Yosemite: Towards Designing a File Format for Long-Term Archival Storage of Structured Datasets

Jayjeet Chakraborty
University of California, Santa Cruz
Santa Cruz, CA, USA
jayjeetc@ucsc.edu

Abstract

Living in the 21st century, we humans collect a huge amount of data from different data sources on a regular basis. It is important to store some of that data reliably for extended periods of time so that our future generations can have access to the knowledge developed by the previous generations and can build on top of that. Much of the data generated or captured are structured, meaning they adhere to a specific schema or a set of rules. Although traditionally, people stored structured data in relational database management systems (RDBMS), with the advent of big data storage technologies such as data lakes and lakehouses, several structured data file formats have been developed to store data cheaply in the cloud. Most of these big data file formats were designed with storage and data access efficiency in mind and not particularly for archival storage. In this paper, we study the existing structured data file formats and analyze them based on their ability to store data for long periods of time. We then propose some features that we feel are essential for an archival storage format to have. Finally, we propose the layout and design of a new archival structured data storage format, Yosemite, based on our analysis.

1 Introduction

In today's world, data is being generated at an enormous rate. About 97 Zettabytes of data was generated in 2022 alone, and the number is expected to grow at a rate of about 23% annually [1]. Even though a small fraction of this data might be essential enough to archive, the absolute numbers are still pretty high. Storing data reliably for the long term is essential to preserve the world's knowledge and findings of the previous generations for future generations to build upon. Rapid innovations in technology result in paradigm shifts every few decades; hence it is essential to ensure that future generations have the means to interpret generations-old archival data assuming technology obsolescence is the norm. This can be ensured by carefully designing solutions with the future in mind while building solutions to archive data [12]. Data comes in different forms, such as binary, text, image, audio, video, and structured tables, and hence it is necessary to ensure that the format in which different forms of data are archived is future-proof: robust, reliable, secure, efficient, backward compatible, and with minimum dependence on other technologies. Out of all the data produced in today's world, about one-fifth of it is structured data. Structured means that the data follows a certain schema and organization. Sources such as IoT sensors, analytics services, financial and commercial sectors, and several scientific experiments generate structured data. Such kind of data is often stored using specialized file formats as preserving the structure, semantics, and relations between the different fields of the data is of utmost importance while storing such data. One of the main purposes of storing structured data for long periods of time is to be able to perform historical analysis on old data and basically compare the current and previous state of the world. Several different file formats have been proposed over the years, such as CSV [19], JSON [20], ORC [7], Parquet [2], HDF5 [6], and NetCDF [5]. All these different file formats were designed for certain use cases and sometimes specific data management engines and hence differ in their internal representation format, orientation, encodings, human readability, compression levels, access speed, and storage efficiency. Although none of these file formats were designed with the goal of preserving structured data for long periods of time, many of them have certain specific features and design choices that can be useful for archival data storage. The contributions of this paper are as follows:

- We introduce several popular file formats used for storing structured data and provide a detailed background on each one of them.
- We analyze the different file formats introduced based on their ability to store archival data reliably for extended periods of time.
- We propose several features that a structured data file format should possess in order to be capable of being used for archiving.
- Based on our feature recommendations, we propose the design of a new archival structured data file format, Yosemite, inspired by the designs of existing file formats but optimized for long-term data storage rather than read/write performance.

This paper is organized as follows: In Section 2, we provide a detailed background on some of the different file formats currently available for storing structured data. Then in Section 3, we do a deep dive into the file formats described in Section 2 and analyze them in terms of their ability to store archival data reliably for extended periods of time. In Section 4, we propose several features that archival file formats should possess based on our learning from Section 3. Finally, in Section 5, we propose the design of a new archival structured data file format and explain our design choices.

2 Background on Different File Formats

File formats for storing structured data occur in broadly two categories: Text-based and Binary. Examples of text-based file formats include CSV, JSON, and XML, while binary file formats include ORC, Parquet, and HDF5. Text-based file formats store data as human-readable plain-text strings using ASCII encoding coupled with a combination of delimiters and parenthesis to preserve semantic information. Binary file formats, on the other hand, store data as bytes using UTF-8 encoding. Text-based file formats are generally preferred when human readability is required, while binary file formats are preferred when storage efficiency is important. Since structured datasets commonly consist of more numeric fields than text fields, storing them in binary format instead of text format is a better choice.

2.1 CSV

CSV or Comma Separated Values format is a simple text-based format for storing 2-dimensional data. CSV uses comma (,) to separate data in different fields. The first row in a CSV file is generally a comma-separated list of fields, and the subsequent fields are comma-separated values. Although CSV files mostly use a comma as a delimiter, they sometimes also use characters like spaces, tabs, pipes (|), or semicolons (;) as delimiters. Most CSV readers and writers allow clients to use their choice of delimiter. CSV files are readable by spreadsheet programs such as Microsoft Excel or Google Sheets and are used to exchange tabular data between databases and spreadsheets. The simplicity of CSV files being just a bunch of ASCII characters makes it a reliable and future-proof format for storing critical information. On the other hand, being text-based, CSV falls behind in its storage and access efficiency.

2.2 JSON

JSON (JavaScript Object Notation) is another text-based file format commonly used in web applications for representing API requests and responses. JSON's structure is a little more complex than just comma separations of CSVs, as JSON supports storing nested fields and complex data types such as arrays. Unlike CSV, JSON has a notion of data types and supports all the data types that JavaScript supports, explaining its preference in the web ecosystem. JSON format is a little more inflated than CSV, as it uses a combination of different brackets and parentheses to represent hierarchical relationships in the data. JSON is more human-readable than CSV due to its ability to represent complex and hierarchical data in a concise manner and hence is preferred where user experience is crucial. Similar to CSV, JSON has widespread usage and is supported by most databases and data management tools.

2.3 ORC

ORC, or Optimized Row Columnar format is a columnar file format, that was designed to be an efficient format for storing Hive tables. ORC is the successor to the Record Columnar (RCFile) format, which was developed in collaboration with Facebook. In ORC files, data is organized as row groups called stripes, and metadata is stored in the file footer. In ORC, each stripe is 250MB by default. The default stripe size was chosen to best match the default HDFS [18] block size, which is 256MB. ORC's popularity slowly declined and is getting replaced with Parquet since it was primarily designed for the Hive and Pig ecosystems while the world moved towards using the Hadoop ecosystem for big data processing.

2.4 Parquet

Apache Parquet is a binary open-source columnar file format for efficient storage and retrieval of columnar data initially designed for the tools within the Hadoop ecosystem. Parquet is quite similar to ORC as it also organizes data in terms of row groups. It supports nested data structures, major data types, and several encoding schemes, such as dictionary encoding, run-length encoding, bit-packed encoding, and delta encoding. Parquet also comes with several choices for column compression, which include GZIP, Snappy, LZ4, and ZSTD. One of the main features of Parquet is that it stores file, row group, and column level metadata in the form of table schema and row group and column chunk statistics such as Min/Max values. The presence of such statistics-based metadata in Parquet files allows readers to read out only the data chunks that are required by a particular query resulting in reduced I/O and overall reduced data movement. Over the years, Parquet has grown to become the file format of choice for almost any data processing system, with most of them being outside the Hadoop ecosystem. The proliferation of Parquet users has resulted in the development of a bunch of Parquet access libraries and SDKs for different programming languages and data processing frameworks.

2.5 HDF5

HDF5 is an open-source binary file format that supports storing large, complex, and heterogeneous datasets for scientific data analysis created at the National Center for Super-Computing Applications (NCSA). HDF5 uses a "file directory" structure as in Linux/Unix, which enables data stored in HDF5 to be accessed in a POSIX-like manner. HDF5 format consists of low-level data objects such as a super-block, B-tree nodes, object headers, collections, local heaps, and free spaces. It uses these low-level objects to form two main high-level objects: groups and datasets. Datasets are typed multi-dimensional elements, and Groups are container structures that hold datasets and other groups. These high-level objects are then exposed to the users using the user-facing access APIs. In HDF5, metadata is stored in the form of user-defined named attributes attached to groups and datasets. HDF5 format has high read speeds as it was designed for fast access to large sets of atmospheric and time-series data. HDF5 has data access libraries in all major programming languages and a mature developer ecosystem.

2.6 NetCDF

The Network Common Data Form (NetCDF) file format is an open-source binary scientific data storage format that was developed at the University Corporation for Atmospheric Research (UCAR) for array-oriented atmospheric data storage. It was derived from the HDF5 file format with some restrictions. NetCDF is self-describing, portable, scalable, appendable, sharable, and archivable. NetCDF is based on the Java common data model, which has a data access layer to handle data reading, a coordinate system layer to identify the coordinates of the data arrays and a scientific data type layer that identifies specific types of data such as grids, images, and point data. NetCDF has officially maintained API interfaces in Java, Fortran, C, and C++. There exists a parallel version of NetCDF called parallel-NetCDF that was developed by Argonne National Laboratory and Northwestern University using MPI-I/O, which is a parallel I/O library.

3 Analysis of Existing File Formats

In Section 2, we looked at several different structured data file formats that are used widely today. In this section, we aim to analyze these formats in terms of their ability to become a reliable archival data format.

First, we looked at text-based file formats such as JSON and CSV. These file formats are easy to read and write and are widely accepted due to their simplicity. Also, their specifications are pretty simple and have remained unchanged for decades and are likely to stay like that. The simplicity and stability are a plus for any archival storage format. The main downside of these text-based file formats is that they use a lot more space than binary file formats due to their ASCII encoding of every character they store. In archival storage, maximizing storage density is of utmost importance; hence text-based file formats fall behind by a huge margin when considered for archival storage. In terms of metadata storage, text-based formats ideally can store metadata in text format, but for some reason, they don't store any.

Then, we looked at some binary file formats such as ORC, Parquet, HDF5, and NetCDF. Parquet and ORC were mostly created for use in data management and processing tools across the Hadoop [11] ecosystem, while HDF5 and NetCDF were developed for use in Scientific analysis to store time-series data, such as in atmospheric monitoring. The main benefit of binary file formats like these is that they are quite space efficient. Structured data being mostly consisting of integers and doubles, binary file formats can store them using the lowest possible number of bits using something like UTF-8 encoding, unlike text-based formats where every character needs to be encoded in ASCII. Since binary file formats mostly store columnar data, they unlock the capability to apply compression and encoding to the columns for better storage efficiency. Some might argue that more compression means data needs to be decompressed before processing, but lately, researchers have come up with algorithms that can execute compute operations directly on compressed and encoded data [9]. Additionally, these file formats have their specifications and format layouts open-sourced and easily available, which is an essential requirement for archival storage formats as it guarantees a long lifetime and high stability. Another benefit of binary file formats is that they store a wide variety of metadata in their headers and footers, that make these file formats self-describing and provide several optimization opportunities to the clients.

Both these text-based and binary file formats also have a wide variety of open-source data access libraries and frameworks available that are maintained by the community, hence making it easy for users/clients to integrate support for these file formats into their applications.

Some areas in which all of these file formats are lacking are data integrity and security. Data integrity is quite important for file formats if they are required to store data for long periods of time. This is because, with time, due to several natural factors, the underlying bits often get corrupted or flipped, and it is important to catch these errors early. Security is also essential as archival data often consists of critical information such as passwords, card numbers, and social security numbers. To protect such information from malicious agents, password protection or encryption support in structured data file formats is critical.

4 Features of an Archival File Format

In this section, we propose some features that we believe are essential for a structured data file format to be able to store archival data for long periods of time. There can always be more ways we can make a strong archival file format, but these are some of the most essential ones, according to us.

4.1 Open-Source and Well-Documented

It is quite important that the file format implementation is not controlled by a closed-source technology company. Although the file format technology might be very robust and well-maintained, it might be the case that a particular company that developed and maintained the file format primarily for its business needs stopped maintaining the project due to organizational changes. In such an occurrence, people outside the organization using that file format to archive their data would be adversely affected as all their archived data would become inaccessible due to the access libraries and format specifications becoming obsolete and difficult to find. Hence, it is important that we use file formats that are open-source, with the format specifications well-documented. So, even after several years, when the file format is not widely used anymore, people using it to archive their data will be able to easily find the source code of the access libraries or maybe even write an access library using the specifications, to be able to read back their files. In this way, using open-source file formats gives users more control over the lifetime of the file format and over their archived data in return [4].

4.2 Space Efficient

Using storage space efficiently is one of the most important considerations when thinking of archival storage. Organizations want to spend the least amount of money for storing archival data because such data does not drive their business decisions anymore. That's why the goal is to be able to pack as much data as possible within a small storage space so that the "$/gigabyte" cost for storing data is minimal. When looking at text-based and binary file formats, binary file formats are the most space efficient as they do not need to ASCII encode everything like text-based formats.

In terms of storing different data types efficiently, it is important that the file format supports variable-sized data types and does not have only fixed-sized data types. This means the data type should not use more bytes than absolutely necessary for storing the particular value. For example, popular file formats usually have support for 8, 16, 32, and 64 bit versions for integer and double data types and variable-sized strings using offset vectors. Another important factor to consider for space efficiency is to have a columnar file format that enables better compression and encoding of data, but compression/encoding deserves a section of its own, so we discuss it separately.

4.3 Highly Compressed and Encoded

As we discussed in Section 4.2, storage space efficiency being critical for archival storage, compression, and encoding techniques, becomes an important factor for archival file formats. Since we are not concerned about read and write speeds while dealing with archival data, we should focus on the compression algorithm that has the highest compression ratio and is lossless. The most widely used lossless compression algorithms in modern columnar file formats for column-level compression are Snappy, LZ4, and ZSTD [10]. Out of these 3, ZSTD trades off speed for its best compression ratio. On the other hand, Snappy is faster but provides only lightweight compression. LZ4 might be the compression algorithm of choice if a decent balance of both read/write speed and compression ratio is required; otherwise ZSTD seems to be the best choice. Once again, tying back to Section 4.1, we would like to reiterate that only widely-used standard compression algorithms with open-source implementations and specifications should be leveraged. In terms of encoding techniques, we can use Run-Length encoding, Dictionary Encoding, and Bit-Packed Encoding as these encoding algorithms integrate nicely with the compression algorithms mentioned above [16].

4.4 Ensure Data Integrity

An archival file format should provide ways to check the integrity of the data stored in it. Integrity checking is important because, depending on the media the file was stored in, it might be the case that the underlying bytes experience some form of bit rot or corruption. External agents such as heat, acoustic waves, magnetic interference, air resistance, or simply mechanical wear and tear can be the cause of data corruption [17].

To deal with such issues, it is often the case that when data is written to a file for archiving, it is checksummed, and the hash is also archived for future integrity-checking purposes. Some widely used algorithms used for checksumming are MD5, CRC-32, SHA-1, SHA-256, and SHA-512. MD5 is the fastest and generates a 128-bit checksum. On the other hand, SHA-256 and SHA-512 generate the most secure hashes but are slow. SHA-1 generates a 160-bit hash and provides a good balance of security and speed [**?**]. Additionally, Error-correcting codes (ECCs), such as Repetition codes and Hamming codes which use parity bits to detect errors in bit strings, can also be used [15].

4.5 Secure

Some categories of archival data should be stored securely, away from the reach of malicious agents. Examples of such kinds of data include classified government information, financial institution's statistics, health records of individuals, etc. At the file level, password protection and encryption are the most common ways of dealing with security. For example, when we password-protect a PDF file, the contents of the PDF are encrypted using a 256-bit AES encryption. Asymmetric encryption techniques such as Public/Private key encryption can also be a viable option [23]. In this case, writers of the archival files would encrypt the file contents with a public key, and the readers are required to have access to a private key in order to decrypt the files.

A major issue while considering encryption of archival data is that there needs to be a way to securely archive the passwords and the private keys also. We leave this as future work as this is out of the scope of our current research.

4.6 Self-Describing and Metadata Rich

Information that describes data is known as metadata. When archiving valuable data for extended periods of time, it is essential that the data is accompanied by information that helps the future readers of the dataset get an idea of the data stored without actually reading all of it out. Structured data file formats should have a dedicated metadata section at the file header or footer so that access libraries can read out the metadata fast without requiring to perform a bunch of random accesses as metadata access often sits on the critical path during dataset discovery [22].

For example, metadata might consist of dataset describing information such as row count, column count, schema, data types of fields, null counts of columns, in-memory size of the data, compression, and encoding information, sort orders, endianness, range of numeric columns, and, primary key. For access efficiency, although not necessary for archival file formats, it might contain Min/Max statistics for rows groups, and column chunks. Finally, files should also have some information describing the origin of the data, basically details about when, where, and by whom the particular file was created. Finally, the file format should store its version along with the version of the access library that was used to create a file.

4.7 Block-Aligned Data Boundaries

Organizations storing a bunch of archived datasets might occasionally want to run some form of compute operation or analysis on them. If the amount of data archived is huge, for example, in the order of petabytes, it might not be economical to read out all of the data to client machines for analysis. In such a case, running compute operations directly inside the storage nodes is a good alternative as it does not require moving the data [8]. Since present storage devices mostly allow access at block granularity, it is important to ensure that disk blocks contain self-contained data so that compute operations can process blocks independently. To have self-contained blocks, file formats should follow block alignment as much as possible. For example, file formats could store row groups or column chunks in sizes of a device block [13]. Block alignment of data inside files has the added advantage of reduced device interference during concurrent accesses to a single file and, in turn, reduced device wear and tear, ultimately increasing the device's lifetime.

5 Yosemite: An Archival Structured Data File Format

In this section, we present the layout and design of our new archival structured data file format, Yosemite.

5.1 Format Layout

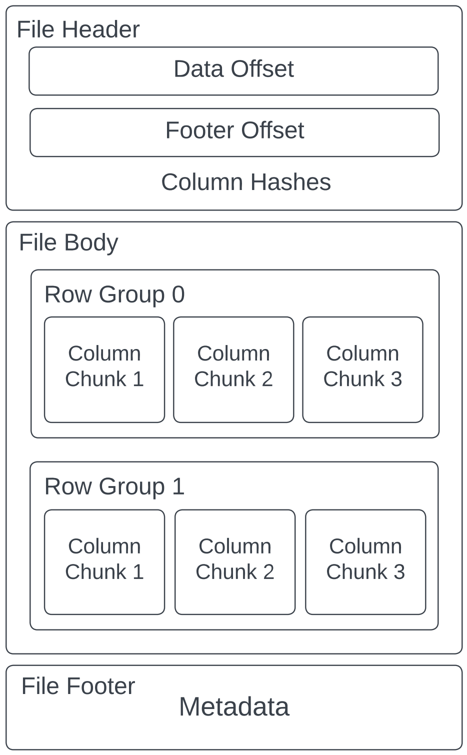

Figure 1: Layout of Yosemite

The layout of our file format is organized into three parts. First comes the file header, then the file body, and then the file footer. In the file header, we store the file body offset, the file footer offset, and then the header information. The file body contains the tabular data in row groups, where every row group is of the same size. Row groups contain column chunks, each chunk representing the part of the column for that row group. Lastly, we have the file footer, which entirely consists of the footer information. The layout of Yosemite is shown in Figure 1.

5.2 Description

Based on our observations and speculations from Section 3 and 4, we propose a new file format, Yosemite, designed to store structured tabular data reliably for extended periods of time.

Since we want our file format to have a long lifetime, we plan it to be an open-source file format from the beginning. We also plan to have the format specification and design documents publicly available in the form of RFCs and GitHub Wikis. Additionally, we would like to have a community grow around our file format, with people trying to use it for archiving their data and providing feedback. We also hope that developers specializing in different programming languages would contribute access libraries for Yosemite and help in expanding its user base.

Archival storage being our major goal, we want a binary file format since binary file formats use less space to store data as compared to text-based formats. We plan on using serialization protocols such as Google Protocol Buffers (commonly known as Protobuf) [14] or Apache Thrift [3] to serialize tabular data into raw bytes. Protobuf is what we would prefer to use since it has a bigger community, support for more data types, better extensibility, and more resources and documentation to refer to.

In terms of the underlying data layout, we base our file formats design to be a columnar file format as columnar file formats can be easily encoded and compressed, allowing to optimize for space which is important for archival storage. Columnar file formats are also more efficient in terms of their read speeds in OLAP (Online Analytical Processing) workloads accessing only a few columns, as columnar file formats result in few sequential accesses rather than a bunch of random accesses [21].

For compression and encoding, we would have support for dictionary encoding, run length encoding, and bit-packed encoding, depending on the field types. For example, fields with repeated values will benefit from run-length encoding (RLE), and fields with a well-defined set of values will benefit from dictionary encoding. After encoding, we would like to compress our columns using ZSTD for maximum space savings.

Our file format should also ensure data integrity, which means readers should be able to tell if there is data corruption present in the file. We plan on implementing this by calculating SHA-256 hashes of every column and storing them in the file header. We would also calculate and store the SHA-256 hash of the file metadata, as metadata is equally important as the data and is often the entry point while reading a file. We leave implementing Error-correcting Codes (ECCs) as future work.

In terms of ensuring data access security, we plan on having RSA-based public/private key encryption support in our file format. After writing out the file, the writers would encrypt the file using a public key. Every organization will have its own unique public key. While reading the file, the clients need to have access to a private key to be able to decrypt the file contents.

We would like to have the data stored in our file format to be self-describing, and there's only one way to make data self-describing: accompany with enough useful metadata. We plan on storing all the metadata fields as discussed in Section 4.6. We also plan to reserve space for a limited amount of user-specified metadata that can be filled in by the file writers based on application-specific use cases. All the metadata would go into the file footer, with the footer offset being stored in the file header.

Finally, to be able to implement in-storage processing of our files, we need to ensure one-to-one mapping between logical data boundaries and block/page size of underlying media. We plan to use row groups as logical boundaries in our file format and allow file writers to use a user-specified byte size for row groups based on the block/page size of the underlying media where the file will be stored. We understand that it is not always possible to write rows to precisely fit a row group chunk; hence our file writers use padding whenever necessary.

6 Future Work

For future work, we plan on exploring more on the security and encryption front of different file formats. Also, techniques for reliably archiving passwords and private keys are an open area for research for us. Finally, we would like to implement the design of our proposed file format by releasing an official specification document and by building and maintaining open-source access libraries.

7 Conclusion

In this paper, we look at structured data file formats from an archival storage point of view. We study several structured data file formats, both text-based and binary, such as CSV, JSON, Parquet, ORC, HDF5, and NetCDF, and analyze them in terms of their ability to store archival data reliably for long periods of time. We identify several features from these file formats, which we believe are essential for an archival structured data file format to have. Along with the features identified, we add some more features that don't currently exist, such as password protection and checksumming, and present an exhaustive list of archival-first file format features. Based on our feature recommendations, we propose the design of Yosemite, a new structured data file format for storing archival data. We feel that we have only explored a fraction of what is required for a file format to store archival data reliably; hence a lot of research is still remaining, which we aim to do as future work.

8 Acknowledgement

I would like to extend my gratitude to Professor Ethan L. Miller for offering the awesome "CSE290S: Preserving digital information for future generations" class. Additionally, I would like to thank my classmates for their valuable perspectives in the class and for helping me make the most out of this class.

References

[1] Amount of data created daily (2023). https://explodingtopics.com/blog/data-generated-per-day.

[2] Apache parquet. https://parquet.apache.org/.

[3] Apache thrift. https://thrift.apache.org.

[4] File formats and standards. https://www.dpconline.org/handbook/technical-solutions-and-tools/file-formats-and-standards.

[5] Netcdf. https://www.unidata.ucar.edu/software/netcdf/.

[6] The hdf5® library file format. https://www.hdfgroup.org/solutions/hdf5/.

[7] Apache orc: High-performance columnar storage for hadoop, 2018.

[8] Adams, I. F., Agrawal, N., and Mesnier, M. P. Enabling near-data processing in distributed object storage systems. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems* (2021), pp. 28–34.

[9] Agarwal, R., Khandelwal, A., and Stoica, I. Succinct: Enabling queries on compressed data. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 337–350.

[10] Aher, R. N., and Pande, M. Analysis of lossless data compression algorithm in columnar data warehouse. In *2022 6th International Conference On Computing, Communication, Control And Automation (ICCUBEA* (2022), IEEE, pp. 1–4.

[11] Apache Software Foundation. Hadoop.

[12] Baker, M., Keeton, K., and Martin, S. Why traditional storage systems don't help us save stuff forever. In *Proceedings of the First Conference on Hot Topics in System Dependability* (USA, 2005), HotDep'05, USENIX Association, p. 7.

[13] Chakraborty, J., Jimenez, I., Rodriguez, S. A., Uta, A., LeFevre, J., and Maltzahn, C. Skyhook: Towards an arrow-native storage system. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (2022), IEEE, pp. 81–88.

[14] Currier, C. Protocol buffers. In *Mobile Forensics–The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*. Springer, 2022, pp. 223–260.

[15] Hamming, R. W. Error detecting and error correcting codes. *The Bell system technical journal 29*, 2 (1950), 147–160.

[16] Hesabi, Z. R., Tari, Z., Goscinski, A., Fahad, A., Khalil, I., and Queiroz, C. Data summarization techniques for big data—a survey. *Handbook on Data Centers* (2015), 1109–1152.

[17] Lauber, J. Bit rot and silent data corruption in digital audiovisual preservation.

[18] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)* (2010), Ieee, pp. 1–10.

[19] Wikipedia contributors. Comma-separated values — Wikipedia, the free encyclopedia, 2023. [Online; accessed 14-June-2023].

[20] Wikipedia contributors. Json — Wikipedia, the free encyclopedia, 2023. [Online; accessed 14-June-2023].

[21] Zeng, X., Hui, Y., Shen, J., Pavlo, A., McKinney, W., and Zhang, H. An empirical evaluation of columnar storage formats. *arXiv preprint arXiv:2304.05028* (2023).

[22] Zhang, W., Byna, S., Tang, H., Williams, B., and Chen, Y. Miqs: Metadata indexing and querying service for self-describing file formats. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), pp. 1–24.

[23] Zhang, Y., Xu, C., Cheng, N., and Shen, X. Secure password-protected encryption key for deduplicated cloud storage systems. *IEEE Transactions on Dependable and Secure Computing 19*, 4 (2021), 2789–2806.