incremented before the checkpoint. The transaction ID is used during the scan to ascertain whether an entry is still valid or if it has been deleted. We must therefore scan the $currenttransactionID - 1$ to exclude deleted tuples.

5.2.4 Modules

Sort order module: The sort order module determines which candidate key columns to use. The implementation requires calculating the cardinalities of the column. Therefore, we create a vector of HyperLogLog (HLL) counters containing one counter per column. The scanned DataChunks are added to the counter incrementally until the entire RowGroup has been scanned.

These counters are passed to a function which contains all possible sorting options, meaning it is possible to add more options in the future using the same infrastructure. We call *Count()* on each HLL and add it as a key column if it meets the requirements. The final key columns are encoded as $Tuple(cardinality, column\_id)$. The list is sorted from lowest to highest cardinality, determining the final sorting order. A vector with the sorted column ids is passed to the sorting module.

Sorting module: Before adding tuples to the sorting function, we initialise two objects. The GlobalSortState and the LocalSortState. The LocalSortState acts as a sink for the data and takes the *keys* and *payload* as an argument. The keys are the columns to sort

with, and the payload contains all the data of the RowGroup. Once the entire RowGroup has been "sunk", we add the LocalSortState to the GlobalSortState, which triggers the sort. The sorting algorithm is DuckDB's implementation of a parallel merge sort (44).

We then create a new RowGroup object which contains a starting point and a tuple count. The starting point is where the previous RowGroup ended, and the count is all non-deleted tuples. We then scan the GlobalSortState and incrementally add the DataChunks as columns to the newly created RowGroup. Finally, the old RowGroup is replaced with the new one.

DuckDB uses two types of blocks on disk, persistent and transient. If there are no changes in a block (i.e. an UPDATE or DELETE) since the last time the data was written to disk, the block is persistent; otherwise, it is transient. Since reordering changes the order of the data, we must remove the persistent blocks from the disk and replace them with transient blocks. This removal is done before starting the compression.

Compression module We did not make any changes to the compression module of DuckDB, as this runs separately after sorting the data. Therefore, we will only explain how the current implementation works.

Compression is done per column. First, the compression method is determined by checking whether a flag has been set to force a compression method (i.e. force RLE compression for all columns). If this is not the case, an analyser is run for all available compression methods. A compression method is skipped the data types are not supported. The analyser returns a "score", which estimates the number of bytes required to store the data. The score is calculated by executing the compression steps without writing the data to disk. For example, RLE's score is estimated by calculating the number and length of runs. The compression method with the lowest score is used. Finally, the actual compression is performed, and the data is written to disk.

5.2.5 Rewriting persistent blocks

Our implementation went through several iterations as the thesis progressed. We started with the naive version to establish baseline performance and continued from here. During our experiments, we noticed a significant increase in checkpoint time for datasets with more than ten tables (See Appendix B for a detailed explanation of the experiments). An investigation showed that data already on disk was being rewritten during every checkpoint.

```
1 CREATE TABLE table1 (column1 INTEGER);
2 --CHECKPOINT (table1)
3 CREATE TABLE table2 (column1 INTEGER);
```

```
--CHECKPOINT (table1, table2)
INSERT INTO table1 VALUES (1);
--CHECKPOINT (table1, table2)
INSERT INTO table2 VALUES (1);
--CHECKPOINT (table1, table2)
UPDATE table1 SET column1 = 2 WHERE column1 = 1;
--CHECKPOINT (table1, table2)
```

Listing 5.1: Naive version - checkpoint behaviour

Listing 5.1 shows code to illustrate the checkpoint behaviour of the naive version. The $CHECKPOINT(table\_x, table\_y)$ comments show what happens when DuckDB automatically checkpoints. We can see that both tables are checkpointed after each table creation or insertion. However, no distinction is made between tables already written. Because there is no distinction, $table1$ will be continuously rewritten as more tables are added, even if there are no changes.

To remedy this, we perform a check before sorting. A RowGroup consists of transient and persistent segments. We check all segments in the RowGroup, and if any are transient, we sort the entire RowGroup again. Any persistent segments are marked as modified when we sort, meaning their disk space is freed to be rewritten. If there are only persistent segments, we check whether any of these segments contain updates. For example, a tuple has been modified with a new value. If this is the case, we also sort and rewrite the RowGroup.

```
CREATE TABLE table1 (column1 INTEGER);
--CHECKPOINT (table1)
CREATE TABLE table2 (column1 INTEGER);
--CHECKPOINT (table2)
INSERT INTO table1 VALUES (1);
--CHECKPOINT (table1)
INSERT INTO table2 VALUES (1);
--CHECKPOINT (table2)
UPDATE table1 SET column1 = 2 WHERE column1 = 1;
--CHECKPOINT (table1)
```

Listing 5.2: Optimised version - checkpoint behaviour

Listing 5.2 shows the checkpoint behaviour after applying the optimisations. Only the updated RowGroup of the table is rewritten.

6

Evaluation in DuckDB

We will now evaluate our self-optimising storage integrated into DuckDB. The evaluation of our experiments will focus on three aspects:

1. Impact on checkpoint speed
2. Impact on compression ratio
3. Impact of reordering individual row groups versus the entire table

6.1 Experimental setup

The experimental setup was the same as described in section 4.1. The Public BI benchmark was used as an example of a realistic dataset, and the experiments were performed on the same cluster machine.

The experiments were run through a Python script. Building DuckDB with the following command: "BUILD_PYTHON=1 BUILD_BENCHMARK=1 BUILD_HTTPFS=1 make GEN=ninja", we install a development version of the Python API. We then loop through all datasets in the Public BI benchmark and load them into the database. We first create all table(s) and then use $COPY$ to insert the data from the *.csv.gz* file. After the data has been loaded, we issue the $CHECKPOINT$; statement, which writes the data to the disk. Finally, the results are collected and written to a CSV file for later analysis. We repeated the process with and without reordering.

As we aim to simulate a realistic scenario, we leave it up to DuckDB to decide which compression method will be used on which column. If no columns are compressible by RLE, we will not reorder. In our results, this will show a 0% change in compression ratio.

6.1.1 Automatic checkpoints

When connecting to a DuckDB database, one can choose to connect to an in-memory database or one (already) on disk. DuckDB does not rely on the user to issue the CHECKPOINT command to start writing to disk. The value of the PRAGMA *wal_autocheckpoint* determines when data is checkpointed. The standard value is 10MB, which means a checkpoint is done when 10MB is contained within the WAL. The WAL, or the write-ahead log, is continuously updated whenever a user changes the database. It is also a safeguard should the database exit unexpectedly.

To have complete control over when checkpoints occur, we can change the value of *wal_autocheckpoint* to a large value. This large value means DuckDB will only checkpoint once the CHECKPOINT command is issued.

6.1.2 Reordering strategies

Stemming from the results detailed in section 4, we determined that reordering the columns from low to high cardinality achieves a balance between compression ratio and time performance. We used this strategy for our implementation, using only RLE compressed columns with a cardinality below 10.000. This cardinality is approximately 10% of a row group.

6.2 Evaluation

In this section, we will evaluate our DuckDB implementation. The first experiment will establish a baseline performance for compression and time by running the benchmark with *wal_autocheckpoint* set to 1TB. The second experiment will evaluate the performance with *wal_autocheckpoint* set to its standard value, which most users will use. The third experiment evaluates time performance as the number of tables increases and compares it to not reordering. Finally, we perform profiling to discover opportunities for optimisation.

6.2.1 Baseline performance

In our first experiment, we are focused on determining whether the reordering strategy improves the compression ratio. The time it takes to run the benchmark is also measured. We realise that performing the reordering will take extra time, but it should keep the impact on the user to a minimum. Our first goal is not to exceed double the checkpoint time. We believe this to be a reasonable goal for our prototype if the reordering improves the compression ratio.

The *wal_autocheckpoint* is set to 1TB, and the *CHECKPOINT* command is issued. Since none of the datasets is larger than 1TB, DuckDB will only checkpoint once. After the checkpoints, we measured the size of the database file on disk in bytes and compared it to not reordering.

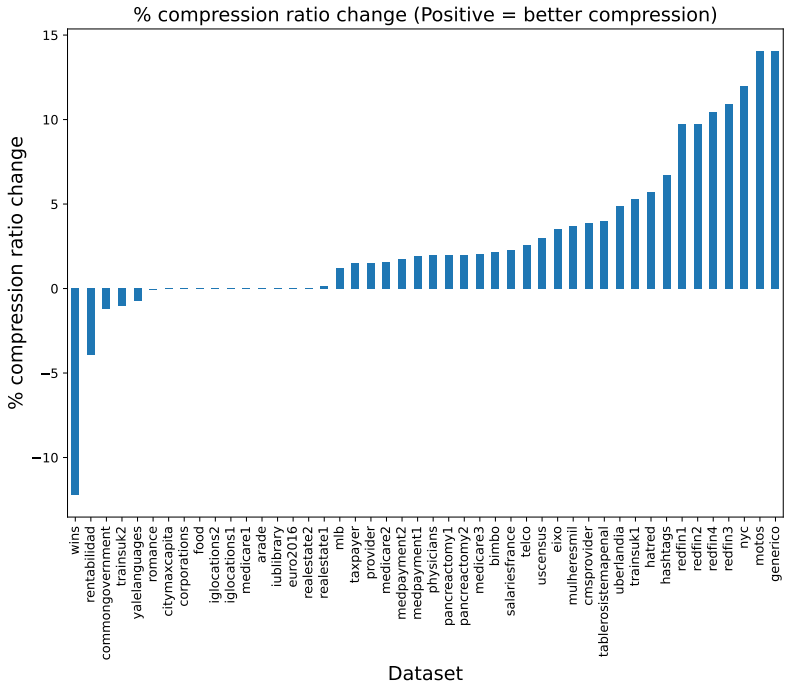

Figure 6.1: Baseline - compression ratio change

Figure 6.1 shows the compression ratio change after reordering compared to not reordering. The x-axis shows the percentage change in compression ratio, where a positive number means better compression (i.e. the file size got smaller). The y-axis shows the datasets from the Public BI benchmark. Our reordering algorithm will be a success if most results are higher than zero. We can see that on 31 datasets, there is an improvement in compression ratio. Nine datasets remain the same, and six see a decrease in compression ratio. On average, the compression ratio improves by 2.76% with peaks of 14.05%. The Wins dataset is notable, as it has a 12.21% decrease in compression ratio.

In section 4.3 the possibility of using the average cardinality to predict compression improvement was discussed. To further evaluate this, we calculate the average cardinality of Wins and Rentabilidad, the two worst-performing datasets. They are compared to Generico and Motos, the two best-performing datasets. The cardinality is calculated on the RLE compressed columns. We scale the results by dividing the cardinality of these

Table name	Row Count / Cardinality
Wins_1	635.96
Rentabilidad_1	6.23
Motos_1	132.09
Generico_1	140.14

Table 6.1: Average cardinalities of tables

columns by the number of rows.

In table 6.1 we can see the results. A higher number means few unique values, while a low number means many. The intuition is that if there are few unique values, RLE can compress the table better. However, the Wins dataset, the worst performing, also has the highest number in the table. These experiments do not show a correlation between this metric and the compression ratio.

We investigated the Wins dataset further. The dataset consists of four tables, three of them with similar structures in terms of columns. These three had a worse compression ratio. We pick one of these tables for further investigation, Wins_2. The table has 647 columns and 519794 rows (i.e. 5 row groups). Since each column has its compression method assigned per row group, we can have a total of $5 \times 647 = 3622$ compressed sections of columns, given that all row groups are full. Before sorting, 1840 column sections were compressed with RLE while 1768 were not. After sorting, 1191 column sections were compressed with RLE and 2431 were not. If we focus on the number of columns used in the sort order, we notice that 100 columns are used to sort on in one row group, while up to 550 columns are used in another. We then determined which columns had the largest increase in size. One column, "nSUPERF", went from occupying 5 blocks to 11 blocks on disk. If we look sort order of the columns, it was approximately in the middle of this order. The large number of columns may make it more challenging to find a sort order which increases the compression ratio since the number of possible orders increases. Another outlier, USCensus with 1562, had an increase in compression ratio. However, we can notice that as the number of columns increases, the consistency of using the cardinality to find the sort order decreases.

Figure 6.2 shows the percentage change in time for the entire checkpoint process to complete when reordering compared to no reordering. The x-axis shows the percentage change in time, where a positive number means the process took longer. The y-axis shows the datasets from the Public BI benchmark. We can see that apart from the two datasets,

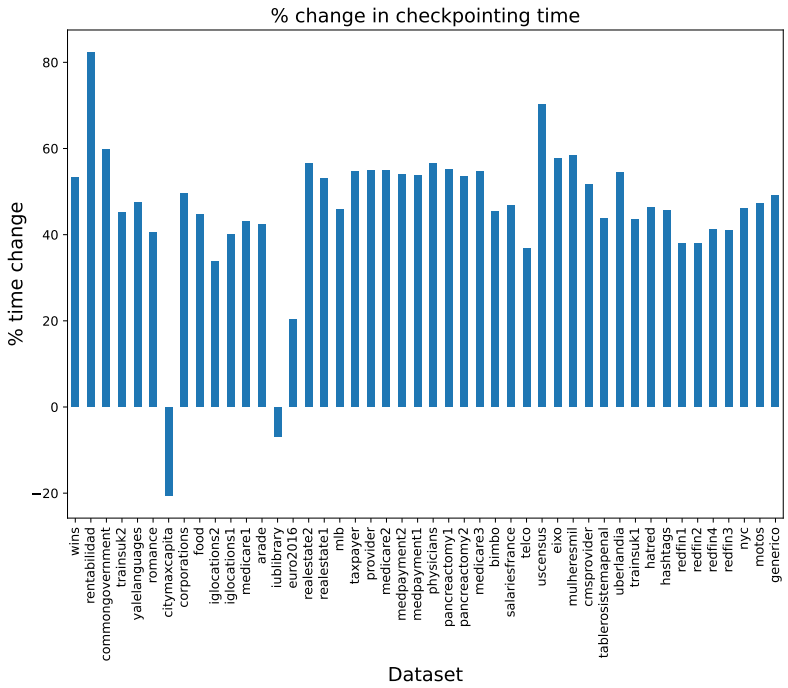

Figure 6.2: Baseline - checkpoint time change

all had an increase in time. The average increase in checkpoint time is 50.74%, with a peak of 82.36%. The average increase is below our goal of 100%.

6.2.2 Real-life performance

We repeat the previous experiment after the changes made to the behaviour of performing a checkpoint multiple times while keeping the *wal_autocheckpoint* value as standard.

Figure 6.3 shows the change in compression ratio. We can see that it is the same as Figure 6.1 as we expected.

Figure 6.4 shows the change in checkpoint time. The checkpoints took longer than in experiment 1. This increase was expected, as setting the *wal_autocheckpoint* to a lower value means the checkpoints will be performed more frequently. The average increase is 74.98%, with a peak at 459.45% for the MLB dataset. The datasets with the most significant increase are datasets with multiple tables. Rentabilidad, SalariesFrance and CommonGovernment all contain approximately ten tables. MLB is an outlier, as it contains 68 tables. We also observe two datasets which are faster than without reordering. However, CityMaxCapita and IUBlibrary are two small datasets and take less than 1 second to checkpoint. While there is a percentage difference, in reality, it is negligible.

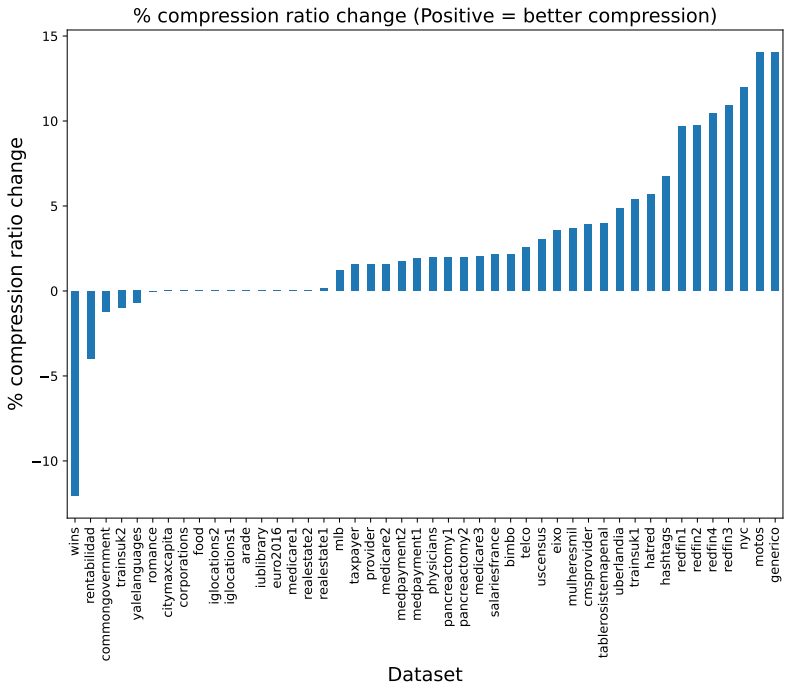

Figure 6.3: Real-life optimised - compression ratio change

6.2.3 Time increase per table

In experiment 2, we noticed a sharp increase in checkpoint time for many tables. We investigate if there is a relation between checkpoint time and the number of tables. We use a unit test that creates several tables with 10 million integers each, alternating between the values 0 and 1. Having alternating values means they must be sorted first for optimal RLE compression. All tables are then checkpointed. For each set of tables, we repeat the experiment 5 times and record the average time in seconds. We use DuckDB's *benchmark_runner*[1].

Figure 6.5 shows the average checkpoint time as the number of tables increases, both when we reorder and when we do not reorder the table. The x-axis shows the number of tables, and the y-axis shows the average time of 5 runs in seconds. We can observe that both reordering and not reordering have a linear relationship with the number of tables. However, we see that the two lines diverge as the number of tables increases. We expect the lines to diverge, as reordering takes a longer time per table than not reordering. These results explain why the MLB dataset, with 68 tables, has an increase of 459.45%.

[1]https://github.com/duckdb/duckdb/tree/master/benchmark

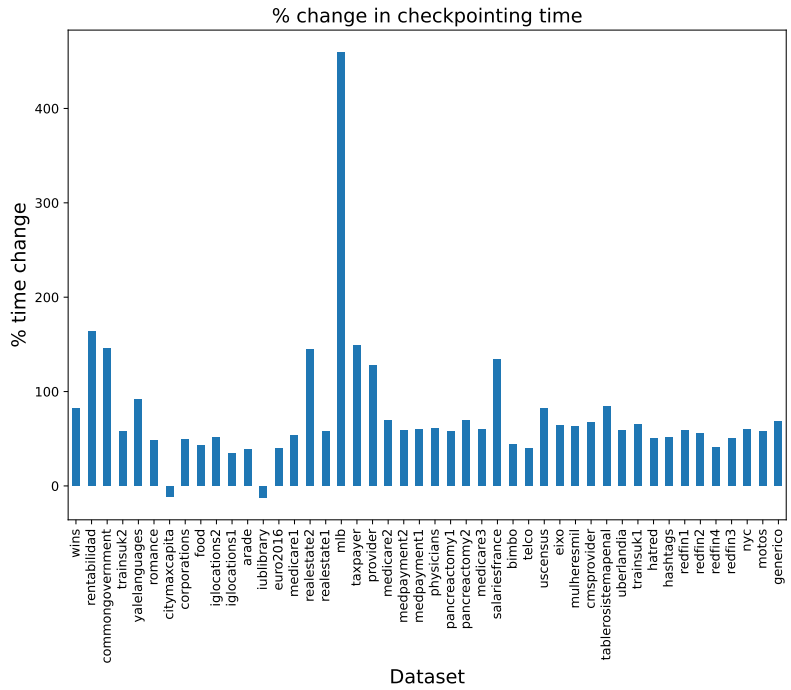

Figure 6.4: Real-life optimised - checkpoint time change

6.2.4 Profiling

In this experiment, we profile the checkpoint process to discover sections which could be optimised to make the implementation more efficient. Optimisations which are quick to implement but could provide a significant performance benefit are preferred.

We start by writing a short unit test that creates a table, loads in data and checkpoints. We then analyse the time each module takes to find bottlenecks using the profiling tool Perf, collecting 15952 performance samples per second. Our profiling is limited to the sort order and sorting modules since they have been newly added.

Table 6.2 contains the information returned by the profiler, broken down by function call. We have not included the underlying calls in the table but will discuss those briefly now. The most considerable time is spent in the *SinkKeysPayloadSort* function, which scans the RowGroup and creates a DataChunk with all tuples. This DataChunk is added to a sink, which will be used to sort the data later. These three operations make up most of the time spent in the function and have few opportunities for easy optimisations. Next is the *CreateSortedRowGroup* function, which scans the sorted data to a DataChunk and appends it to the new RowGroup. This situation, again, leaves little room for quick optimisations. After this, we have the *GlobalSortState* :: *AddLocalState* that performs

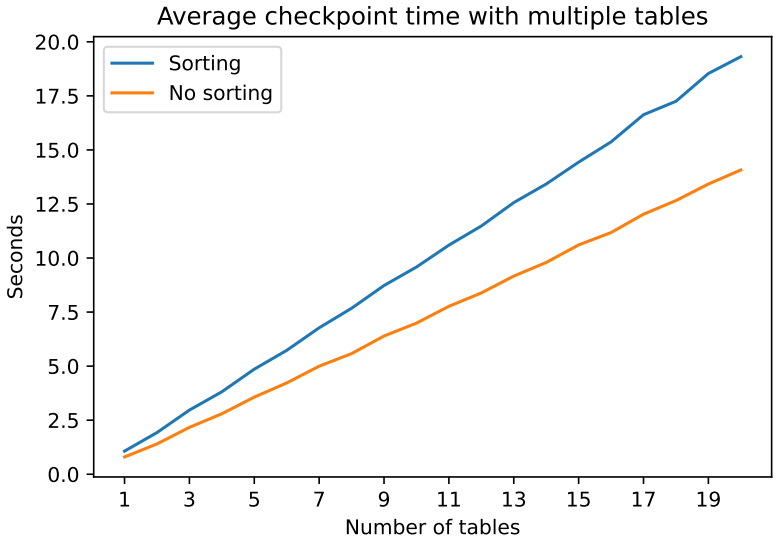

Figure 6.5: Checkpointing multiple tables

% time spent	Function name
33.9	SinkKeysPayloadSort()
31.1	CreateSortedRowGroup
22.8	GlobalSortState::AddLocalState
11.6	FilterKeyColumns
0.6	Miscellaneous calls

Table 6.2: Profiling result of the storage-optimiser

the actual sorting. This functionality was already implemented in DuckDB as explained in section 5.2.5. We do not see a possibility for easy optimisation. The last function is *FilterKeyColumns*, which uses HyperLogLog to determine the counts of the columns and filters key columns out according to some metric. Again, the largest time is spent scanning the data (76.3%) to a DataChunk, while the cardinality estimation only takes up a fraction of the time (1.3%).

This experiment concludes that most of the time is spent scanning the data to DataChunks. The best optimisation opportunities are in reducing the frequency of scans or increasing their performance.

6.3 Benefit analyser

Throughout the experiments, we observed that some tables have a worse compression ratio after reordering. Thus, we implement a benefit analysis module to ensure the compression ratio is never worse after reordering than before.

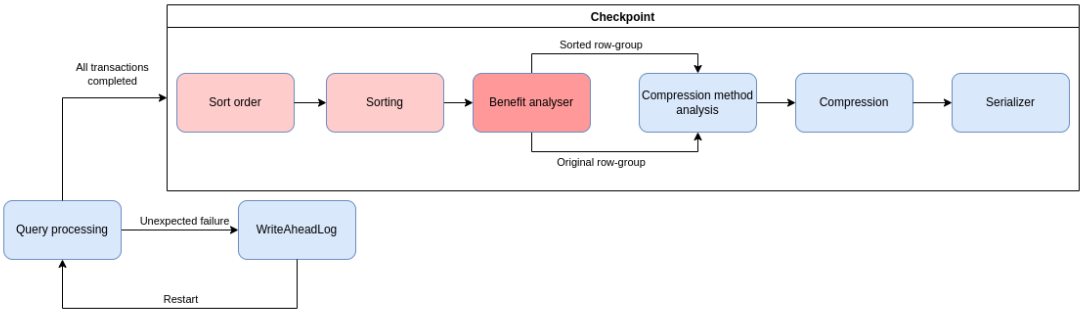

Figure 6.6: Architecture of the checkpoint process with benefit analysis module

In Figure 6.6 we can see where it fits into the overall design. Since cardinality estimation and sorting are expensive operations, the storage-optimiser should ideally determine whether a reorder is necessary. However, as seen in chapter 4 we have not found a definitive metric which can assist us in making this decision beforehand. The benefit analyser will therefore calculate the potential benefit of compression before reordering and after. It will then compare values and decide whether the reordering should happen.

We leverage the "score" returned by the various compression methods to analyse the benefit of reordering. After reordering, the compression score of the sorted row group is calculated. We compare this against the score of the unsorted row group. If the sorted row group has a lower score (i.e. file size is smaller), then we replace the unsorted row group with the new one.

6.3.1 Evaluation of benefit analyser

The performance of the benefit analysis module is measured for both compression ratio and total time.

In Figure 6.7 we can observe the change in compression ratio after the benefit analyser has been applied. None of the datasets has a decrease in compression ratio, and the lowest point is now 0%. The Wins dataset, which consists of 68 tables, even has an increase in compression ratio because some tables have been sorted and others have not. The

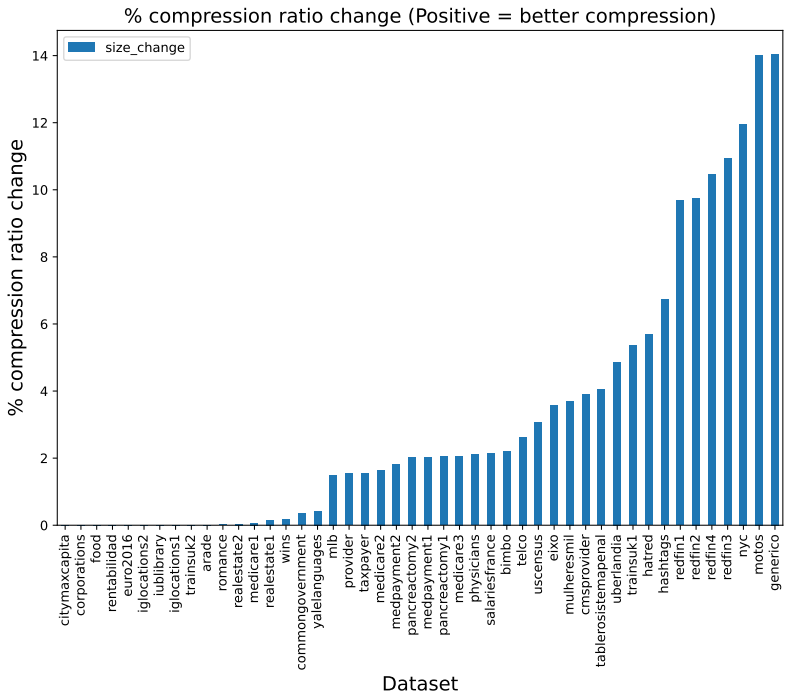

Figure 6.7: Benefit analyser - compression ratio change

average compression ratio is 3.23%, compared to 2.76% before adding the benefit analyser. However, doing an extra analysis does have a time cost.

Figure 6.8 shows the change in checkpoint time. If we compare it to Figure 6.4 we notice an increase for most tables. This increase is expected as we perform an extra step during the reordering. The average checkpoint time is 98.29% higher than not reordering, compared to 74.98% in the real-life performance experiment. Again, CityMaxCapita is faster because it takes less than 1 second to checkpoint this small dataset.

6.3.2 Using all columns

The previous experiments were performed by only selecting columns that would have been compressed with RLE as "key" columns for sorting. In this experiment, we compare this method against using all columns and only filtering them based on their cardinality ($<$ 10% of the row-group size).

Figure 6.9 shows the difference in compression ratio between using only columns which would have been compressed as RLE columns for sorting, versus using all columns. We can observe that some datasets benefit hugely from using all columns to sort on. Especially Bimbo, which achieves a 49.07% better compression rate, as opposed to 2.21%. It is likely

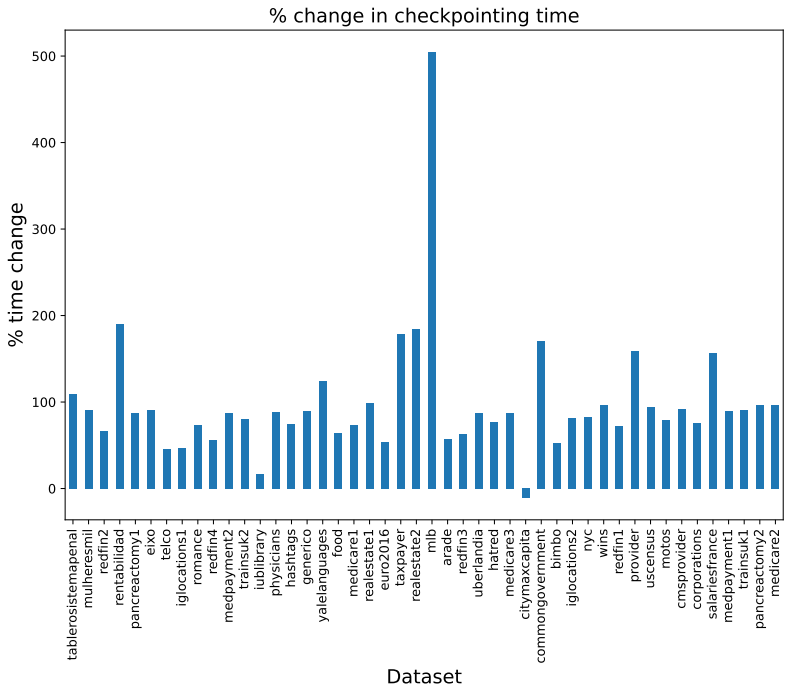

Figure 6.8: Benefit analyser - checkpoint time change - average of 5 runs

that after sorting, more columns are compressable by RLE. The average compression ratio is increased to 4.37% from 3.23%.

However, there is a caveat. Using more columns to sort with has a time cost. A comparison can be seen in Figure 6.10. The time change compared to not sorting increased for almost every dataset. The average increase in time increased from 98.29% to 139.21%, going above our goal of keeping the average checkpoint time below double.

This experiment shows that sorting by columns results in better compression ratios, especially if sorting causes more columns to be compressable with RLE. However, sorting by more columns requires more time. To counter this, one could make the cardinality selection criteria stricter, causing fewer columns to be used in the sort order. This selection criterion becomes a parameter to be tuned to find the optimal balance between compression ratio and time. For now, we use our original implementation as this remains below our goal of doubling the checkpoint time.

6.4 Summary

The experiments show that on the Public BI dataset, the average compression ratio of RLE improves by reordering row groups from low to high cardinality. The average improvement

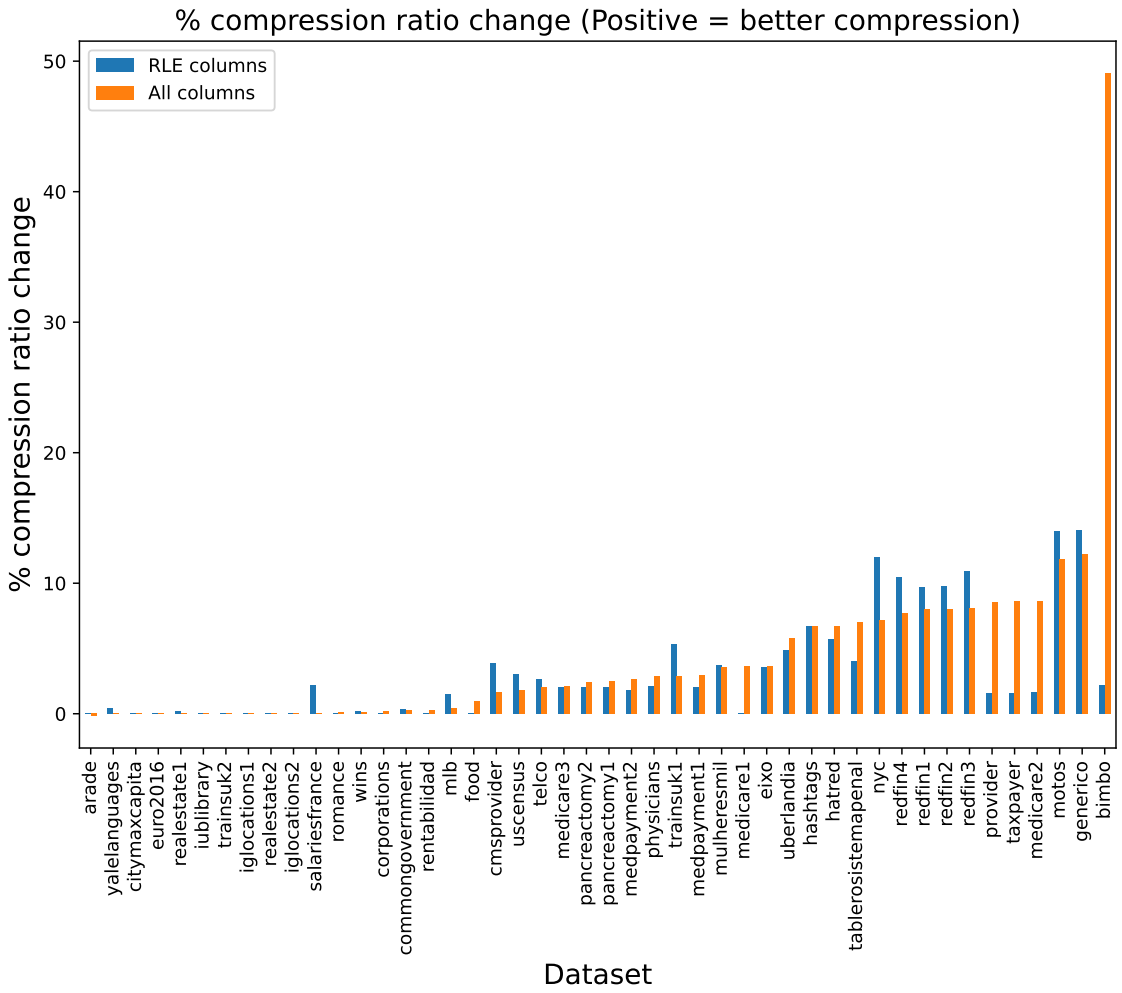

Figure 6.9: Reordering only with RLE compressable columns as "key" columns or with all columns - compression ratio change

is 2.76%, with a maximum of 14.05% before adding the benefit analyser. Predicting the compression ratio change after reordering using the average cardinality seems unfeasible. From our experiments, we do not see a correlation between the two. However, it is more difficult to find a sort order for tables with a large number of columns.

The reordering on our experimental machine takes, on average, 84.62% longer than not reordering. However, datasets with many tables suffer more and checkpoint time can increase to 459.45%. This increase is because the checkpoint time increase per table is linear but grows faster than if we do not reorder.

Enabling the benefit analyser increases the average checkpoint time increase to 98.29%. However, the average compression ratio is also increased to 3.23%. This time increase

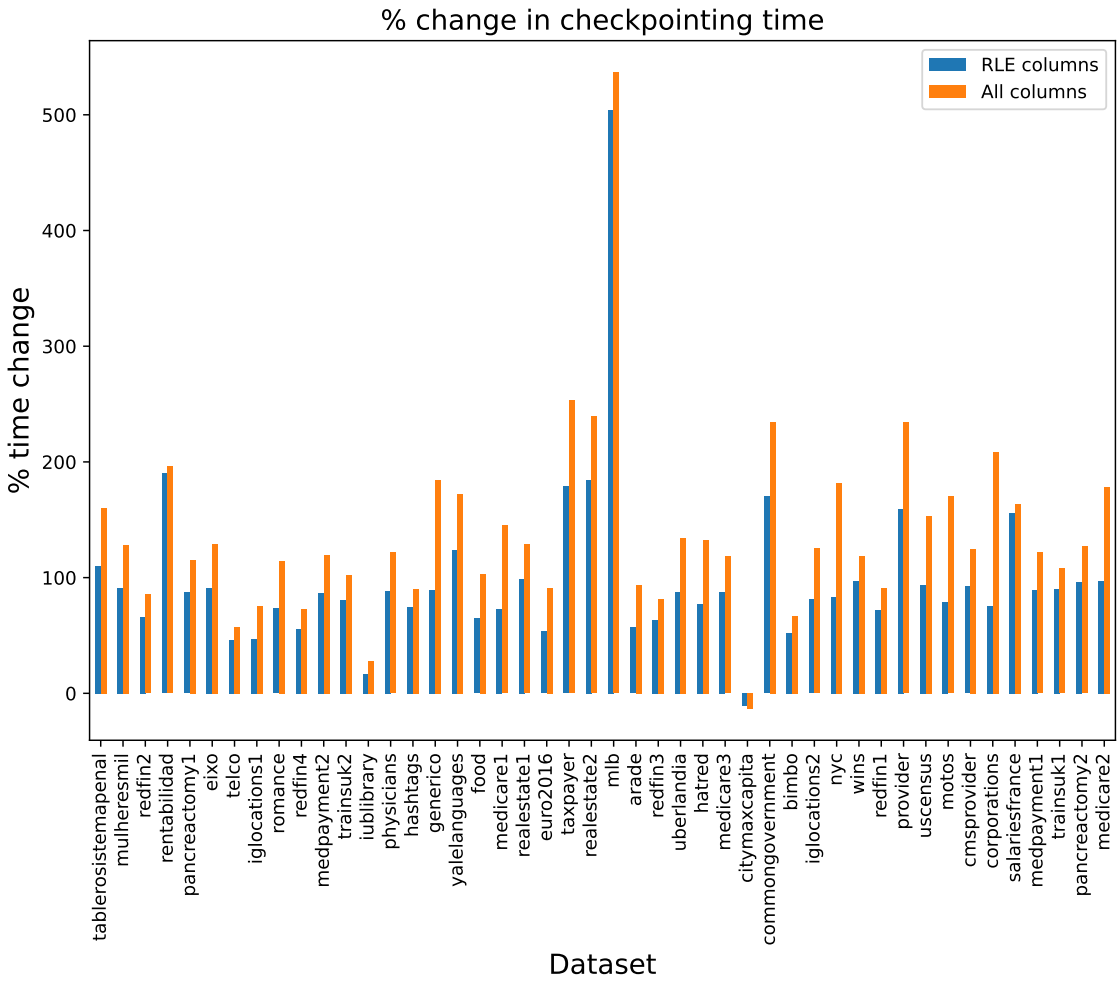

Figure 6.10: Reordering only with RLE compressable columns as "key" columns or with all columns - checkpoint time change - averaged over 5 runs

is because an additional step is required to evaluate the effectiveness of the reordering. The advantage of this method is that the datasets can never regress (i.e. the file after reordering is never larger than before). If we sort by all columns instead of only columns which would have been compressed with RLE, we further increase the average compression ratio to 4.37%. However, this costs more time as more columns are used in the sort order.

The main performance bottleneck of the implementation is scanning the RowGroup multiple times to convert it to a DataChunk, which now happens when calculating the cardinalities and before sorting. By caching the first scan, we could reduce the time spent scanning by approximately half. Another factor is the value of *wal_autocheckpoint*. A larger value will cause DuckDB to checkpoint less frequently, resulting in a faster time to load the data into the table and the entire file being written to disk.

7

Conclusion

7.1 Research questions

In this thesis, we explored various reordering methods to improve RLE compression. After exploration, we implemented data reordering in the analytical database system DuckDB. Our main research question was: "How can we apply low overhead automatic reordering techniques to improve compression ratios and query performance of an analytical database system?". We will answer this by answering the research questions posed in section 1.1.

How can we best reorder data to optimise RLE compression ratios? How do we identify columns of interest? (e.g. cardinality, column correlation)

Our final implementation was determined by using related work and our exploration. The implementation sorts the columns from low to high cardinality and the rows lexicographically. The strategy is dependent on the dataset and can also decrease the effectiveness of compression. On average, it improved the compression of the Public BI benchmark by 3.23%. We also evaluated a novel method, which uses the combined cardinalities of columns. However, in our experiments, the results were similar to sorting by cardinality while introducing more complexity in the implementation. The Vortex algorithm was also evaluated. Vortex did not improve the compression ratio over the natural order in our experiments. Another downside of this method is the time requirement. In our exploration, Vortex was slower than a lexicographical sort. While this is due to the overhead of our Python implementation, even in the original proposal by Lemire et al. (20) the method is three times slower than lexicographical sorting. Lexicographical sorting can leverage DuckDB's sorting algorithm to perform the sort rapidly. In contrast, implementing the novel Vortex algorithm would likely take significant engineering time to be sufficiently fast

to be included in DuckDB.

At what granularity should these optimisations be applied? Moreover, what is their impact on query performance?

In DuckDB, a row group is approximately 100.000 rows which is the granularity we compared against reordering an entire table. We initially suspected that reordering an entire table would always result in better compression than reordering at a lower granularity. We believed it would result in data forming longer runs, which translates to better compression. However, this turned out to depend on the dataset. Some datasets have better compression when reordering the entire table, and others do not. While we have not verified the exact reason for this, we suspect it happens because we determine the order to sort the columns per RowGroup. For example, sorting on column x might result in longer runs in RowGroup x but not in RowGroup y. Therefore, we conclude that a larger granularity does not equal better compression and depends on the dataset. We have not found a metric to predict if the compression ratio will improve. While we did not evaluate query performance, it is possible that reordering will bring improvements here even if the compression ratio does not improve.

From related work, we determined that better compression could result in faster query performance if the data is operated on in compressed form (9). Compressed execution is currently not supported in DuckDB, making it impossible to have an experimental evaluation for our current implementation. Instead, we focused on the impact reordering had on writing the database file to disk (checkpointing), as this would directly influence the user experience. Our goal was to stay below double the original checkpointing time. On average, our implementation increased this time by 84.62%, staying below our initial goal. It increases by 98.29% when using the benefit analyser. However, the time increases as the number of tables increases. The user can choose to decrease the value of $wal\_autocheckpoint$, causing DuckDB to checkpoint less and reducing the time spent checkpointing.

How do we apply these optimisations automatically, and what is the optimal moment to do this?

During the exploration, we sought a metric to determine whether reordering benefits a dataset. Had we found a reliable metric, we could have used this in the benefit analysis module, as touched on in the design. However, we have not found such a metric. DuckDB already assigns a score to its various compression methods, meaning we do not reorder

tables with no RLE compression. We have therefore chosen the naive approach in our implementation, meaning we always reorder since compression improves on average.

In our design, we determined a moment in the database's lifecycle when transactions had been completed (i.e. no further changes to the data) but before compression. In DuckDB, this moment is during checkpointing. Checkpointing occurs when the user gracefully closes the database, executes the CHECKPOINT command or when a certain amount of data has been changed. We determine the sorting order and then sort based on the selected key columns and proceed through the checkpointing as usual.

Our final implementation considers only columns which would have been compressed with RLE as "key" columns. It then filters those out based on their cardinality. If they have a cardinality of more than 10% of a row group (< 10.000), they are removed from the key columns. Using all columns as key columns and only filtering them on their cardinality increased the average compression ratio. However, it increased the average time to checkpoint as more columns had to be sorted. Thus, the cardinality filter is a parameter which can be optimised, in the future, to achieve a balance between compression ratio and time performance.

How do we apply these optimisations in DuckDB?

Our main contribution was applying the reordering strategy in DuckDB, which meant we had to produce robust, readable code to pass the unit tests and code review process[1]. Reordering can be turned off or on using a PRAGMA. Before reordering, we check whether the data is already on disk to avoid unnecessary rewrites. To sort the data, we must scan it several times to convert it from a RowGroup object to a DataChunk. The cardinalities of the columns are determined with HyperLogLog, which determines the sorting order. This order sorts the columns from low to high cardinality and the rows lexicographically. The DataChunk is then converted back to a RowGroup object, and the checkpointing process is continued. The implementation is extendable with new reordering methods, allowing others to build on this work.

7.2 Future work

This thesis lays the foundation for further optimisation of compression in DuckDB. Implementing the current strategy in an extendable way proved to be a significant engineering effort, leaving many directions for future work which can build on these foundations.

[1] https://github.com/duckdb/duckdb/pull/3913

We mainly focused on RLE compression. DuckDB currently implements dictionary compression and bit-packing as well. Research could determine whether these compression methods could be optimised through reordering and how this would interact with the optimised RLE compression. The DuckDB implementation can also be made more efficient by, for example, reducing the frequency of scans.

A reasonable metric for deciding when to perform reordering is also an open avenue of research. We briefly touched on this during our exploration. Possibly, one could use the average cardinality of all columns to determine whether reordering should be performed. More advanced methods such as machine learning could also be applied, being an upcoming topic in databases.

More research could be done into reordering methods. We also briefly touched on using the combined cardinalities of various columns, with the intuition that reordering correlated columns could result in longer runs.

Finally, we could add a parameter to our implementation which adds a limit to the checkpoint time, allowing the user to find a balance between compression ratio and time.

References

[1] Pedro Holanda. **DuckDB: An embedded database for data science**. https://github.com/pdet/duckdb-tutorial/blob/master/Part%201/duckdb-part1.pdf, 2021. Accessed on 04/07/2022. iii, 5, 11

[2] Elastic. **Frame of Reference and Roaring Bitmaps - Elastic Blog**. https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps, 2015. Accessed on 04/07/2022. iii, 7

[3] David Eppstein. **File:Z-curve.svg**. https://commons.wikimedia.org/wiki/File:Z-curve.svg, 2008. Accessed on 04/07/2022. iii, 9

[4] **Apache Parquet**. https://parquet.apache.org/docs/, 2022. Accessed on 18/07/2022. iii, 10

[5] Jon King Holden Ackerman. *Operationalizing the Data Lake*. O'Reilly Media. Inc., 2019. iii, 10

[6] **Apache Orc**. https://orc.apache.org/specification/ORCv1/. iii, 10, 11

[7] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David Cohen, Timothy Mattson, and Nesime Tatbul. **Self-Organizing Data Containers**. *Memory*, **1**:2, 2022. iii, 22

[8] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. **DeepSqueeze: Deep Semantic Compression for Tabular Data**. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1733–1746, 2020. iii, 25

[9] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. **Column-stores vs. row-stores: how different are they really?** In *Proceedings of the 2008 ACM*

SIGMOD international conference on Management of data, pages 967–980, 2008. 1, 5, 60

[10] Meikel Pöss and Dmitry Potapov. **Data compression in oracle**. In *Proceedings 2003 VLDB Conference*, pages 937–947. Elsevier, 2003. 1, 14

[11] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. **Super-scalar RAM-CPU cache compression**. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006. 1, 6, 7

[12] Daniel Lemire and Owen Kaser. **Reordering columns for smaller indexes**. *Information Sciences*, **181**(12):2550–2570, 2011. 1, 13, 14

[13] Daniel Lemire, Owen Kaser, and Kamel Aouiche. **Sorting improves word-aligned bitmap indexes**. *Data & Knowledge Engineering*, **69**(1):3–28, 2010. 1, 13, 14

[14] Daniel Lemire, Owen Kaser, and Eduardo Gutarra. **Reordering rows for better compression: Beyond the lexicographic order**. *ACM Transactions on Database Systems (TODS)*, **37**(3):1–29, 2012. 1, 6, 15, 34, 36

[15] Bogdan Ghita, Diego G Tomé, and Peter A Boncz. **White-box Compression: Learning and Exploiting Compact Table Representations.** In *CIDR*, 2020. 1, 27

[16] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. *C-Store: A Column-Oriented DBMS*, page 491–518. Association for Computing Machinery and Morgan & Claypool, 2018. 5

[17] Fabian Corrales, David Chiu, and Jason Sawin. **Variable length compression for bitmap indices**. In *International Conference on Database and Expert Systems Applications*, pages 381–395. Springer, 2011. 5

[18] James E Fowler and Roni Yagel. **Lossless compression of volume data**. In *Proceedings of the 1994 symposium on Volume visualization*, pages 43–50, 1994. 6

[19] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. **Compressing relations and indexes**. In *Proceedings 14th International Conference on Data Engineering*, pages 370–379. IEEE, 1998. 7

[20] Chun-Hee Lee and Chin-Wan Chung. **Compression Schemes with Data Reordering for Ordered Data**. *Journal of Database Management (JDM)*, **25**(1):1–28, 2014. 8, 19, 59

[21] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. **Supporting table partitioning by reference in oracle**. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1111–1122, 2008. 8

[22] Alex Petrov. *Database Internals: A Deep Dive into How Distributed Data Systems Work*, page 294. "O'Reilly Media, Inc.", 2019. 8, 12

[23] Jack A Orenstein and Tim H Merrett. **A class of data structures for associative searching**. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190, 1984. 8

[24] Deepak Vohra. **Apache parquet**. In *Practical Hadoop Ecosystem*, pages 325–335. Springer, 2016. 9

[25] **Apache Avro™ 1.11.0 specification**. https://avro.apache.org/docs/current/spec.html, Oct 2021. 9

[26] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. **Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm**. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007. 10

[27] Stefan Heule, Marc Nunkesser, and Alexander Hall. **Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm**. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692, 2013. 10

[28] Mark Raasveldt and Hannes Mühleisen. **Duckdb: an embeddable analytical database**. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019. 11

[29] A. Pinar, T. Tao, and H. Ferhatosmanoglu. **Compressing bitmap indices by data reorganization**. In *21st International Conference on Data Engineering (ICDE'05)*, pages 310–321, 2005. 13, 14

[30] Tan Apaydin, Ali Şaman Tosun, and Hakan Ferhatosmanoglu. **Analysis of basic data reordering techniques**. In *International conference on scientific and statistical database management*, pages 517–524. Springer, 2008. 13, 14

[31] Elaheh Pourabbas, Arie Shoshani, and Kesheng Wu. **Minimizing index size by reordering rows and columns**. In *International Conference on Scientific and Statistical Database Management*, pages 467–484. Springer, 2012. 14

[32] Jane Jovanovski, Maja Siljanoska, and Goran Velinov. **A Genetic Algorithm Approach for Minimizing the Number of Columnar Runs in a Column Store Table**. In Marco Tomassini, Alberto Antonioni, Fabio Daolio, and Pierre Buesser, editors, *Adaptive and Natural Computing Algorithms*, pages 485–494, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 17

[33] Jane Jovanovski, Nino Arsov, Evgenija Stevanoska, Maja Siljanoska Simons, and Goran Velinov. **A meta-heuristic approach for RLE compression in a column store table**. *Soft Computing*, **23**(12):4255–4276, 2019. 17

[34] Jia Shi. *Column Partition and Permutation for Run Length Encoding in Columnar Databases*, page 2873–2874. Association for Computing Machinery, New York, NY, USA, 2020. 19

[35] Adriana Tufă. **Self-Organizing Data Layouts for Databricks Delta**. *CWI*, 2019. 20, 29

[36] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. **Skipping-oriented partitioning for columnar layouts**. *Proceedings of the VLDB Endowment*, **10**(4):421–432, 2016. 21

[37] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. **CORDS: Automatic discovery of correlations and soft functional dependencies**. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658, 2004. 21

[38] Tim Kraska. **Towards instance-optimized data systems**. *Proceedings of the VLDB Endowment*, **14**(12), 2021. 23

[39] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. **LEA: A Learned Encoding Advisor for Column Stores**. In *Fourth Workshop in Exploiting AI Techniques for Data Management*, pages 32–35, 2021. 24

[40] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. **Towards estimation error guarantees for distinct values**. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 268–279, 2000. 31

[41] **SQL Introduction**. https://duckdb.org/docs/sql/introduction, 2022. Accessed on 04/07/2022. 41

[42] Mark Raasveldt. **DuckDB Testing - Present and Future [Keynote address]**. In *9th International Workshop on Testing Database Systems*, 2022. https://youtu.be/BgC79Zt2fPs. 41

[43] **About Sqllogictest**. https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki. Accessed on 04/07/2022. 41

[44] Laurens Kuiper. **Fastest table sort in the West - Redesigning DuckDB's sort**. https://duckdb.org/2021/08/27/external-sorting.html, 2021. Accessed on 04/07/2022. 43

Appendix A

Exploratory results

Dataset	S1e	S1g	S2e	S2g	S3e	S3g	S4e	S4g	S5g
pancreactomy2	-19.11	1.85	-19.11	1.85	-18.71	1.88	-13.42	0.2	0.56
rentabilidad	-1.64	-1.64	-2.37	-2.01	-1.82	-1.64	-11.86	-11.31	NaN
corporations	-0.69	-0.35	-0.87	-0.69	-1.04	-0.52	-1.56	-1.21	-1.21
redfin2	9.87	6.46	NaN	NaN	9.76	7.2	9.13	4.85	NaN
wins	-2.16	-3.09	NaN	NaN	-2.32	-3.09	-7.57	-5.72	NaN
hashtags	10.48	5.41	10.61	6.14	10.55	5.47	10.48	5.41	NaN
cmsprovider	-16.75	2.15	-16.75	2.15	-16.45	2.18	-12.41	0.47	-1.27
generico	-1.54	-5.39	NaN	NaN	-1.19	-5.39	26.27	7.68	NaN
euro2016	1.62	-0.57	1.62	-0.57	1.62	-0.57	-0.19	-0.19	-0.76
medpayment1	-18.77	2.24	-18.77	2.24	-18.77	2.24	-13.92	0.9	1.14
iglocations2	-2.22	-0.17	-2.22	-0.17	-2.22	-0.17	-0.07	0.67	-0.4
commongovernment	14.74	1.01	14.74	1.01	14.74	1.0	-7.05	-8.88	NaN
realestate2	1.35	0.79	NaN	NaN	1.35	0.79	0.09	0.11	NaN
trainsuk2	10.9	-14.42	10.9	-14.36	10.95	-14.42	31.88	-8.76	NaN
uberlandia	8.69	7.14	NaN	NaN	8.56	7.14	-5.62	-2.52	NaN
medicare1	-3.4	1.91	-3.4	1.91	-3.4	1.91	2.4	5.64	NaN
redfin1	9.4	6.08	NaN	NaN	9.29	6.08	8.62	4.56	NaN
pancreactomy1	-19.15	1.79	-19.15	1.79	-19.15	1.82	-13.52	0.2	NaN
uscensus	2.6	1.04	3.13	2.54	2.6	1.04	0.83	-0.87	NaN
telco	7.05	1.88	NaN	NaN	7.1	1.71	-3.01	-3.53	NaN
iglocations1	-11.73	-11.73	-11.73	-11.73	-11.73	-11.73	5.87	5.87	-5.87
realestate1	5.57	4.29	5.57	4.29	5.57	4.29	27.0	-4.28	NaN
food	18.15	11.22	18.15	11.22	18.15	11.22	17.49	13.2	1.65
citymaxcapita	3.28	3.16	3.28	3.16	3.28	3.16	-9.34	-2.4	NaN
hatred	7.89	6.21	7.89	6.21	7.89	6.21	-8.28	3.75	NaN
romance	3.93	0.92	4.39	0.87	3.93	0.92	-0.69	-2.31	0.46
bimbo	55.54	49.47	53.63	48.22	55.54	49.69	9.24	-6.74	-37.31
medicare3	-19.73	1.37	-19.73	1.37	-19.73	1.37	-15.81	-0.46	NaN
yalelanguages	-5.06	-20.25	-5.06	-20.25	-5.06	-20.25	-6.96	-33.53	NaN

tablerosistemapenal	48.33	29.85	48.33	29.85	48.33	29.85	48.33	34.12	NaN
arade	-35.96	-39.41	-36.12	-39.41	-36.12	-39.41	-34.81	-39.57	-18.72
redfin3	10.8	7.16	NaN	NaN	11.34	7.03	9.27	5.08	NaN
mlb	5.33	1.86	NaN	NaN	5.33	1.86	2.38	-0.39	NaN
trainsuk1	-39.89	-5.74	NaN	NaN	-39.89	-5.74	-61.15	-8.67	NaN
medpayment2	-16.89	2.13	-16.93	2.1	-16.62	2.13	-11.38	1.0	NaN
physicians	-18.89	2.18	-18.89	2.18	-18.65	2.18	-14.03	0.84	NaN
salariesfrance	7.02	-0.58	6.36	-0.41	7.35	-0.5	12.14	4.46	NaN
medicare2	-14.53	-0.47	-14.53	-0.47	-14.53	-0.47	-16.04	0.29	NaN
redfin4	10.68	6.36	NaN	NaN	10.34	6.4	8.6	4.38	NaN
iublibrary	0.0	0.0	NaN	NaN	NaN	0.0	0.0	0.0	NaN
eixo	6.24	4.83	NaN	NaN	6.24	4.83	-2.91	-2.26	NaN
mulheresmil	6.59	5.0	NaN	NaN	6.34	5.38	-5.33	0.69	NaN
motos	-6.16	-9.2	NaN	NaN	-6.16	-9.2	26.19	7.25	NaN
taxpayer	-14.49	-0.43	-14.49	-0.43	-14.49	-0.43	-16.0	0.29	NaN
provider	-14.53	-0.4	-14.53	-0.4	-14.53	-0.4	-16.04	0.29	NaN

Table A.1: Exploratory results of all datasets and strategies in percentage. Positive number means file got smaller. S1 is strategy 1, "e" or "g" is for "entire table" or "granular" application level

Appendix B

DuckDB implementation - pre-optimised

We evaluate DuckDB using the standard value of *wal_autocheckpoint*.

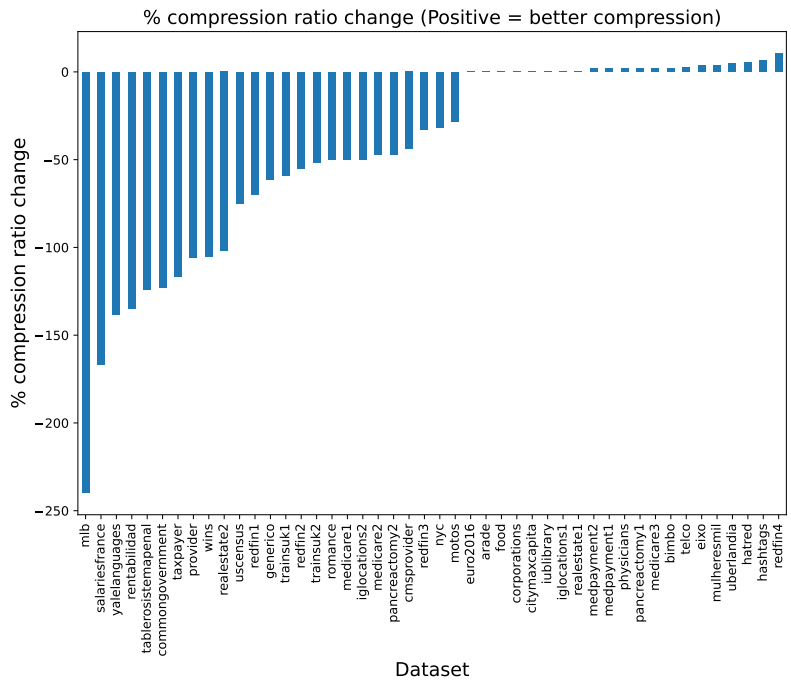

Figure B.1: Real-life - compression ratio change

Figure B.1 shows the compression ratio change after reordering, compared to not reordering. The x-axis shows the percentage change in compression ratio, where a positive number means better compression (i.e. the file size got smaller). The y-axis shows the datasets from the Public BI benchmark. We can see a significant increase in the compressed file

size for more than half of the datasets, especially datasets which consist of many tables.

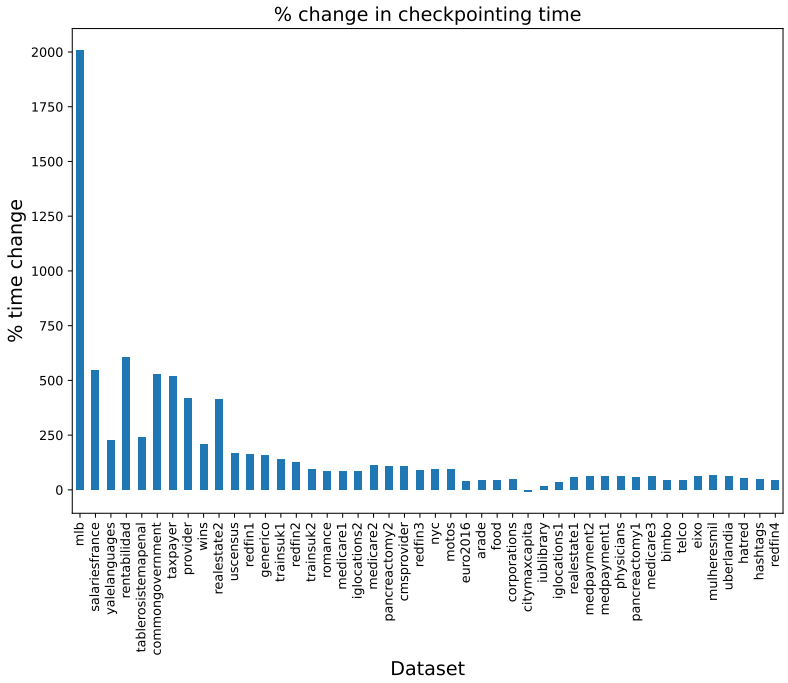

Figure B.2: Real-life - checkpoint time change

Figure B.2 shows the change in time, where a positive value means the experiment took longer. When we compare the change in compression ratio to the time change, we can see that datasets with more tables, such as MLB, take much longer to checkpoint. Since we checkpoint approximately every 10MB, it means that many more checkpoints have to be run. Furthermore, DuckDB automatically checkpoints whenever a table is created with these parameters. As an example, MLB went from 380.35 seconds to 5176.10 seconds.

After these experiments, we investigated the cause of this increase in file size and time and made several changes to the code as detailed in section 5.2.5.