BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

COLUMN-ORIENTED AND IMAGE DATA FORMAT BENCHMARKS

BENCHMARK DATOVÝCH FORMÁTŮ PRO OBRAZOVÁ A TABULÁRNÍ DATA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MARIÁN TARAGEĽ**
AUTOR PRÁCE

SUPERVISOR **Ing. JAKUB ŠPAŇHEL**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

154531

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**TarageĽ Marián**
Programme:	Information Technology
Title:	**Column-oriented and Image Data Format Benchmarks**
Category:	Image Processing
Academic year:	2023/24

Assignment:

1. Study the principles and properties of dataset storage. Focus on the principles and properties that are important in relation to storing tabular and image data.
2. Based on the principles and properties found, design an appropriate set of benchmarks to compare the properties of tabular and image data storage formats.
3. Study the available data formats for storing column-oriented and image data to which your proposed set of benchmarks can be applied.
4. Implement the proposed set of benchmarks for your chosen data formats, including a visualization of the benchmark results to allow comparison of the results of each format.
5. Automate the process of running the benchmarks so that you can run the benchmarks and present the results on a regular basis (for example, using continuous integration).
6. Compare results and discuss options for future development.

Literature:

- Zeng, X., Hui, Y., Shen, J., Pavlo, A., McKinney, W., & Zhang, H. (2023). An Empirical Evaluation of Columnar Storage Formats. *arXiv preprint arXiv:2304.05028*.
- Comparison of different file formats in Big Data
- The Best Format to Save Pandas Data
- Dále dle pokynů vedoucího

Requirements for the semestral defence:

- Completion of the first three points of the assignment.
- The fourth point of the assignment is considerably developed.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Špaňhel Jakub, Ing.**
Consultant:	Ing. Petr Chmelař
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	9.11.2023

Abstract

This bachelor's thesis aims to evaluate different data formats for storing tabular and image data. To accomplish this task, this work designed a new benchmark of data formats. The benchmarks are divided into three benchmark suites. These include the benchmarking of uncompressed tabular formats, compressed tabular formats, and an image storage benchmark. Overall tabular benchmark results suggest that the best tabular data format for speed saving and reading is Feather, and the most memory-efficient format is Parquet. The results of the image storage benchmark show that the fastest image storage is SQLite and the least space is required by PNG format. The results of this work can contribute to a better understanding of how different data formats behave and help to choose the right format for tabular and image data.

Abstrakt

Cieľom tejto bakalárskej práce je ohodnotiť rôzne dátové formáty pre ukladanie tabulárnych a obrazových dát. K zvládnutiu tejto úlohy táto práca navrhuje nový benchmark dátových formátov. Benchmark je rozdelený do troch benchmarkových skupín. Tie zahŕňajú benchmark nekomprimovaných tabulárnych formátov, komprimovaných tabulárnych formátov a benchmark obrazových úložísk. Celkové výsledky tabulárnych benchmarkov naznačujú, že najlepší tabulárny formát pre rýchle ukladanie a čítanie je Feather a najviac pamäťovo efektívny je Parquet. Výsledky benchmarkov ukladania obrázkov ukazujú, že najrýchlejšie úložisko obrázkov je v SQLite a najmenej miesta vyžaduje formát PNG. Výsledky tejto práce môžu prispieť k lepšiemu pochopeniu správania sa rôznych dátových formátov a pomôcť pri výbere správneho formátu pre tabulárne a obrazové dáta.

Keywords

data formats benchmark, data format, data storage features, tabular data, image data, dataset, storing data, visualization

Kľúčové slová

benchmark dátových formátov, dátový formát, vlastnosti ukladania dát, tabulárne dáta, obrazové dáta, dátová sada, ukladanie dát, vizualizácia

Reference

TARAGEĽ, Marián. *Column-oriented and Image Data Format Benchmarks.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jakub Špaňhel

Rozšírený abstrakt

Táto práca sa zaoberá výskumom v oblasti ukladania tabulárnych a obrazových dát. Jedným z rozhodujúcich faktorov, ktorý ovplyvňuje efektivitu uloženia dát, je výber správneho dátového formátu. Preto hlavným cieľom tejto bakalárskej práce je porovnať a zhodnotiť širokú škálu tabulárnych formátov od textových po binárne. Okrem tabulárnych dát sa práca venuje aj obrazovým dátam, ktoré sa v dnešnej dobe často používajú na trénovanie neurónových sietí. V tejto časti bolo porovnaných šesť spôsobov ukladania obrázkov vzhľadom k preddefinovaným metrikám.

K porovnaniu dátových formátov a obrazových úložísk bol navrhnutý nový benchmark. Tento benchmark sa skladá z troch sád, kde každá sa zameriava na inú oblasť ukladania dát. Prvá benchmarkuje dátové formáty na všeobecných tabulárnych dátach bez kompresie, druhá pre vybrané dátové formáty povoľuje kompresiu a posledná skúma najvhodnejší spôsob uloženia obrazových dát.

K správnemu vyhodnoteniu sú použité hlavne nasledujúce metriky: čas potrebný k uloženiu dát, čas potrebný k načítaniu dát z disku, veľkosť výsledného súboru po uložení a maximálna alokovaná pamäť v RAM pri ukladaní aj čítaní. Benchmarky boli vykonané na rôznych dátových sadách. Jednu kategóriu testovacích dát tvoria synteticky vygenerované dáta podľa definovanej schémy. Vytvorenie umelých dát zabezpečuje syntetický, parametrizovateľný generátor dát. Okrem týchto dát boli použité aj statické dátové sady, ktoré boli načítané pred spustením benchmarkov. Tieto sa využili hlavne v prípade benchmarkov obrazových dát, kde boli použité dátové sady CIFAR-10 a ImageNet-100.

Výsledky meraní benchmarkov sú prehľadne spísané v tabuľkách v tejto práci. Celkovo sa práca snaží zistiť najvhodnejší dátový formát alebo obrazové úložisko vzhľadom na určité kritérium. Finálne výsledky benchmarkov tabulárnych dátových formátov naznačujú, že v prípade potreby čo najrýchlejšieho ukladania a načítania dát je najvhodnejšie zvoliť dátové formáty Feather alebo Pickle. Čo sa týka kritéria najmenšieho zabraného priestoru na disku, z benchmarkov vyšli najlepšie dátové formáty Parquet a ORC. V kompresnom benchmarku dosiahol najlepší kompresný pomer formát HDF5. Výsledky benchmarkov obrazových úložísk ukazujú, že najrýchlejší spôsob ukladania a načítania obrázkov poskytujú SQLite a LMDB. Najmenej miesta na disku zaberie uloženie obrázkov do samostatných PNG súborov s metadátami uloženými v CSV súbore.

Výsledky tejto práce môžu byť použité v prípade výberu vhodného dátového formátu alebo obrazového úložiska. Ak má užívateľ vybrané kritériá, ktoré sú pre neho dôležité, môže spustiť benchmark aby mu odporučil najvhodnejší dátový formát. Vždy je však potrebné uvažovať aj nad kontextom, v akom budú dáta používané.

Okrem tohto môžu výsledky pomôcť k lepšiemu pochopeniu, aké atribúty vo významnej miere zvyšujú alebo znižujú efektivitu dátových formátov. Na základe týchto zistení sa môžu v budúcnosti navrhnúť nové moderné dátové formáty, ktoré dokážu zvládať aj veľmi veľké dátové sady, ktoré sa používajú hlavne v oblastiach ako strojové učenie alebo počítačové videnie. Dúfam, že táto práca prispeje do oblasti skúmania výkonnosti jednotlivých dátových formátov.

Column-oriented and Image Data Format Benchmarks

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Jakub Špaňhel. The supplementary information was provided by Ing. Petr Chmelař. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Marián Tarageľ
May 5, 2024

Acknowledgements

I would like to convey my gratitude to Ing. Jakub Špaňhel for his supervision. I also express my thanks to my consultant Ing. Petr Chmelař. Both of them provided me with support and advice during the work on this thesis. Last but not least, I would like to thank the external submitter, the Innovatrics company, for their professional help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The area of effective tabular and image data storage is attracting considerable interest because the data size is growing rapidly. These huge datasets, used for machine learning purposes, need data formats that will match their needs.

In recent years, articles [18, 29] evaluated column-oriented data formats Parquet, ORC, Arrow, and Feather. Those papers extracted properties of the real-world dataset, created workloads, and designed benchmarks. Concerning effective image storage, some research has also been done. For example, the conference paper [17] has conducted a benchmark of several image storage options on three different image datasets. Although some research has been done in the field of data formats benchmarking, there is still space for deeper research.

This bachelor's thesis focuses on finding key features that affect data formats. Based on those principles, it will extensively compare popular and state-of-the-art data formats. This thesis should answer the question which data formats are the best based on some metrics and why. Furthermore, it will evaluate different possible ways to store images. To answer those questions, the benchmark of data formats is the way to fairly and comprehensively evaluate each of them. The results of the benchmarks will be visualised in the graphs.

The work is divided into several chapters that will first describe theoretical key points, then describe the methodology of the benchmarks, and finally present an evaluation of the results. Chapter 1 is a general introduction to the research topic with related work. Chapter 2 will study the principles and features that affect data storage formats. Each data format that will be benchmarked will be described in Chapter 3. This chapter also lists different options for storing image data. Those will be benchmarked in the image storage benchmarks. Chapter 4 will propose a set of benchmarks and how they were conducted. Chapter 5 will describe the design and implementation of the benchmarks that were introduced in the previous chapter. Chapter 6 will discuss the results of the benchmarks and their implications. Chapter 7 will conclude this research and its results.

Chapter 2

Principles and features of storing tabular data

This chapter will introduce important principles and features of tabular data storage. These principles affect the efficiency of data formats. Different data formats implement various principles described in this chapter. One of the most important are the internal layout, encoding, compression, and data types. Table 2.1 summarises the possible options for each feature. The data format features taxonomy was taken from article [29].

Table 2.1: Summary of data storage features taxonomy.

Internal layout (2.1)	NSM, DSM, PAX
Encoding (2.2)	DICT, RLE, Delta, FOR, BP, Huffman, ...
Compression (2.3)	Gzip, Zlib, LZO, Bzip2, Snappy, LZ4, zstd, Brotli, ...
Data types (2.4)	defined type system, without support for data types
Indexes (2.5)	Zone Map, Bloom Filter, Page Index, ...

Tabular data is a type of data that can be represented by a table. A table consists of rows and columns, as shown in Figure 2.1. A row is a record with a defined scheme, based on a table header. The scheme indicates the data type of each column. The table is a collection of records. In the Python Programming Language, this data structure is represented by `pandas.DataFrame`.

Table Representation		
Column 1	**Column 2**	**Column 3**
A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4
A5	B5	C5
A6	B6	C6

Figure 2.1: Table Representation of DataFrame. Scheme taken from the webpage [21].

2.1 Internal layout

Internal layout defines how the data are physically stored on a disk. There exist three major approaches to how tabular data can be stored, *N-ary Storage Model* (NSM), *Decomposition Storage Model* (DSM) and *Partition Attributes Across* (PAX). Those layouts differ in orientation on rows, columns, or a combination of both. All three will be part of this benchmark, for example NSM in CSV or Avro, DSM in JSON or Feather and PAX in Parquet or ORC.

N-ary Storage Model

The N-ary Storage Model, or row-oriented approach, is the most simple layout that stores all records sequentially. Its main disadvantage is that encoding algorithms and compression are ineffective because mixed-typed tuples are stored in a line. It is used in popular data formats, for example CSV or XML. Figure 2.2 visualises this type of layout.

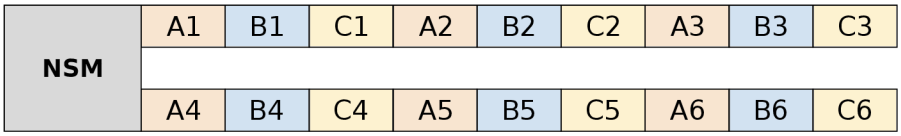

Figure 2.2: Layout of N-ary Storage Model. Scheme taken from the webpage [21].

Decomposition Storage Model

The decomposition storage model is a fully decomposed storage layout introduced by Copeland and Khoshafian (1985) in the article [9]. Column-oriented fashion stores tabular data by columns, as can be seen in Figure 2.3. As pointed out by Abadi *et al.* [1], this layout significantly improves the similarity of the neighbouring values on a disk, thus improving the effectiveness of compression algorithms. On the other hand, back reconstruction to table format takes more time, because more seeking is needed, as stated by Ailamaki *et al.* [2].

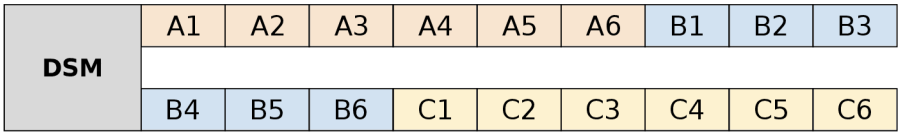

Figure 2.3: Layout of Decomposition Storage Model. Scheme taken from the webpage [21].

Partition Attributes Across

The most modern data storage layout is Partition Attributes Across, which was presented by Ailamaki *et al.* (2001) in the proceedings [2]. It is a combination of NSM and DSM. The columns are divided into column chunks and grouped in row groups. This approach combines inter-record spatial locality and low record reconstruction cost. Ailamaki's study also noted that PAX is performing faster query executions than DSM and incurs 75% less data cache stall time than NSM. Data formats like Parquet or ORC internally follow this storage model. Figure 2.4 shows the PAX storage layout.

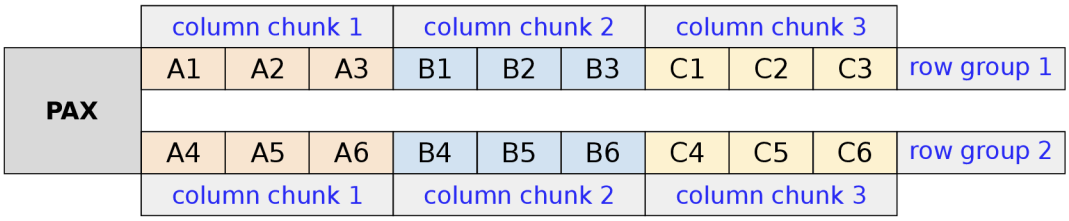

Figure 2.4: Layout of Partition Attributes Across. Scheme taken from the webpage [21].

2.2 Encoding

Applying lightweight encoding algorithms can be highly beneficial to data formats. It can reduce the overall file size by compressing the input data. Some data formats use multiple encoding schemes to get an even better compression ratio. Furthermore, encoding can decrease the time needed for I/O operations, as stated in article [1]. However, decoding will require CPU time.

There exist two types of data formats, plain text and binary. Text formats, such as CSV or XML, do not use any special encoding algorithm. Typically, they are coded to Unicode or ASCII. They can store only text characters, which means they do not have support for binary data, such as images. On the other hand, binary formats use a variety of encoding algorithms to store data more effectively. Common ones are described below.

Dictionary Encoding (DICT) stores frequent data values with fixed-length codes. For example, „apple“ is 0, and „banana“ is 1. The key-value dictionary is used to store this mapping. It works well when the number of distinct values in the column is low and those values have high frequencies. The drawback is that the dictionary must be included in the encoded data. Dictionary encoding cannot be used effectively when the data has many unique values.

Run-Length Encoding (RLE) replaces the consecutive sequences of the same value, with the tuple that contains the value and the number of repetitions. This is particularly effective in column-oriented formats, because there is a higher probability of repetition of the same data value, as was stated in article [1]. This encoding can be combined with others, leading to hybrid encoding, for example, RLE/Dictionary or RLE/Bit-packing.

Delta Encoding or Delta binary packed as presented by Lemire and Boytsov [16] uses the differences, deltas, between the encoded value and the previous value. For example, the sequence 4, 3, 7, 11, 10, 14, 13, 17 would be stored as the sequence 4, -1, 4, 4, -1, 4, -1, 4, which consists of only 4 and -1. Parquet uses this encoding on numerical, but also string data. When the prefix of the string is the same as before, it will store only the prefix length and the rest of the string.

Frame-of-Reference (FOR) from Goldstein *et al.* [11] has a similar approach to delta encoding. FOR can make relatively high numbers smaller, which means they can be coded on fewer bits. For example, numbers in the sequence 135, 141, 144, 148, 149, 152, 156, 160 range from 135 to 160. That means that 135 can be subtracted from each number and the sequence will be 0, 6, 9, 13, 14, 17, 21, 25. Numbers from 0 to 25 can be represented on 5 bits in binary code, instead of 8 bits. What is needed to take into consideration is the fact that the number that was subtracted must be included in the encoded data, and also the fact that the numbers are encoded on n bits.

Bit-Packing (BP) encodes the numerical data into fixed-length codes, by cutting off the leading zeros. To give an example, the sequence 1, 2, 3, 4, 5, 6, 7 can be encoded on 3 bits as follows: 001, 010, 011, 100, 101, 110, 111. Originally, it would be encoded on 8 bits with 5 bits of redundant zeros.

Huffman Encoding, introduced in article [13], is a method that can find the minimum redundancy prefix code. It is an entropy coding, which means it works with probabilities of input symbols. Huffman encoding is used in many compression formats, such as ZIP, JPEG or MP3.

2.3 Compression

As was noted by Abadi *et al.* [1], some data formats use general-purpose block compression codes, to further decrease the file size. This feature also comes with disadvantages. More time is needed to load data into memory because decompression takes CPU time. Different compression levels can be set to increase the speed, but the trade-off is a worse compression ratio. Furthermore, enabling compression can be detrimental to the end-to-end query speed of data formats. This issue was experimentally proved by Zeng *et al.* [29].

An overview of popular compression algorithms is given in Table 2.1. Older algorithms, such as Gzip (1992) or Zlib (1995), have native support in programming languages. Modern compression algorithms like zstd (2015) need external libraries, but they often lead to better compression results.

2.4 Data types

The majority of data formats support various data types. However, a few formats do not have this support because they are encoded in plain text. Data types are defined in a type system, which can be highly complex or simple with only basic types. As stated in the presentation by Andrew Pavlo [22], the type system can be either physical or logical. A physical type is a machine-specific byte representation, such as IEEE-754. Logical data types are defined by the data formats. Those types are mapped to physical types.

Data types can be primitive or composite. Common primitive types are `Boolean`, `Integer`, `Float` and `Double`. These types can be represented on various numbers of bits, for example, an integer can be `Int8`, `Int16`, `Int32` or `Int64`. Composite data types are compounded from several primitive types. An example of composite data types can be `Struct`, `List`, `Map`, or `Union`. The data format can create auxiliary types to store a specific kind of data.

2.5 Indexes

Indexes and filters can significantly boost the query performance of a data format. This section will describe *Zone map* and *Bloom filter*. Both of those indexes are supposed to ensure faster queries. The zone map defines ranges of values for columns. Bloom filter is a special data structure based on hash functions. Indexes are optional for most data formats and they can be omitted.

Zone map

Zone map stores maximal and minimal values in a part of a file, known as a zone. If the query value is larger or less than this range, then the whole zone can be skipped. This can significantly speed up query execution. It works best when the similar values are near each other or the columns are partially sorted. Typically, a zone map stores a value range for more zones.

Bloom filter

Bloom filter [4] is a space-efficient, randomised data structure that answers the question whether the element is in a set. It uses a hash functions to insert elements into the bit array. The results of the membership query can generate false positives. This is not a big issue if the probability of error is small enough. The advantage of the Bloom filter is its time and space complexity. The time complexity of the insertion and search is $\mathcal{O}(k)$, where k is the number of hash functions. The space complexity is $\mathcal{O}(n)$, where n is the length of the bit array. There exist many variations of the Bloom filter, such as dynamic, compressed, spectral, or split block Bloom filters. An example of a Bloom filter is shown in Figure 2.5.

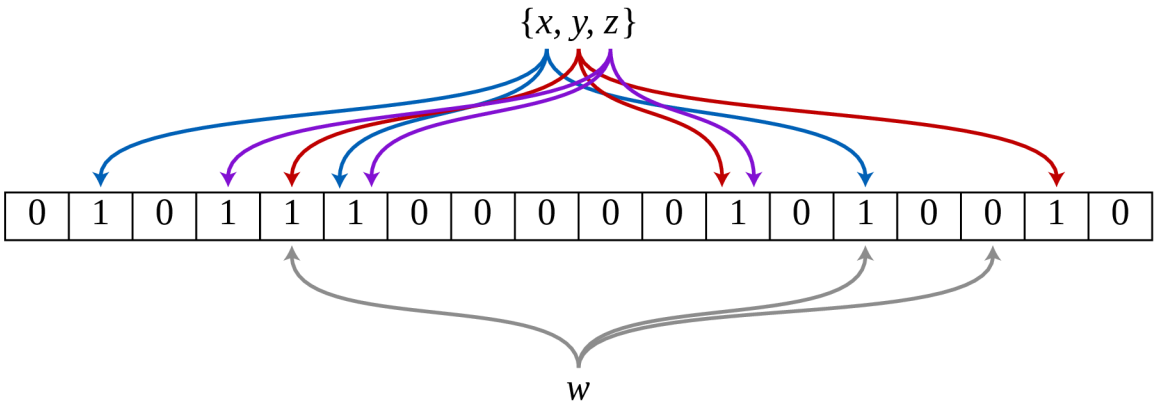

Figure 2.5: Example of a Bloom filter with three elements x, y and z. Element w is not in the set, because it has 0 as an output of the hash function. Scheme taken from slides [10].

Chapter 3

Data formats

Based on the principles described in Chapter 2, this chapter will present tabular data formats in Section 3.1 and list options for storing image data in Section 3.2. Data formats and image storages, that are included in this chapter, will be benchmarked in Chapter 4.

3.1 Tabular storages

The data formats that will be presented in this section were chosen because they are either state-of-the-art or they are generally popular and widely used. Every format will be described separately. Table 3.1 summarises the key attributes of each file format.

Table 3.1: Overview of tabular data formats properties. Properties taken from the articles [18, 29] and data formats specifications.

	Internal layout	**Encoding variants**	**Compression**	**Data types**
CSV	NSM	Text	No	No
JSON	NSM, DSM	Text	No	Yes
XML	NSM	Text	No	No
HDF5	Hierarchical	RLE, Huffman	LZ4, Snappy, Zlib, zstd	Yes
Parquet	PAX	DICT, RLE, Delta, BP	Snappy, Gzip, LZO, Brotli, LZ4, zstd	Yes
Feather	DSM	DICT	LZ4, zstd	Yes
ORC	PAX	DICT, RLE Delta, FOR, BP	Zlib, Snappy, LZO, LZ4, zstd, Brotli	Yes
Pickle	Stream	Binary code	No	Yes
Excel	NSM	multiple XMLs	Zip	Yes
Lance	PAX	VLB, DICT	No	Yes
Avro	NSM	Binary, JSON	Deflate, Snappy	Yes

Comma-separated values

Comma-separated values (CSV) is a type of delimiter-separated file format which uses a comma as a separator. Each record is located on a separate line and divided by commas

into fields. The comma can be replaced by other characters, common ones are tab, colon or semicolon. Optionally, the fields can be enclosed in double-quotes. CSV structure can be ambiguous due to the lack of adherence to official standards. This often leads to improperly formatted CSV files. An example of a formal specification is RFC 4180 [24], which defines the MIME type `text/csv`.

CSV is one of the most popular data formats because it is encoded in human-readable plain text and can be imported to spreadsheet editors, like Microsoft Excel or LibreOffice Calc. On the other hand, CSV is not suitable for large datasets with millions of records, because it does not use any special encoding to compress the data.

JavaScript Object Notation

JavaScript Object Notation (JSON) [38] is a lightweight, plain text and language-independent data format. It is based on the JavaScript language, standard ECMA-262 [30]. JSON is popular because it is easy to parse in any programming language and is also human-readable. JSON is common in REST API as a data-interchange serialisation format.

Although JSON is a text format, it supports basic data types such as boolean, string, integer and float. JSON uses two main data structures, *Object* and *Array*. Objects and arrays can be nested into each other. Deep nesting greatly decreases the speed of deserialisation, because a recursive approach is needed.

As stated on the JSON's official webpage [38], the object is a set of key-value pairs. The object is enclosed in braces, and each pair is separated by a comma. In programming, it can be realised as an object, record, struct, dictionary, hash table, keyed list or an associative array. The array is an ordered list of values. The array is enclosed in brackets and each value is separated by a comma. In programming, it can be realised as an array, vector, list or a sequence.

JSON is a column-oriented data format, but it can be oriented on rows with JSON Lines. JSON Lines is a line-delimited JSON format. On each line, it stores one record in the form of a JSON object. JSON Lines supports parallel I/O operations. This is the approach that I will benchmark.

Extensible Markup Language

Extensible Markup Language (XML) [6] is an open, free data format, standardised by the World Wide Web Consortium. It was first introduced in 1998. It is designed as a plain-text format for application data interchange, which means it is easily understandable for both machines and humans. XML is a subset of the Standard Generalized Markup Language (SGML), which is a meta-language for defining markup languages. Its main purpose is to separate data and style. Extensible Stylesheet Language Transformations (XSLT) can transform XML to other formats, such as an HTML web page or a PDF document.

Unlike CSV or JSON, XML syntax is complex and verbose. It can express almost any kind of data as a tree. The basic syntactic structure is an *element*. The element consists of a start tag and an end tag. The start label is enclosed in a less-than sign and a greater-than sign, `<name>`. The end tag is similar to the start tag, but it adds a slash after a less-than sign, `</name>`. If the element is empty, it can be shortened as `<name/>`. Elements can have attributes to further specify the data. XML document must have one root element in which other elements are nested. Optionally, it can have an XML declaration on the first line,

for example `<?xml version='1.0' encoding='utf-8'?>`. If the XML document follows the rules defined in specification [6], then it is called well-formated.

XML is highly criticised because it is verbose and redundant. Nowadays, it is replaced by other modern data formats, such as JSON or YAML. They can store the same data more effectively with less syntactic sugar. This leads to a smaller total file size.

Hierarchical Data Format 5

Hierarchical Data Format 5 (HDF5) [44] is a self-describing binary data format. It has a more complex internal layout that can be used on a wide variety of heterogeneous data. Figure 3.1 shows a simplified layout of an HDF5 file. As noted by Byna *et al.* [7], HDF5 is popular in the scientific world because it is suitable for a high volume of experimental and observational data (EOD). HDF5 also allows data compression. An overview of supported compression algorithms is given in Table 3.1.

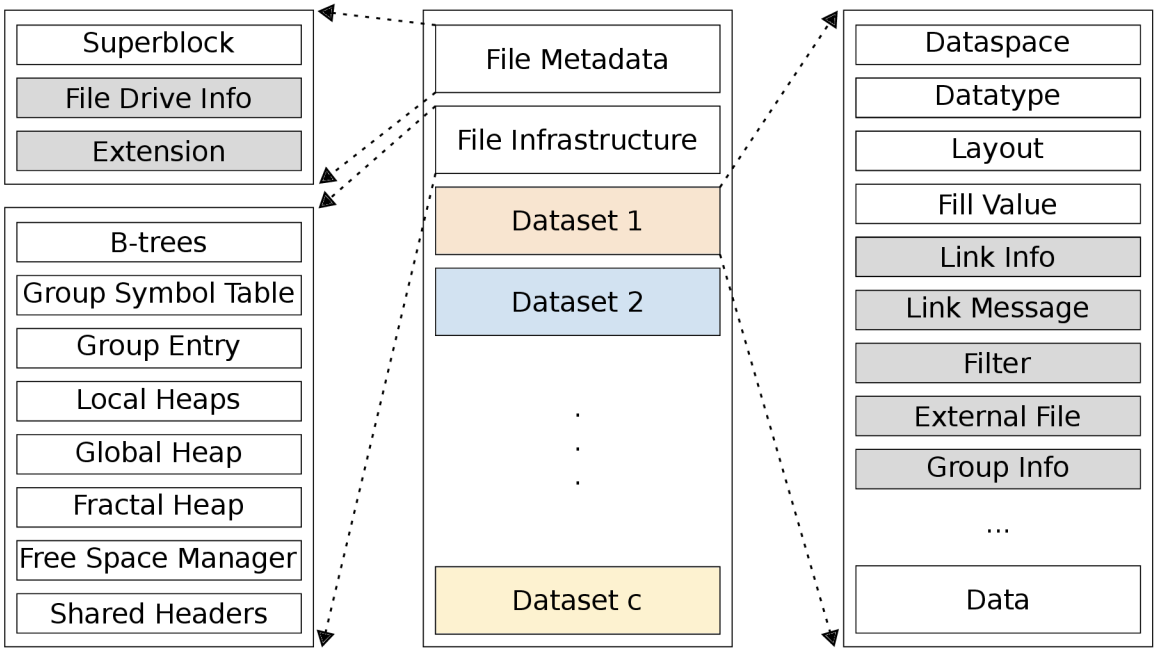

Figure 3.1: Simplified internal layout of HDF5 format. Gray parts are optional.

Moreover, HDF5 I/O operations can run in parallel processes. As experimentally proved by Xie *et al.* [28], to obtain superior performance, it is important to select the right parameters for this feature. Enabling collective I/O on file metadata write, placing the metadata at the file start, and flushing the metadata cache at the file close deliver the best results for HDF5 write.

Furthermore, HDF5 is portable and multi-platform. It is an open-source project, with an official API provided in Java, C++, C, and Fortran. External libraries are developed in Python, Perl and other programming languages.

As is visible from HDF5 specification [37], HDF5 layout is similar to the file system structure. It consists of three main components: *dataset*, *group* and *attribute*. Dataset is mainly used to store application data. Each dataset can have its own structure. Typically, each column is stored in a separate dataset. Groups organise similar objects together. They can be seen as directories. Attributes annotate datasets, groups and other objects.

HDF5 file also includes metadata that describe the application data. In the begging of every HDF5 file must be a superblock which defines the file itself. Descriptive metadata are stored before each dataset, for example for updating the location of the dataset. Datasets, groups, and attributes are HDF5 high-level objects. At the low level, HDF5 is made of a superblock, b-tree nodes, heap blocks, object headers, object data and free space.

Two different formats of HDF5 will benchmarked – fixed and table. Fixed format performs faster save and write operations, whereas table offers greater flexibility and allows operations, such as searching or selecting.

Apache Parquet

Apache Parquet [35] is the first presented data format that uses the PAX model, designed by X (Twitter) and Cloudera. It was inspired by the Google Dremel model, which was presented by Melnik *et al.* [20]. Parquet uses record shredding and the assembly algorithm. Parquet is one of the most memory effective data formats because it uses a wide variety of encoding algorithms. With a combination of compression codes, it can greatly reduce file size. The supported encoding variants and compression algorithms are listed in Table 3.1.

The internal layout of the Parquet can be seen in Figure 3.2. It follows the PAX format, which means division into *Row Groups* and *Column Chunks*. A column chunk typically consists of multiple *Pages*. Each page has defined encoding and compression. After the application data, *Bloom Filter* and *Page Index* can be included. Parquet uses the Split Block Bloom Filter, which was introduced by Jim Apple [3], and takes advantage of modern SIMD instructions. The Parquet file ends with a *Footer* and its length. The footer contains information about the file and about row groups. File metadata may include information such as the version or the schema. Row groups metadata can be, for example, the offset and the type.

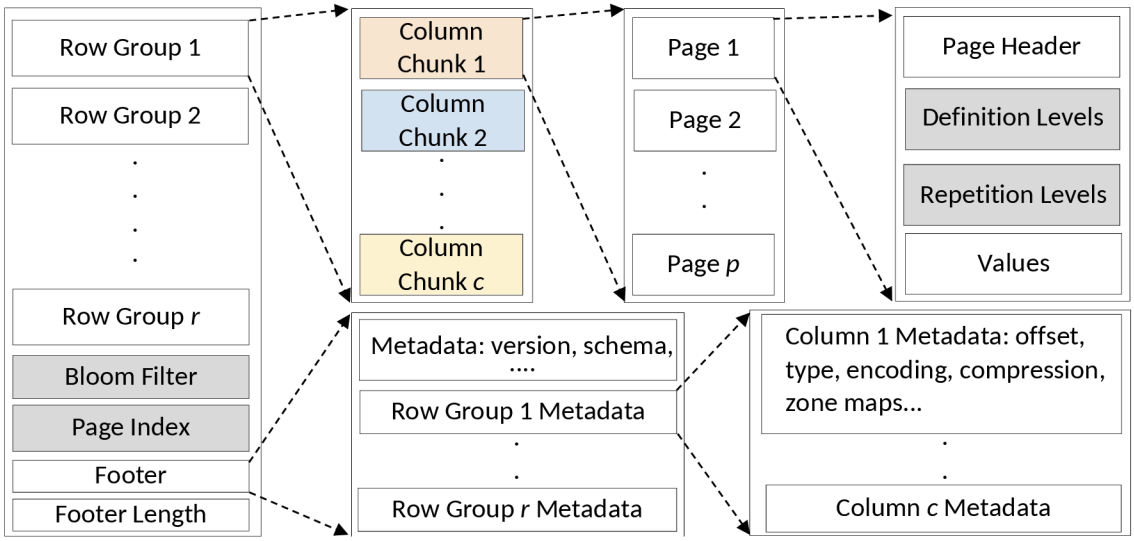

Figure 3.2: Internal layout of Parquet data format. Gray parts are optional. Scheme taken from article [29].

Feather

Feather [19] is a binary column-oriented data format. It was created as a file format for easy and fast transfer of data between Python's DataFrame and R's DataFrame. It has an official API written in Python, R, and Julia. Feather is fast, interoperable, and has high performance in read and write operations. For strings, it uses dictionary encoding. As mentioned on Apache Arrow's documentation webpage [36], there exist two versions of Feather. *Feather V1* is a legacy version. This version lacks some features of V2. It does not support compression and cannot store all Arrow data types. *Feather V2* is represented by the Arrow IPC file format. It allows all Arrow data types and also compression codes LZ4 and ZSTD. This version will be used for the benchmarks in Chapter 4.

Optimized Row Columnar

Optimized Row Columnar (ORC) [12, 45] is a fast, small, binary data format. It was created as a part of the Apache project, where it replaces the old RCFile in the Hive data warehouse. ORC supports many encoding schemes and compression algorithms. An overview is given in Table 3.1. Moreover, ORC follows ACID rules, which are common for database transactions. It uses Zone Maps and Bloom Filters for query speed-up. In contrast to Parquet, ORC supports a rich variety of data types. These include primitive types, but also complex data types, such as `Map` or `Union`.

ORC file structure implements the PAX data storage model. ORC layout scheme is shown in Figure 3.3. ORC consists of *Row Groups* and *Column Chunks*. The row groups are stored sequentially and internally divided into column chunks. Each row group has multiples *Indexes*. Zone maps are required, but Bloom filters are optional. ORC file ends with a *Footer*. The footer stores the file metadata and information necessary for each row group, such as their offset or data length.

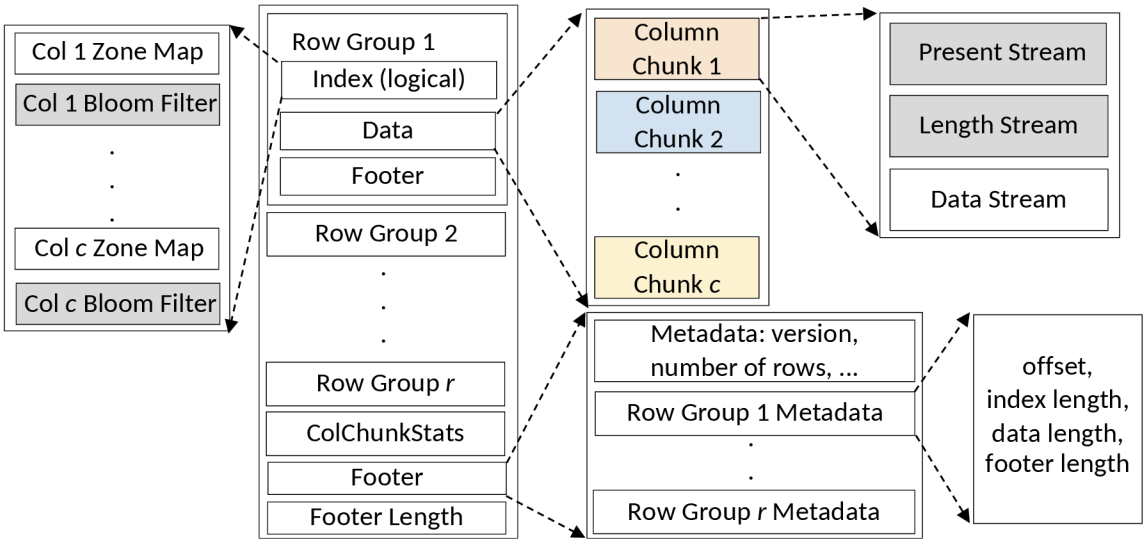

Figure 3.3: Internal layout of ORC data format. Gray parts are optional. Scheme taken from article [29].

Pickle

Pickle [42] is a binary serialisation data format. It is Python-specific and is not supported in other programming languages. The layout of Pickle is a compact byte stream. Pickle does not use special encoding or compression, which means that it is not suitable for big data storage. On the other hand, its I/O performance is great and works best on `numpy.ndarray` and `pandas.DataFrame`, as was pointed out in blog [26]. The process of creating a Pickle file is called pickling. Pickle can use six different protocols for serialisation. Higher versions of the protocols require newer versions of Python. Unpickling the data is not secure and can result in the execution of arbitrary code.

Office Open XML Workbook

Office Open XML (OOXML) [41] is a file format that is used in Microsoft Office. It is defined by the standard ECMA-376 [31]. It has four defined extensions: *WordprocessingML* for `.docx` documents, *SpreadsheetML* for `.xlsx` spreadsheets, *PresentationML* for `.pptx` presentations and *DrawingML* for drawing shapes and schemes. OOXML file consists of multiple XML files that are compressed by Zip to one archive. SpreadsheetML, commonly known as Excel, has the following file structure:

- `[Content_Types].xml` – Required XML file, which is located in the archives root. It links other XML files with their content type.
- `_rels/` – Folder with one file, called `.rels`. This file defines relationships between other XML files and external resources.
- `docProps/` – Folder with two files, `app.xml` and `core.xml`. Both of them define document properties.
- `xl/` – Main folder with workbook definition and at least one or more wokrsheets. It also contains the theme and style of the spreadsheet.

Lance

Lance [39] is a modern columnar data format, used primary to store ML data. These can include images, videos, 3D points clouds, but also tabular data. Lance supports automatic versioning of dataset, that means it is possible to access old versions of the dataset. Lance data format structure is organised in a separate directory. It includes data directory, manifest file with latest version of the dataset, directory with manifests for older versions of the dataset, indexes and deletion files. Data layout employs the PAX format, it divides rows into batches and each batch consists of more pages. After data, there follows metadata with information about batch length or page table position. Then optionally manifest and finally footer.

Apache Avro

Avro [34] is a binary, row-oriented serialisation data format. Avro is heavily based on a schema, which is defined in JSON. Since the schema is present in both read and write, there is no need to determine the data types of the columns. This can reduce overhead and speed up the process of storing and loading data. Avro supports a rich choice of data types, from primitive to complex, such as maps or unions. Furthermore, it specifies two encodings:

Binary and JSON. Binary is used more, because it is faster and smaller. JSON encoding can be useful for web-applications or debugging. Avro also supports compression codecs – DEFLATE is required and SNAPPY, BZIP2, XZ and ZSTD are optional.

3.2 Image storages

After introducing tabular data formats in previous section, this section will introduce image data formats. This set of image data storage was chosen to cover most of today's used image storage options. It includes the PNG format, Base64 encoding, HDF5, Parquet, SQLite, and LMDB. Table 3.2 shows the image storage options and multiple variants of the same storage type. Only a subset of those will be benchmarked.

Table 3.2: Options of storing image data.

Image formats	PNG, JPEG, SVG ...
Encoding	Base64, BinHex, Quoted-printable ...
Binary formats	HDF5, Parquet, ORC, ...
Key-value DB	LMDB, LavelDB, TileDB, ...
BLOB	SQLite, PostgreSQL, ...

Portable Network Graphics

Portable Network Graphics (PNG) [5] is a single raster image storage format. It can store greyscale or truecolor RGB (red, green, blue) images with an optional alpha channel. PNG files are well compressed because they use the lossless data compression algorithm DEFLATE, which is a combination of LZ77 and the Huffman encoding. The PNG format is about 10% to 30% more compressed than the GIF format, as stated by Shivashetty and Rajput [25]. On the other hand, PNG encoding takes more time because the whole process consists of more parts and includes compression.

The drawback of PNG is that it is a single image storage format and more images require multiple files. For example, the ImageNet-1k dataset used for training neural networks consists of approximately 1,3 M images, and each of them would be stored in a separate file. Moreover, most of the image datasets also include labels or metadata about the images. PNG is only an image format, and other data cannot be stored along them. This benchmark will store metadata into a CSV file with column ID, which will link the image with the metadata.

The PNG format layout begins with a PNG signature and continues with different chunks. Four chunks are critical and must be included in the PNG file. Other are optional and can be omitted. Required chunks are `IHDR`, `PLTE`, `IDAT`, and `IEND`. `IHDR` is the header of a PNG datastream with information about image height, width, bit depth and more. `PLTE` links the colour palette with a PNG image. `IDAT` contains compressed and filtered image data. `IEND` or image trailer, indicates the end of the PNG datastream. Figure 3.4 visualises the layout of the PNG image format.

Base64

Base64 [14] is an encoding of arbitrary binary data to a subset of printable ASCII characters. Base64 algorithm creates a group of 3 bytes (24 bits). This group is divided into 4 concate-

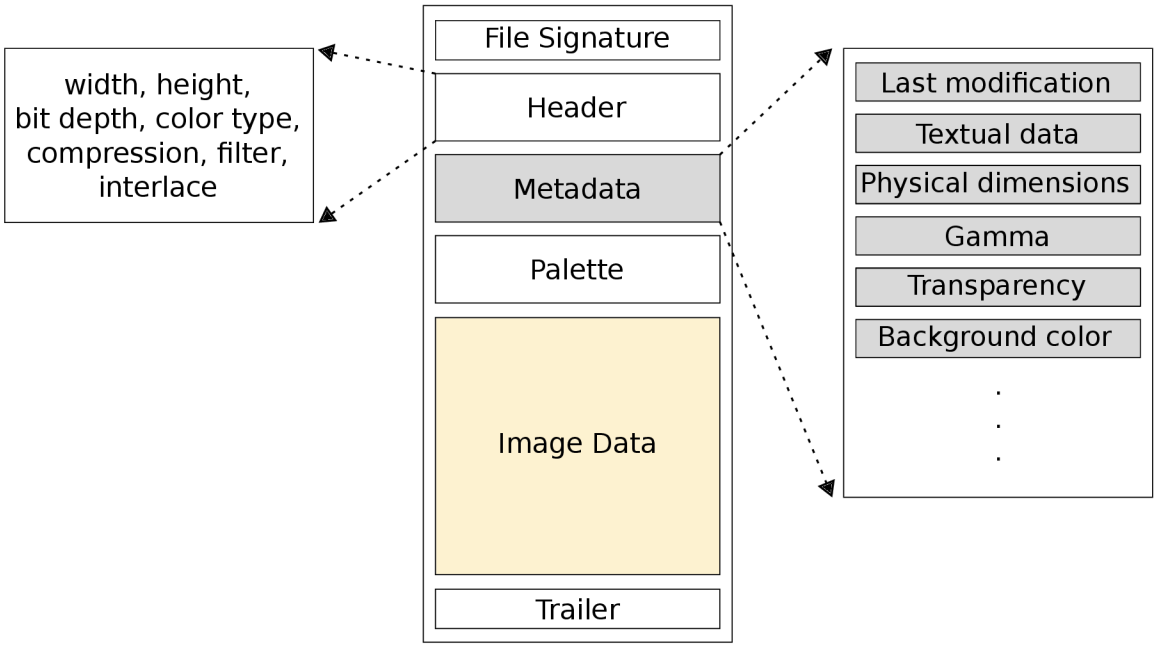

Figure 3.4: Internal layout of PNG image format. Gray parts are optional.

nated 6-bits groups, each of them is an index to the table of characters. If the end of binary data is not aligned to 24 bits, then it is padded with zeros, and a special character '=' is used for representation of them. The output of Base64 is an ASCII string. Base64 is considered not memory-efficient because it always causes 33 % lengthening of the input: $\frac{32}{24} = 1,\overline{3}$. This benchmark will encode the images to Base64 strings and store them in a single CSV file with labels on the same row.

Binary data formats

Images can be represented as n-dimensional arrays (width, height, channels). These arrays are flattened, encoded into binary code, and stored into a file. It is necessary to include the shape of the image and the color palette, because image datasets can contain images with different shapes, RGB or greyscale. The back reconstruction is performed by array reshaping. This is another option for storing images in binary data formats. This thesis will benchmark HDF5 and Parquet binary data formats in image data storage benchmarks. These two formats were previously described in more detail.

LMDB

Lightning Memory-Mapped Database (LMDB) [8] is a read-optimised database. Originally, it was slapd backend for OpenLDAP. LMDB is a key-value storage which can store arbitrary data, including binary. As the Figure 3.5 shows, internally LMDB implements B+ tree with Multi-Version Concurrency Control. The name „Memory-Mapped“ comes from the fact that LMDB is mapping the whole database to the virtual memory, and every data access returns a direct pointer to the data, to avoid intermediate copies. LMDB demonstrates high efficiency in data reading by employing the Single-Level Store. This concept groups the entire computer memory hierarchy into a single address space, allowing LMDB to retrieve the whole database in a single read operation.

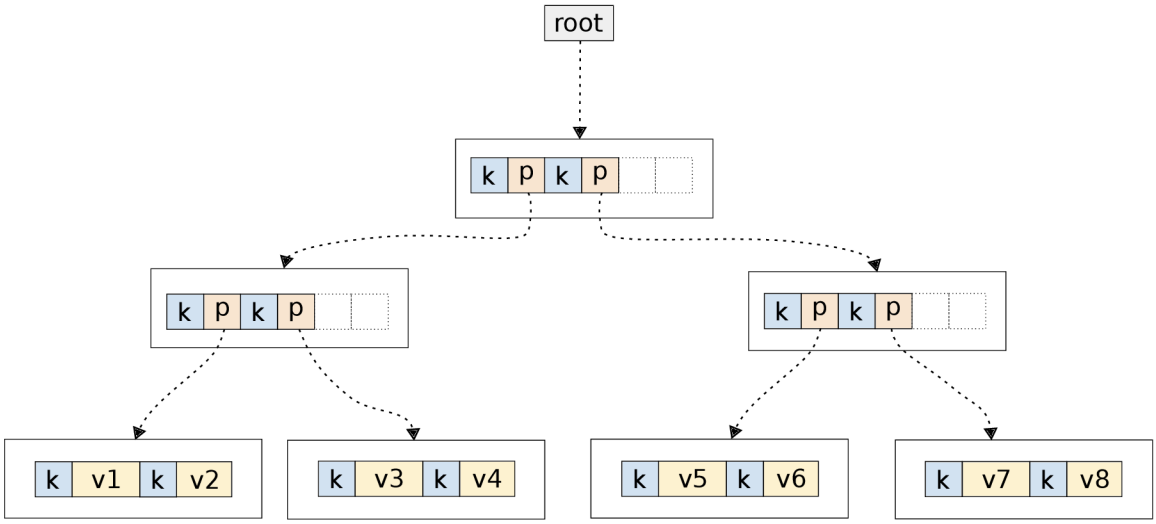

Figure 3.5: Internal layout of LMDB is a B+ tree. In the figure k stands for key, p for pointer and v for value. Scheme taken from webpage [40].

SQLite

SQLite [43] is self-contained, cross-platform database engine in a single file. It is fast, small and full-featured relational SQL database. SQLite supports all advanced concepts from database systems, such as transactions, indexes, materialised views and window functions. SQLite is a single file data storage, which does not need any external resources. Furthermore, it provides support for various data types, including BLOB. BLOB (Binary Large OBject) is a special data type capable of storing arbitrary binary data. According to webpage [32], SQLite has about 35 % faster writes and reads on small BLOBs, for example thumbnail images, than the file system using functions `fread()` and `fwrite()`.

Chapter 4

Benchmark methodology

This chapter will outline the methodology for benchmarking tabular and image data formats. Section 4.1 describes the environment in which the benchmarks were conducted. Section 4.2 lists all the metrics that were used to evaluate data formats. Section 4.3 presents a collection of benchmarks that were performed on different data formats. Section 4.4 presents the generated data and datasets that were used in the benchmarks.

4.1 Environment

Benchmarks were conducted on a server trafficgpu1 with 64 CPUs of Intel Xeon Silver 4314, 264 GB of RAM and 1,47 TB of M.2 NVMe SSD. The operating system is Ubuntu 22.04.4 LTS. Most data format files are created by calling `pandas` functions with default parameters without index. Avro is created by the `fastavro` package and Lance is created by the `pylance` package. JSON is stored in JSON Lines format, and the Parquet engine is `pyarrow`. Both table and fixed HDF5 formats are part of the benchmarks. Compression is turned off unless the benchmark specifies otherwise. Recommended version of Python for running benchmark is 3.10 or newer. All Python packages used in this benchmark with their versions are listed in Table 4.1.

4.2 Metrics

This benchmark uses five different metrics: save time, read time, save peak memory usage, read peak memory usage and total file size. Save and read time is measured by the Python module `timeit` in seconds. Save time measures the time required for the saving from `pandas.DataFrame` to the benchmark data format. Read time measures the opposite process, loading data from the disk to the `pandas.DataFrame`. Peak memory usage measures the maximum allocated resident memory while reading or writing data. Resident memory is allocated by a process in the RAM. This metric is measured by the Python module `resource` in MB or GB. Total file size metric is the final size of all created files on the disk. It is measured by the Python module `os` in MB or GB. Other metrics are derived from the above metrics.

Table 4.1: Overview of required Python packages and their versions used in benchmarks.

Package	Version
`asv`	0.6.3
`pandas`	2.2.1
`numpy`	1.26.4
`dask`	2024.4.1
`dask-expr`	1.0.10
`tables`	3.9.2
`lxml`	5.2.1
`openpyxl`	3.1.2
`pylance`	0.10.9
`fastavro`	1.9.4
`pillow`	10.3.0
`h5py`	3.10.0
`lmdb`	1.4.1
`datasets`	2.19.0
`matplotlib`	3.8.4

4.3 Benchmark suites

Benchmark suite described in this section should compare different data formats and image storages from Chapter 3. This benchmark focuses on the efficiency of tabular data storage, compression of data formats and performance of image storages. In order to accomplish this, three different benchmark suites were designed:

1. **Tabular** – Tabular benchmark will compare all data formats without compression on datasets with different data types. The results are given in Section 6.1.

2. **Compression** – Compression benchmark will compare data formats with enabled compression codes. The benchmark will consider two different compression codes, LZ4 and ZSTD. Only data formats that support internal compression with those two codes will be bechmarked. That includes HDF5 table, Parquet, Feather and ORC. The results are given in Section 6.2.

3. **Image** – Image benchmark will compare all image storages on two different image datasets, CIFAR-10 and ImageNet-100. Results are given in Section 6.3.

4.4 Datasets

The benchmark described in this chapter uses two types of datasets. One part of them is static and persistent datasets, which are loaded from memory into the `pandas.DataFrame`. The other part consists of dynamic datasets. They are generated within the benchmark application and only last until the end of the benchmark run. Tabular benchmarks will use synthetically generated data and the WebFace10M dataset. The parameters of tabular datasets are summarised in Table 4.2. The image storage benchmark will use the CIFAR-10 and ImageNet-100 datasets.

Table 4.2: Number of rows and data type distribution in tabular datasets.

Dataset name	Rows	Columns			
		Int64	Float64	Boolean	String
eq	1 M	2	2	2	2
int	1 M	5	1	1	1
float	1 M	1	5	1	1
bool	1 M	1	1	5	1
str	1 M	1	1	1	5
webface	10 M	6	21	0	6

Synthetic data

Synthetic data are generated before the start of the benchmarks and have limited lifespan. After the end of the benchmarks, they are destroyed, and a new one needs to be created. Actual values are different for each execution of the benchmarks, but they follow the defined schema. The schema defines the number of entries and the number of columns for each data type (`Int64`, `Float64`, `Boolean` and `String`). Values in all columns are generated with a uniform distribution. `Int64` and `Float64` have a value range from 0 to 100 and `String` length is fixed to 10 characters. These constants were taken from article [29], in this paper they extracted parameters from real datasets and about more than half of the values were below those constants. Synthetic data are generated by the data generator, which will be described in more detail in Chapter 5 about implementation.

This benchmark methodology defines five different synthetic datasets, four of them will be focused on different data types, and one will have a uniform distribution of all data types in columns. Datasets `int`, `float` `bool` and `str` are biased by data types. They have 5 columns of the main data type and 3 columns are equally distributed between other data types. The dataset `eq` with equal frequencies of data types is default and will be considered as a base. All generated datasets have 1 M rows and 8 columns.

WebFace10M

WebFace10M is a synthetic generated dataset that was provided by the Innovatrics company. It consists of only tabular data that annotate face images. The dataset schema that was used for data generation corresponds to their face datasets. It has 10 M rows and 33 columns, some examples of its column names are `age_iface`, `brightness_iface` or `contrast_iface`. It was generated as a benchmark model of real data that are used for face classification, identification and verification. The distribution of data types in the WebFace10M dataset is also shown in Table 4.2.

CIFAR-10

CIFAR-10 [15] is an image dataset that contains 60000 colour images, 50000 of them are training images and 10000 are test images. All images are small – they have a fixed resolution of 32×32 pixels. The images are divided into 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Each image is labelled with one of those classes. The label is a `uint8` ranging from 0 to 9. There exists also a variant with 100 classes called

CIFAR-100. CIFAR-10 and CIFAR-100 are labelled subsets of the 80 Million Tiny Images[1] dataset. Figure 4.1 shows a few example images from the CIFAR-10 dataset.

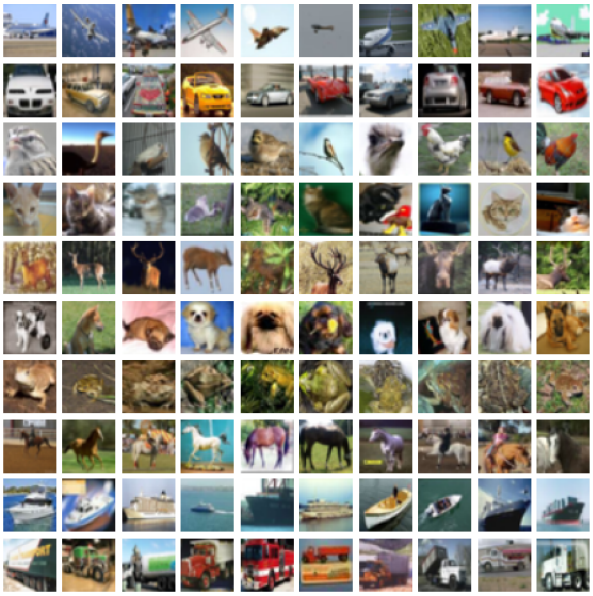

Figure 4.1: Example images from CIFAR-10 dataset.

ImageNet-100

ImageNet-100 [27] dataset is a subset of a larger dataset, ImageNet-1k [23], and is comprised of 100 classes that have been chosen randomly. Each image has also been resized to 160 pixels on the shorter side. In contrast to the CIFAR-10 dataset, ImageNet-100 has images with different shapes. In total, ImageNet-100 has 126689 training images and 5000 validation images. All images are labelled with an index of the class. A few examples of classes are bonnet, green mamba or langur. The label is a `uint8` in range from 0 to 99. The images are sorted according to their class index from the lowest to the highest. ImageNet-100 can be used in machine learning for the task of multiclass image classification. Figure 4.2 shows a few example images from the ImageNet-100 dataset.

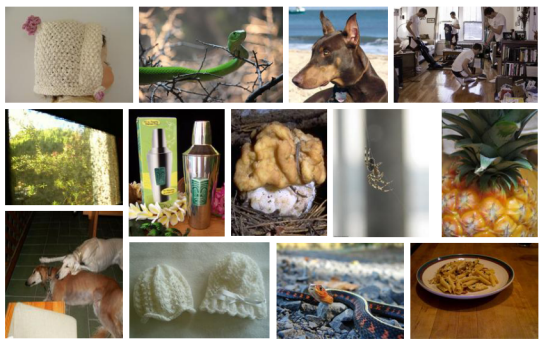

Figure 4.2: Example images from ImageNet-100 dataset.

[1] http://groups.csail.mit.edu/vision/TinyImages/

Chapter 5

Implementation of the benchmarks

This chapter will describe the implementation of the benchmarks designed in Chapter 4. I implemented two versions of the benchmarks. Both were programmed in the Python programming language using object-orientated programming. One uses the Airspeed Velocity framework, details in Section 5.2, and the other one was implemented from scratch, further description in Section 5.3. Although they are different, they share the same data generator module and classes for each data format and image storage. These modules are described in more detail in Section 5.1.

The high-level view of the benchmark implementation is shown in Figure 5.1. Firstly, the data generator creates the synthetic dataset, or static dataset is loaded from the disk to `pandas.DataFrame`. In case of image benchmarks, images are extracted from the dataset to the list of `numpy.ndarrays`, and the labels are loaded similarly into a list. Once the data are generated or loaded, they are stored to different data formats. Subsequently, the data are read back from the data format to the memory. Throughout the process of storing and reading the data, benchmark metrics are tracked. When the benchmarks are finished, results can be visualised in the form of graphs.

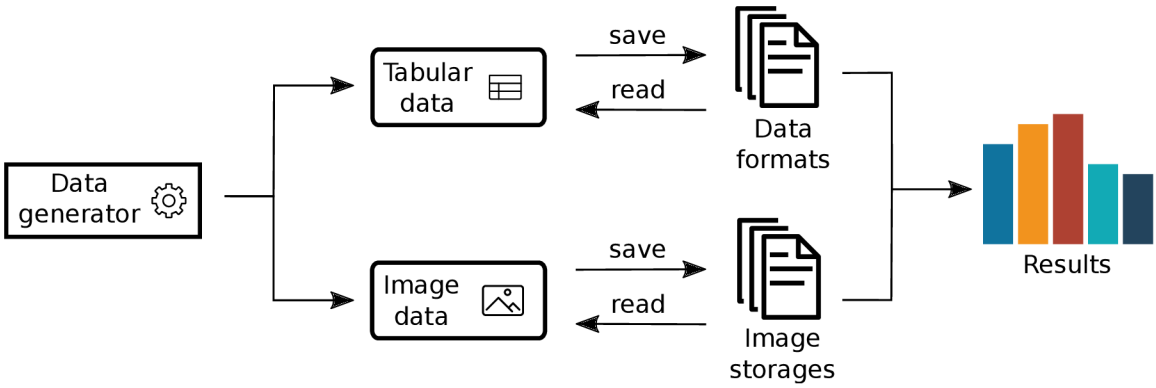

Figure 5.1: Overview of the benchmark implementation.

5.1 Core modules

Core modules are shared by both implementations and are necessary for them. They include a data generator and classes for each data format and image storage.

Data generator

Data generator module is stored in the folder `data_generator/`. Its purpose is to create synthetic parameterized datasets or load static ones. It consists of three classes: `GenDtype`, `Generator` and `Dataset`.

`GenDtype` class creates lists of random values of the same data type. Methods of this class are implemented with `numpy`, `random` and `string` packages. It is able to create a list of random integers, floats, booleans and strings with fixed or varying length, or a list of real English words. The list of words is created by random choices from the list of one hundred predefined words.[1]

`Dataset` is a class with no methods. It has only the attributes `df`, `images`, `labels` and `name`. It is a structure for holding benchmark data. This class also overrides method `__repr__()`, for correct visualization of the dataset name. All methods of `Generator` class create an instance of `Dataset` class.

`Generator` is the main class of the data generator, which creates and loads datasets. It has four methods. Method `gen_dataset()` generates synthetic, tabular dataset. It has multiple parameters that specify the schema of a created dataset, like the number of entries or the number of columns for each data type. Internally, this method calls methods of `GenDtype` to create columns of values with the same data type. When all the columns are generated, then they are shuffled, to create random order of the columns. Functions `zip()` and `dict()` are then used to create a dictionary of values. Lastly, this dictionary is passed to `pandas.DataFrame()`, which creates table representation from the dictionary. This dataframe is wrapped in the instance of `Dataset` and returned.

The remaining three methods are `load_(webface10M|cifar_10|imagenet_100)()`. All of them are responsible for loading static datasets. Each dataset needs a separate function because every dataset is stored differently. Webface10M is stored in a single HDF5 file, ImageNet-100 and CIFAR-10 are downloaded via Hugging Face. As was mentioned above, all methods return an instance of the `Dataset` class.

Data formats

Each data format that is a part of benchmark has a separate class. All of these classes inherit the implementation from the class `DataFormat`. This class defines attributes `filename` and `pathname`. Furthermore, it provides an interface for important abstract methods: `save()`, `read()`, `save_parallel()` and `read_parallel()`. Save and read methods are benchmarked in tabular benchmarks. These methods are abstract, and the concrete implementation depends on the data format. Most data formats are implemented via the calls of `pandas.to_*` or `pandas.read_*`. Implementation of Lance data format requires `pylance` package and Avro format requires `fastavro` package. Moreover, the `DataFormat` class implements four other methods that are supposed to return the size of the created file(s), or remove them. The implementation of these methods is encapsulated in a separate class `FileUtils`.

Image storages

Similarly as with data formats, every image storage is implemented in a separate class in the folder `image_storages/`. The implementation of image storage is more difficult than the storage of tabular data, because there does not exist one package that could

[1] Words taken from https://becomeawritertoday.com/list-of-random-words/.

store all images to different storages. Thus, specific packages are needed for each format. The implementation structure is also similar to the data formats, with one base class `ImageStorage`. `ImageStorage` is a class from which all other image storage classes inherit. This class declares abstract methods `save()` and `read()`. Those methods are benchmarked in image benchmarks. Furthermore, the `ImageStorage` class defines the `remove()` method to remove created files and the `size()` method to measure the size of images after storing them.

5.2 Airspeed Velocity

Airspeed Velocity (`asv`) [33] is a benchmark framework for benchmarking Python projects over their lifetime. It enables timing, raw timing, memory and peak memory benchmarks. Furthermore, custom computed generic values can be tracked. The results of the benchmarks can be easily published in the form of an interactive web frontend with graph grid. This website requires only a static webserver to host. Examples of real-world projects that use Airspeed Velocity are `numpy`, `scipy` or `astropy`.

The implementation of airspeed velocity benchmarks consists of multiple files and directories. The root folder requires the `asv.conf.json` file. It is a configuration file for `asv`, with information on the required Python packages, the version of Python that runs the benchmarks and the type of virtual environment. In addition to this file, the `setup.py` file must also be included in the root directory. This file is necessary because `asv` expects that the package, which is benchmarked, is installable by `pip`.

Other files of `asv` implementations are located in directories `benchmarks/` and `.asv/`. The `benchmark/` folder contains all the implementation of the benchmarks. The hidden `.asv/` directory is divided into three subdirectories. Folder `env/` stores virtual environment for running benchmarks, folder `html/` stores generated interactive website and folder `results/` stores results of the benchmark runs.

Airspeed Velocity supports a wide variety of benchmark types. This benchmark will use the timing benchmark, the peak memory benchmark and the tracking of generic values for the total size metric. Timing benchmarks are all methods that have a name with the prefix `time`. Peak memory benchmarks are all methods that have a name with the prefix `peakmem`. For better reproducibility, `asv` runs every benchmark in a virtual environment with its own process. Algorithm 1 shows how `asv` runs timing benchmarks. Algorithm taken from `asv` documentation [33].

Algorithm 1: ASV TIMING BENCHMARK

```
1: for round in range(‘rounds‘):
2:    for benchmark in benchmarks:
3:       with new process:
4:          <calibrate ‘number‘ if not manually set>
5:          for j in range(‘repeat‘):
6:             <setup ‘benchmark‘>
7:             sample = timing_function(<run benchmark ‘number‘
                times>) / ‘number‘
8:             <teardown ‘benchmark‘>
```

The results of the `asv` benchmarks can be displayed in the form of an interactive website. An example of such a website is in Figure 5.2. The website comprises of multiple pages. Home page presents all the benchmarks from the initial run to the last run in a graph grid. After clicking on a graph, new page with the selected graph appears, where the user can modify the parameters of the graph. Another page of the website is Benchmark list, which shows all the measured values in the latest run and compares them with the previous run. The last page of the website shows the regression.

Running `asv` benchmarks is simple and can be done with only a few commands, which are listed below. The command `asv run` has optional parameters. If the benchmarks are slow, `--quick` runs every benchmark only once, or `--bench` can run specific benchmark suites only.

```
$ asv run [--quick] [--bench benchmark_suite]
$ python3 postprocessing.py
$ asv publish
$ asv preview
```

5.3 Custom benchmark

Custom benchmark was implemented without any framework. A major advantage over `asv` benchmarks is in its faster and more effective algorithm for running benchmarks. Algorithm 2 shows how the benchmark runner works. Runner in `asv` runs every benchmark in a new environment. Process of recreating the environment and removing it takes a lot of time. Furthermore, it deletes saved benchmark data and that means they need to be stored again for new benchmark. To overcome this issue, custom benchmark uses a chain of benchmarks and one run of benchmark collects all defined metrics.

Algorithm 2: Custom Benchmark runner

```
1:  all_results = []
2:  for format in ‘formats‘:
3:      results = dict()
4:      with new process p_save:
5:          results[save_time] = timing_function(<format save dataset>)
6:          results[save_peakmem] = peakmem_function(p_save)
7:      results[total_size] = size_function()
8:      with new process p_read:
9:          results[read_time] = timing_function(<format read dataset>)
10:         results[read_peakmem] = peakmem_function(p_read)
11:     <remove created files>
12:     all_results.append(‘results‘)
13: return all_results
```

For every data format, the algorithm first creates a new dictionary for results. Afterwards, in a new process, it saves the benchmark data and measures the statistics. The total file size is then retrieved. Because benchmarks continue in a chain, a new process reads the data from the disk and measures the metrics. Finally, created files are removed and results are added to a list. This algorithm is more efficient in the number of disk operations.

This implementation can run only one benchmark suite in one run. In order to execute benchmarks, run the script `main.py` with the option `--tabular`, `--compression` or `--image` on. Results of the run are printed to the terminal and saved to a CSV file. To make results more visual, the user might run benchmarks with the option `--report` turned on. This creates a report from the benchmark results. Example of such a report is in Figure 5.3. A report consists of five bar graphs, each one visualizing specific metric. Green bars are for values that are above average and red bars are for values that are below average.

```
$ python3 main.py (--tabular|--compression|--image)
            [--webface <path>] [--report]
```

5.4 Continuous integration

In order to automate the process of running the benchmarks, continuous integration was used. This enables the user to run the benchmarks and presents the results on a regular basis. For the Airspeed Velocity benchmarks, the script `cron.sh` was written. This script first pulls the current versions of the benchmarks from the `git`. Then it runs every benchmark suite. After the benchmarks are finished, results are published to the GitHub webpage https://mariantaragel.github.io/asv-format-bench/. This process is done by the command `asv gh-pages`. It creates a websites and pushes it to the `gh-pages` branch. Then it start a GitHub Action pipeline that builds the page and deploys it.

To make this process fully automatic, a cron job can be employed on a server. Cron jobs can be set to run a specific script repeatedly. To run the script `cron.sh` on the first day of every month at 1:00, the below stated line can be added to a cron job list.

```
0 1 1 * * /home/xtarag01/asv_format_bench/cron.sh
```

Figure 5.2: Example of a website with Airspeed Velocity benchmark results. Top figure shows a graph with the results of a single benchmark. Bottom figure shows the main graph grind page. Figures are only illustrative.

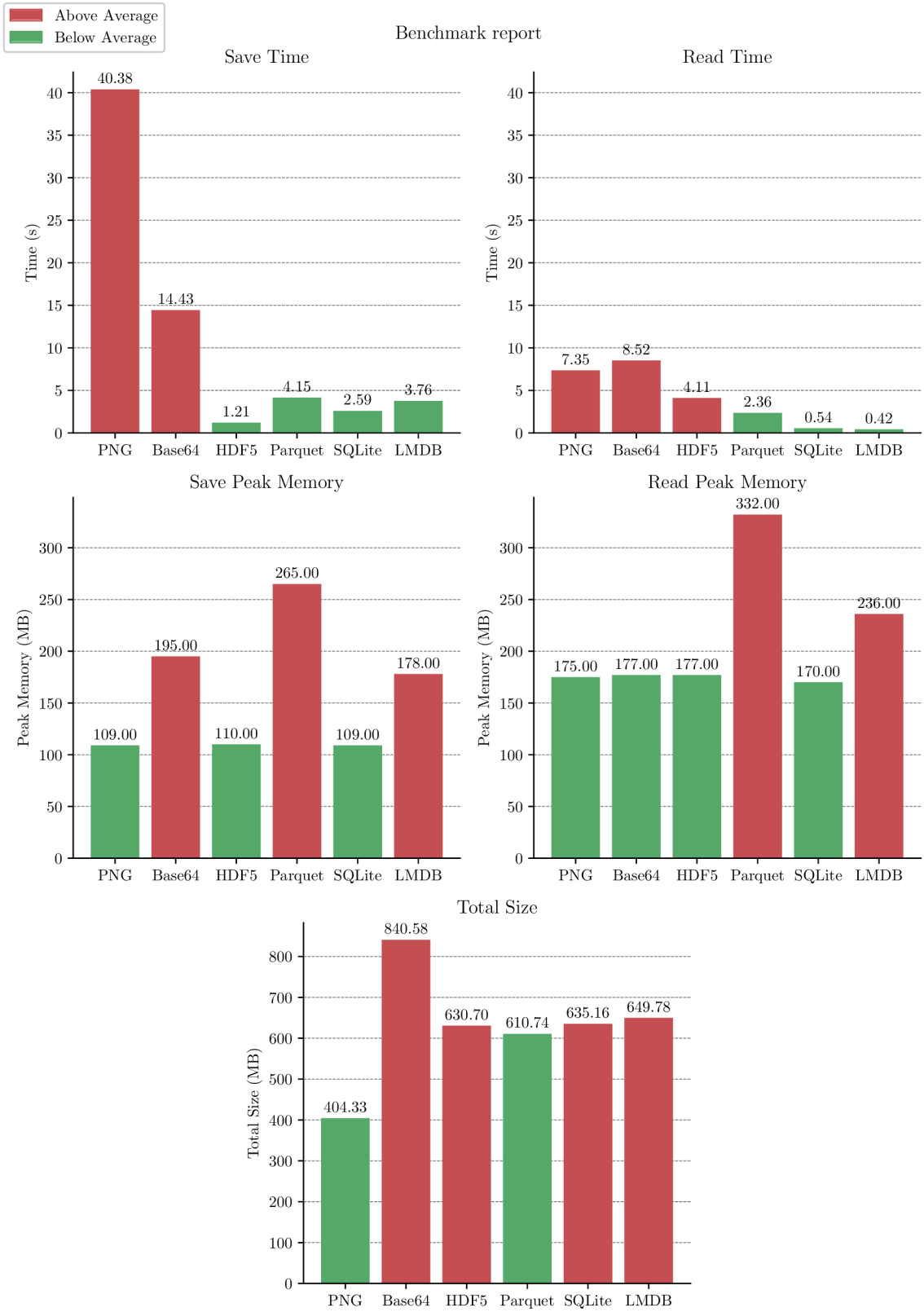

Figure 5.3: Example of a report with results from the custom benchmark implementation. Values in the graphs are only illustrative.

Chapter 6

Results evaluation

This chapter will analyse the results of the benchmarks that were designed in Chapter 4 and implemented in Chapter 5. The purpose of this chapter is to evaluate which data format or image storage is the best based on some metric. In each section, results related to a single benchmark suite will be analysed. All the findings are summarized in Table 6.1 and Table 6.2.

Table 6.1: Summary of key findings from tabular benchmarks.

Metric	Format	Key advantage
Save time	Feather	Arrow columnar memory specification
Read time	Pickle	Optimized on deserialisation of Python objects
File size	Parquet	Wide variety of encodings
Save peak memory	CSV	Storing only text strings
Read peak memory	Pickle	Optimized on deserialisation of Python objects
Compression ratio	HDF5	Effective metadata compression
Compression speed	Parquet	PAX layout

Table 6.2: Summary of key findings from image benchmarks.

Metric	Format	Key advantage
Save time	SQLite	Lightweight save process and BLOBs
Read time	LMDB	Single-Level Store
File size	PNG	DEFLATE compression

6.1 Tabular benchmarks

This section will describe the results of tabular benchmarks without compression. The metrics that will be used are save time, read time, file size, save peak memory and read peak memory.

Save time. The first metric based on which data formats will be evaluated is saving time, which means the time required to save `pandas.Dataframe` to a disk. The results are given in Table 6.3. According to the benchmarks, the fastest data format to save

Table 6.3: Experimental results of required saving time on different datasets.

Format	Save time					
	`eq` [s]	`int` [s]	`float` [s]	`bool` [s]	`str` [s]	`webface` [s]
CSV	2,65	1,84	4,85	1,91	2,30	250,33
JSON	1,88	1,39	1,80	1,36	1,75	111,43
XML	16,64	15,85	17,23	15,57	16,85	—
H5 fixed	1,78	1,48	1,49	1,45	2,59	—
H5 table	1,92	1,58	1,58	1,69	2,73	51,30
Parquet	0,39	0,24	0,33	0,28	0,71	17,72
Feather	0,24	0,10	0,13	0,11	0,45	10,96
ORC	0,43	0,33	0,27	0,34	0,74	16,58
Pickle	0,47	0,31	0,23	0,31	1,11	21,22
Excel	81,78	78,06	81,99	77,63	88,37	—
Lance	1,41	1,00	1,18	0,90	1,85	105,09
Avro	4,00	3,36	3,64	3,76	5,62	138,39

tabular data is Feather. It performs equally well on all data formats, regardless of the data type distribution. On the Webface10M dataset, it outscored other formats with only 10 s required for saving. Its main advantage against other formats is the Arrow Columnar memory specification, which is particularly effective for I/O operations. Other formats that have fast save times are Parquet and ORC. What is a bit surprising is the performance of Pickle, which is 2× slower on the Webface10M dataset than Feather. On the other side of the spectrum are XML and Excel. JSON stands out as the fastest text format for saving tabular data. It is faster than the CSV format and its performance is stable on all data types.

Table 6.4: Experimental results of required reading time on different datasets.

Format	Read time					
	`eq` [s]	`int` [s]	`float` [s]	`bool` [s]	`str` [s]	`webface` [s]
CSV	1,26	0,86	1,13	0,91	1,84	65,84
JSON	2,68	2,65	2,70	2,35	2,97	192,34
XML	23,70	24,10	24,20	23,13	23,52	—
H5 fixed	1,60	1,45	1,47	1,37	2,15	—
H5 table	2,03	1,65	1,66	1,76	3,11	50,72
Parquet	0,64	0,28	0,33	0,28	1,31	15,23
Feather	0,42	0,24	0,24	0,27	0,87	11,13
ORC	0,58	0,29	0,30	0,35	1,42	15,37
Pickle	0,20	0,13	0,12	0,16	0,46	7,17
Excel	62,11	52,94	55,35	52,36	82,38	—
Lance	0,62	0,36	0,31	0,29	1,35	25,22
Avro	2,99	2,85	2,65	2,45	4,03	127,87

Read time. The next metric that is going to be analysed is the opposite process, read time. Read time is the time required for the back-reconstruction from disk to a table rep-

resentation. The results are in Table 6.4. Based on this metric, the fastest deserialisation format is Pickle. On every dataset it needed the lowest amount of time. On the large Webface10M dataset, it required only 7 *s* to load the entire dataset, which is twice as fast as Parquet. Its key advantage over other formats is its tight integration with the Python environment. Pickle is optimised for fast (de)serialisation of Python objects. Other honorable mentions based on the read time metric are Feather and Parquet. Interestingly, good results were also measured on the Lance data format. As with the save time metrics, the worst data formats are XML and Excel. This time, the fastest text data format for reading is CSV. Compared to JSON, it has a simpler syntax, which can be deserialised faster.

Total size. One of the most important metrics is the total file size on the disk. The results of this metric are given in Table 6.5. Based on the total file size metric, the best data formats are Parquet and ORC. ORC is especially memory-efficient for smaller datasets. For larger datasets, the winner is Parquet. On the Webface10M dataset, the difference between these two formats is significant. Parquet requires only 4,6 GB, while ORC needs 6,9 GB. Both data formats are memory efficient because they employ the modern PAX format and use a wide variety of encoding schemes. Another data format worth mentioning is Avro. On same datasets, its results are even better than Parquet. What is also clearly visible from the results is that text data formats require a lot more space than binary formats. Among the text data formats, CSV is the best in total file size, because it has a simpler syntax than JSON or XML.

Table 6.5: Experimental results of required space on different datasets.

Format	**Total size**					
	`eq` [MB]	`int` [MB]	`float` [MB]	`bool` [MB]	`str` [MB]	`webface` [GB]
CSV	75	49	110	60	82	9,0
JSON	128	105	148	115	145	12,7
XML	228	202	263	213	235	—
H5 fixed	69	71	71	43	91	—
H5 table	62	67	67	39	75	24,4
Parquet	47	27	57	24	80	4,6
Feather	60	62	62	31	86	6,9
ORC	38	23	51	20	59	6,3
Pickle	60	62	62	34	82	6,9
Excel	66	58	95	52	51	—
Lance	69	66	66	35	107	7,2
Avro	43	27	53	25	65	6,5

Save peak memory. Besides the total file size, the memory that was allocated in the RAM through the data saving process is also an important metric. The results of this experiment are shown in Table 6.6. In the benchmark, the least allocated memory is required by CSV. CSV is storing only simple strings, which means that almost no other memory is needed. From the binary formats, the best data format is Pickle. Pickle can effectively store numerical values with minimal allocated memory. On strings, its performance is considerably worse, with 199 MB necessary for storing the `str` dataset. On the Webface10M dataset, Pickle needed only 3,6 GB allocated in RAM. Parquet, Feather and ORC require similar

Table 6.6: Experimental results of peak allocated memory during the saving process.

Format	Save peak memory					
	`eq` [MB]	`int` [MB]	`float` [MB]	`bool` [MB]	`str` [MB]	`webface` [GB]
CSV	5	4	8	5	5	0,01
JSON	630	515	728	566	712	62,1
XML	3196	3123	3358	3210	3202	—
H5 fixed	82	16	16	66	248	—
H5 table	107	61	61	61	299	29,7
Parquet	79	72	71	70	129	6,8
Feather	65	38	38	39	116	6,9
ORC	66	38	38	39	112	6,9
Pickle	68	2	2	35	199	3,6
Excel	2707	2677	2802	2739	2677	—
Lance	264	220	269	174	395	13,1
Avro	429	399	524	398	399	18,7

save peak memory allocated. As in the previous metric, the worst data formats are XML and Excel.

Read peak memory. During the reading process, the peak memory metric was also evaluated. The exact results are in Table 6.7. In this metric, all the measured values are higher compared to save peak allocated memory. From text formats, CSV needs the least additional memory while reading the data. Its performance is not as impressive as in the save peak memory metric, but its results are still solid. From binary formats, the best according to experiments is Pickle. From other formats, Parquet achieved satisfactory results. On the opposite side are JSON and XML. For Webface10M JSON requires 97,5 GB, and for the `eq` dataset XML requires 5,2 GB of extra memory allocated in RAM.

Table 6.7: Experimental results of peak allocated memory during the reading process.

Format	Read peak memory					
	`eq` [MB]	`int` [MB]	`float` [MB]	`bool` [MB]	`str` [MB]	`webface` [GB]
CSV	175	144	159	80	404	14,8
JSON	1564	1457	1708	1436	1841	97,5
XML	5212	5203	5376	5267	5210	—
H5 fixed	277	180	173	136	784	—
H5 table	314	213	212	173	738	45,2
Parquet	524	362	341	274	1090	20,2
Feather	243	196	179	148	452	11,3
ORC	462	307	306	245	996	13,5
Pickle	125	64	63	63	420	10,4
Excel	486	440	572	435	676	—
Lance	474	312	310	250	1016	13,8
Avro	619	602	704	529	830	37,6

6.2 Compression benchmarks

This section is dedicated to the analysis of the benchmark results when the compression codec was enabled. The benchmarks were executed twice, first with LZ4 compression and next with ZSTD compression at level 1. Data formats that allow both of these compressions are HDF5 table, Parquet, Feather, and ORC. The metrics that were measured are compression ratio, save time delta and read time delta. Dataset Webface10M was used in this benchmark. The results of this benchmark are shown in Table 6.8 for LZ4 and in Table 6.9 for ZSTD.

Table 6.8: Experimental results of enabling compression codec LZ4.

Format	**Save time** $[\Delta s]$	**Read time** $[\Delta s]$	**Compression ratio**
H5 table	+0,51	+0,12	0,17
Parquet	+8,81	−0,73	0,50
Feather	+19,03	+2,41	0,57
ORC	+3,00	−0,46	1,00

Table 6.9: Experimental results of enabling compression codec ZSTD.

Format	**Save time** $[\Delta s]$	**Read time** $[\Delta s]$	**Compression ratio**
H5 table	+18,27	+7,56	0,13
Parquet	+10,36	−0,77	0,21
Feather	+21,89	+6,36	0,30
ORC	+14,39	+3,83	0,41

Compression ratio. The first metric that was extracted from the compression benchmark results is the compression ratio. Compression ratio is the ratio of the file size with compression over the file size without compression. The lower the number, the better. If the number is equal to 1, then the file size with compression is the same as without compression. When the compression ratio is greater than 1, the file size increases with compression. For both compression codes, the best ratio was achieved by the HDF5 table format. The cause of this is that the original uncompressed size is large, 24,4 GB, with a lot of redundant metadata. Those can be effectively removed by compression and the file size afterwards is approximately 3 GB. Other data formats also reach acceptable compression ratios. After ZSTD compression, Parquet needs only 960 MB to save the Webface10M dataset with 10 million rows. An interesting behaviour was observed in the ORC compression benchmark results. Its compression ratio for LZ4 codec was equal to 1, which means the file was neither compressed nor extended. This is because ORC disables compression when it detects LZ4, since it would cause the file size to increase. This behaviour was also noticed in article [18].

Time of compression. The time of compression metric is the time added to save time without compression. It was calculated as the difference between save time with compression and save time without compression. From the results it is clear that all the save times were extended. For LZ4 compression, the lowest overhead was observed in the HDF5 table

format save time. For the ZSTD codec, Parquet needs the least additional time for compression. The results of the Feather format are interesting. Without compression, Feather exhibits extremely fast save and read times. After applying compression, its I/O performance plummets. Compression LZ4 added +19,03 *s* and ZSTD added +21,89 *s* to the original save time.

Time of decompression. The last metric measured in the compression benchmark was the time of decompression. It was measured as the difference between read time with compression and read time without compression. The results of this metric show that decompression does not add too much time compared to the original read time. Parquet even decreases its read time when compression is applied, -0,73 *s* for LZ4 and -0,77 *s* for ZSTD. This comes mainly from the fact that the file size is low, and the required number of accesses to the disk is therefore minimal.

6.3 Image benchmarks

This section will analyse the results of image storage benchmarks. The metrics that will be considered are save time, read time and total size. The results are included in Table 6.10 for the CIFAR-10 dataset and in Table 6.11 for the ImageNet-100 dataset.

Table 6.10: Experimental results of image storages on the CIFAR-10 dataset.

Format	**Save time** [*s*]	**Read time** [*s*]	**Total size** [MB]
PNG	10,54	5,72	113
Base64	2,63	1,55	205
HDF5	2,75	22,04	157
Parquet	0,43	1,45	154
SQLite	0,47	0,30	205
LMDB	0,75	0,32	207

Table 6.11: Experimental results of image storages on the ImageNet-100 dataset.

Format	**Save time** [*s*]	**Read time** [*s*]	**Total size** [GB]
PNG	1076,31	164,61	8,0
Base64	216,00	117,40	17,7
HDF5	24,38	76,81	13,3
Parquet	44,67	28,45	12,5
SQLite	22,12	12,16	13,4
LMDB	30,38	11,87	13,7

Save time. The first metric that will be studied is the required time to save an image dataset. In this, the results were comparable. On the smaller CIFAR-10 dataset, the winner is Parquet, but on the larger ImageNet-100 dataset, results show that the fastest was SQLite. SQLite's performance on CIFAR-10 was almost as good as Parquet's, with only 0,04 *s* difference. SQLite fast save and read operations are also benchmarked on SQLite's

webpage [32]. The slowest option is to save every image to a single PNG file. The main reason why storing images to PNG is slow is because PNG uses DEFLATE compression, which requires a lot of time and the whole process is complicated. Encoding to Base64 is also slow and not recommended for fast saving time. The results of HDF5 and LMDB are reasonable.

Read time. The second metric based on which image storages will be compared is read time. It is the opposite process of loading images from a disk to the list of `numpy.ndarray`s. In this metric, the first place was also tight. The fastest loading on the CIFAR-10 dataset was measured on SQLite, but for ImageNet-100 the fastest was LMDB. LMDB employs the Single-Level Store concept which enables extremely fast loading of images, as stated by Howard Chu [8]. On the other side are PNG, Base64 and HDF5. All of them exhibit slow reading of images.

Total size. The last metric that will be analysed from the image storage benchmark results is the total size. With regard to this metric, the best image storage option is to save every image to a single PNG file with labels in a CSV file. As was mentioned above, PNG is using compression, while other tested formats are not using any specific compression or encoding. Other formats, except Base64 encoding, are getting approximately the same results. If we take average results of HDF5, Parquet, SQLite and LMDB, and compare them with PNG, then PNG brings about 40 % size reduction, but the required save time is prolonged about 34 times and the read time is extended about 5 times.

Chapter 7

Conclusion

In conclusion, the main aim of this thesis was to explore the performance characteristics of data formats. In addition to that, the work has also evaluated different options for image storages. The main outcome is the designed set of benchmarks and their results. The benchmarks were divided into more benchmark suites with defined metrics. Every benchmark should suggest the best data format based on some metric.

The key findings of the benchmark results are summarised in Table 6.1 for tabular formats and Table 6.2 for image storages. The results of tabular data formats show that data format with the fastest save and read times is Feather. Parquet stands out as the most memory-efficient data format. In image storage benchmarks, the results indicate that the fastest image storages are SQLite and LMDB. The least memory space is required when every image is stored in a single PNG file with labels in a CSV file.

This study can contribute to a better understanding of how different formats behave and what the most important features and principles of modern data formats are. This can help future data format engineers create new data formats that will match the needs of large machine-learning datasets. Furthermore, benchmark results can help data scientists choose the right data format for their data. If they have specific criteria for data formats, such as fast save time or high memory efficiency, the benchmark can recommend the best option.

Future study in the research area of benchmarking data formats and image storages offers several opportunities. This benchmark could be extended to new data formats. Interesting new data formats that were not benchmarked are Protocol Buffers, MessagePack or Ron. What is more, new benchmark suites could be designed. One option is to add a benchmark of query speed execution, that is, search in loaded data. Another suite that could be interesting is the benchmark of save and read time from/to different types of disks, for example HDD, SSD, NVMe or RAMdisk.

All in all, I hope this Bachelor's thesis can contribute to the area of benchmarking data formats and image storages.

Bibliography

[1] Abadi, D. J.; Madden, S. R. and Ferreira, M. C. Integrating Compression and Execution in Column-Oriented Database Systems. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2006, p. 671–682. ISBN 1595934340. Available at: https://doi.org/10.1145/1142473.1142548.

[2] Ailamaki, A.; DeWitt, D. J.; Hill, M. D. and Skounakis, M. Weaving Relations for Cache Performance. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 2001, p. 169–180. ISBN 1558608044. Available at: https://dl.acm.org/doi/10.5555/645927.672367.

[3] Apple, J. Split block Bloom filters. *ArXiv.org*. 5th ed. Cornell University Library, 2021. ISSN 2331-8422. Available at: https://doi.org/10.48550/arXiv.2101.01719.

[4] Bloom, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*. Association for Computing Machinery, 1970, vol. 13, no. 7, p. 422–426. ISSN 0001-0782. Available at: https://doi.org/10.1145/362686.362692.

[5] Boutell, T. *PNG (Portable Network Graphics) Specification Version 1.0* RFC 2083. March 1997. Available at: https://datatracker.ietf.org/doc/html/rfc2083. [cit. 2024-03-26].

[6] Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; Maler, E. and Yergeau, F. *Extensible Markup Language (XML) 1.0* online. 5th ed. 26. november 2008. Available at: https://www.w3.org/TR/xml/. [cit. 2024-03-17].

[7] Byna, S.; Breitenfeld, M. S.; Dong, B.; Koziol, Q.; Pourmal, E. et al. ExaHDF5: Delivering Efficient Parallel I/O on Exascale Computing Systems. *Journal of Computer Science and Technology*. Springer, 2020, vol. 35, p. 145–160. ISSN 1000-9000. Available at: https://doi.org/10.1007/s11390-020-9822-9.

[8] Chu, H. MDB: A Memory-Mapped Database and Backend for OpenLDAP. In: *Proceedings of the 3rd International Conference on LDAP*. 2011. Available at: https://www.openldap.org/pub/hyc/mdb-paper.pdf.

[9] Copeland, G. P. and Khoshafian, S. N. A decomposition storage model. *SIGMOD Record*. Association for Computing Machinery, 1985, vol. 14, no. 4, p. 268–279. ISSN 0163-5808. Available at: https://doi.org/10.1145/971699.318923.

[10] Eppstein, D. and Goodrich, M. T. *Streaming Algorithms for Straggler Detection* online. Irvine: University of California, Computer Science Department, 2007,

2010-03-13. Available at: https://ics.uci.edu/~eppstein/pubs/EppGoo-WADS-07.pdf. [cit. 2024-04-07].

[11] Goldstein, J.; Ramakrishnan, R. and Shaft, U. Compressing relations and indexes. In: *Proceedings 14th International Conference on Data Engineering*. IEEE Computer Society, 1998, p. 370–379. ISBN 0-8186-8289-2. Available at: https://doi.org/10.1109/ICDE.1998.655800.

[12] Huai, Y.; Chauhan, A.; Gates, A.; Hagleitner, G.; Hanson, E. N. et al. Major Technical Advancements in Apache Hive. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2014, p. 1235–1246. ISBN 9781450323765. Available at: https://doi.org/10.1145/2588555.2595630.

[13] Huffman, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*. IEEE, 1952, vol. 40, no. 9, p. 1098–1101. ISSN 0096-8390. Available at: https://doi.org/10.1109/JRPROC.1952.273898.

[14] Josefsson, S. *The Base16, Base32, and Base64 Data Encodings* RFC 4648. October 2006. Available at: https://datatracker.ietf.org/doc/html/rfc4648. [cit. 2024-03-29].

[15] Krizhevsky, A. *Learning Multiple Layers of Features from Tiny Images*. 2009. Master's thesis. University of Toronto, Department of Computer Science. Available at: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[16] Lemire, D. and Boytsov, L. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*. John Wiley & Sons, Inc., 2015, vol. 45, no. 1, p. 1–29. ISSN 0038-0644. Available at: https://doi.org/10.1002/spe.2203.

[17] Lim, S.; Young, S. R. and Patton, R. An analysis of image storage systems for scalable training of deep neural networks. *System*, 2016, vol. 5, no. 7, p. 11. Available at: http://tiny.cc/k4mkxz.

[18] Liu, C.; Pavlenko, A.; Interlandi, M. and Haynes, B. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proceedings of the VLDB Endowment*. VLDB Endowment, 2023, vol. 16, no. 11, p. 3044–3056. ISSN 2150-8097. Available at: https://doi.org/10.14778/3611479.3611507.

[19] McKinney, W. *Feather: fast, interoperable data frame storage* online. 2024. Available at: https://github.com/wesm/feather. [cit. 2024-03-21].

[20] Melnik, S.; Gubarev, A.; Long, J. J.; Romer, G.; Shivakumar, S. et al. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment*. VLDB Endowment, 2010, vol. 3, 1-2, p. 330–339. ISSN 2150-8097. Available at: https://doi.org/10.14778/1920841.1920886.

[21] Paradarami, T. *High-performance genetic datastore on AWS S3 using Parquet and Arrow* online. 8. february 2021. Available at: https://medium.com/23andme-engineering/genetic-datastore-4b213256db31. [cit. 2024-03-10].

[22] Pavlo, A. *Data Formats & Encoding I* online. Carnegie Mellon University, 2024. Available at: https://15721.courses.cs.cmu.edu/spring2024/slides/02-data1.pdf. [cit. 2024-05-02].

[23] Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S. et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*. Springer, 2015, vol. 115, no. 3, p. 211–252. ISSN 0920-5691. Available at: https://doi.org/10.1007/s11263-015-0816-y.

[24] Shafranovich, Y. *Common Format and MIME Type for Comma-Separated Values (CSV) Files* RFC 4180. October 2005. Available at: https://datatracker.ietf.org/doc/html/rfc4180. [cit. 2024-03-16].

[25] Shivashetty, V. and Rajput, G. A Survey on Different Types of Image Formats and Compression Techniques. *International Journal of Science and Research*, 2014, vol. 3, no. 6, p. 798–802. ISSN 2319-7064. Available at: https://www.ijsr.net/archive/v3i6/MDIwMTQyNA==.pdf.

[26] Shokrzad, R. *Pickle, JSON, or Parquet: Unraveling the Best Data Format for Speedy ML Solutions* online. 15. november 2023. Available at: https://medium.com/@reza.shokrzad/pickle-json-or-parquet-unraveling-the-best-data-format-for-speedy-ml-solutions-10c3f7bf4d0c. [cit. 2024-03-23].

[27] Tian, Y.; Krishnan, D. and Isola, P. Contrastive Multiview Coding. In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, Proceedings, Part XI*. Springer, 2020, p. 776–794. ISBN 978-3-030-58620-1. Available at: https://doi.org/10.1007/978-3-030-58621-8_45.

[28] Xie, B.; Tang, H.; Byna, S.; Hanley, J.; Koziol, Q. et al. Battle of the Defaults: Extracting Performance Characteristics of HDF5 under Production Load. In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE Computer Society, 2021, p. 51–60. ISBN 978-1-7281-9586-5. Available at: https://doi.org/10.1109/CCGrid51090.2021.00015.

[29] Zeng, X.; Hui, Y.; Shen, J.; Pavlo, A.; McKinney, W. et al. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment*. VLDB Endowment, 2023, vol. 17, no. 2, p. 148–161. ISSN 2150-8097. Available at: https://doi.org/10.14778/3626292.3626298.

[30] *ECMA-262: ECMAScript Language Specification*. 3rd ed. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), december 1999. Available at: https://ecma-international.org/publications-and-standards/standards/ecma-262/.

[31] *ECMA-376: Office Open XML File Formats*. 5th ed. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), december 2021. Available at: https://ecma-international.org/publications-and-standards/standards/ecma-376/.

[32] *35% Faster Than The Filesystem* online. 5. december 2023. Available at: https://www.sqlite.org/fasterthanfs.html. [cit. 2024-04-02].

[33] *Airspeed velocity* online. 2024. Available at: https://asv.readthedocs.io/en/stable/index.html. [cit. 2024-04-10].

[34] *Apache Avro™ – a data serialization system* online. December 2023. Available at: https://avro.apache.org/. [cit. 2024-04-03].

[35] *Apache Parquet* online. 2024. Available at: https://parquet.apache.org/. [cit. 2024-03-21].

[36] *Feather File Format* online. 2024. Available at: https://arrow.apache.org/docs/python/feather.html. [cit. 2024-03-21].

[37] *HDF5 File Format Specification Version 3.0* online. 2024. Available at: https://docs.hdfgroup.org/hdf5/develop/_f_m_t3.html. [cit. 2024-03-21].

[38] *Introducing JSON* online. 2023. Available at: https://www.json.org/json-en.html. [cit. 2024-03-16].

[39] *Lance: modern columnar data format for ML* online. April 2024. Available at: https://lancedb.github.io/lance/. [cit. 2024-04-02].

[40] *LMDB freelist* online. 2024. Available at: https://github.com/ledgerwatch/erigon/wiki/LMDB-freelist. [cit. 2024-04-09].

[41] *[MS-XLSX]: Excel (.xlsx) Extensions to the Office Open XML SpreadsheetML File Format* online. 2023. Available at: https://learn.microsoft.com/en-us/openspecs/office_standards/ms-xlsx/. [cit. 2024-03-24].

[42] *Pickle — Python object serialization* online. 2024. Available at: https://docs.python.org/3/library/pickle.html. [cit. 2024-03-23].

[43] *SQLite* online. 9. march 2024. Available at: https://www.sqlite.org/index.html. [cit. 2024-04-02].

[44] *The HDF5® Library & File Format* online. 2023. Available at: https://www.hdfgroup.org/solutions/hdf5/. [cit. 2024-03-21].

[45] *The smallest, fastest columnar storage for Hadoop workloads* online. 2024. Available at: https://orc.apache.org/. [cit. 2024-03-22].

Appendix A

Poster

Last addition to this work is a poster, which is depicted in Figure A.1.

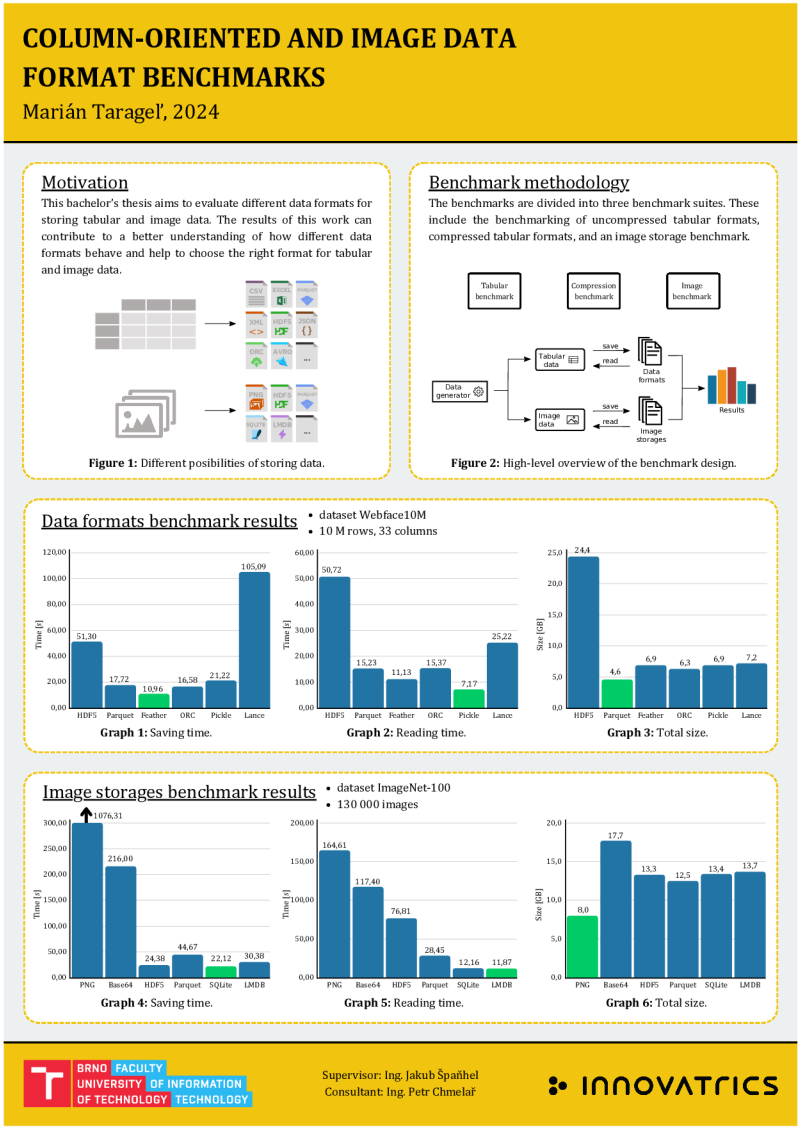

Figure A.1: Poster presenting this bachelor's thesis, its goals and results.