Large Web Archive Collection Infrastructure and Services

Xinyue Wang

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Edward A. Fox, Co-chair
Zhiwu Xie, Co-chair
Hoda M. Eldardiry
Ismini Lourentzou
Jiangping Chen

Dec 5, 2022
Blacksburg, Virginia

Keywords: Web Archive, Digital Library, Big Data Infrastructure

Large Web Archive Collection Infrastructure and Services

Xinyue Wang

(ABSTRACT)

The web has evolved to be the primary carrier of human knowledge during the information age. The ephemeral nature of much web content makes web knowledge preservation vital in preserving human knowledge and memories. Web archives are created to preserve the current web and make it available for future reuse. A growing number of web archive initiatives are actively engaging in web archiving activities. Web archiving standards like WARC, for formatted storage, have been established to standardize the preservation of web archive data. In addition to its preservation purpose, web archive data is also used as a source for research and for lost information recovery. However, the reuse of web archive data is inherently challenging because of the scale of data size and requirements of big data tools to serve and analyze web archive data efficiently.

In this research, we propose to build web archive infrastructure that can support efficient and scalable web archive reuse with big data formats like Parquet, enabling more efficient quantitative data analysis and browsing services. Upon the Hadoop big data processing platform with components like Apache Spark and HBase, we propose to replace the WARC (web archive) data format with a columnar data format Parquet to facilitate more efficient reuse. Such a columnar data format can provide the same features as WARC for long-term preservation. In addition, the columnar data format introduces the potential for better computational efficiency and data reuse flexibility. The experiments show that this proposed design can significantly improve quantitative data analysis tasks for common web archive data usage. This design can also serve web archive data for a web browsing service. Unlike the conventional web hosting design for large data, this design primarily works on top of the raw large data in file systems to provide a hybrid environment around web archive reuse. In addition to the standard web archive data, we also integrate Twitter data into our design as part of web archive resources. Twitter is a prominent source of data for researchers in a variety of fields and an integral element of the web's history. However, Twitter data is typically collected through non-standardized tools for different collections. We aggregate the Twitter data from different sources and integrate it into the suggested design for reuse. We are able to greatly increase the processing performance of workloads around social media data by overcoming the data loading bottleneck with a web-archive-like Parquet data format.

Large Web Archive Collection Infrastructure and Services

Xinyue Wang

(GENERAL AUDIENCE ABSTRACT)

The web has evolved to be the primary carrier of human knowledge during the information age. The ephemeral nature of much web content makes web knowledge preservation vital in preserving human knowledge and memories. Web archives are created to preserve the current web and make it available for future reuse. In addition to its preservation purpose, web archive data is also used as a source for research and for lost information discovery. However, the reuse of web archive data is inherently challenging because of the scale of data size and requirements of big data tools to serve and analyze web archive data efficiently.

In this research, we propose to build a web archive big data processing infrastructure that can support efficient and scalable web archive reuse like quantitative data analysis and browsing services. We adopt industry frameworks and tools to establish a platform that can provide high-performance computation for web archive initiatives and users. We propose to convert the standard web archive data file format to a columnar data format for efficient future reuse. Our experiments show that our proposed design can significantly improve quantitative data analysis tasks for common web archive data usage. Our design can also serve an efficient web browsing service without adopting a sophisticated web hosting architecture. In addition to the standard web archive data, we also integrate Twitter data into our design as a unique web archive resource. Twitter is a prominent source of data for researchers in a variety of fields and an integral element of the web's history. We aggregate the Twitter data from different sources and integrate it into the suggested design for reuse. We are able to greatly increase the processing performance of workloads around social media data by overcoming the data loading bottleneck with a web-archive-like Parquet data format.

Acknowledgments

I would first like to thank my supervisors and co-advisors, Dr. Edward A. Fox and Dr. Zhiwu Xie, for consistent support and guidance throughout my Ph.D. research. Your expertise was invaluable in formulating and pushing my research goals.

I would like to thank my external committee member, Dr. Jiangping Chen, who guided me to the scientific research field in computer science. Your guidance helped me to pursue a journey toward my research.

I would also like to thank the rest of my committee, Dr. Hoda Eldardiry and Dr. Ismini Lourentzou, for your insightful suggestions.

In addition, I would like to thank the people I have met in our lab: Abhinav Kumar, Amirsina Torfi, Bill Ingram, Bipasha Banerjee, Eman Abdelrahman, Liuqing Li, Ola Karajeh, Prashant Chandrasekar, Satvik Chekuri, Saurabh Chakravarty, Siyu Mi, Xuan Zhang, Yufeng Ma, and Ziqian Song. Our cherished time spent together in the lab and in social settings was precious.

I would like to thank my family: my father Xiaowu Wang, my mother Hemei Chen, my uncle Xiaoji Liu, my aunt Yinong Chen, and other relatives. Your understanding, support and help are invaluable to my years while pursuing my degrees.

Thanks go to the Department of Computer Science and University Libraries for funding and for facilities that supported my research.

Thanks also go to the Institute of Museum and Library Services for support under grant LG-71-16-0037-16 and National Science Foundation under grants IIS-1619028 and IIS-1619371.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The web has evolved to become a primary carrier of our knowledge and memory. It is, however, inherently decentralized, dynamic, and ephemeral. Despite valiant efforts [36, 108] to inject a sense of temporality in the web architecture [63], most of the web of today is focused on disseminating the most up-to-date information and does not leverage any built-in mechanism to persist representations of web resources that are either purposefully retired, replaced, or vanishing due to negligence. As a consequence, increasingly more hyperlinks break or no longer point to their intended representations [69, 97]. This phenomenon, referred to as "link rot," endangers the lineage of our history.

To battle this loss, web archives initiatives started collecting and preserving the current web content for potential future reuse. A growing number of cultural heritage institutions actively engage in web archiving. A 2011 survey reported 42 web archiving initiatives around the world [52]. The number increased to 68 in 2015 [33]. A 2018 survey reported 119 web archiving programs in the United States alone [43], 61% of which were part of university libraries or archives. Large-scale, comprehensive web archiving initiatives include the Internet Archive [14], the Common Crawl [34], and many programs at national libraries and archives. These initiatives have preserved large amounts of web content, some dating as far back as the 1990s. As of 2021, the Internet Archive alone had archived about 50 petabytes of web content, including over 500 billion web pages [16].

Similar to search engines, the vast majority of the content in web archives is automatically collected by web crawlers. Unlike search engines, however, the collected content is not primarily intended for immediate consumption. There is no rush to have it be fully indexed and made available for search and browsing through a highly available and highly scalable platform like Google. Instead, the HTTP requests from the crawlers and the responses sent back by the origin servers are concatenated and aggregated into larger files intended to be stowed away for safekeeping in archival storage. Addressing this need, the Web ARChive, or WARC [62] file format became an international standard in 2009. Archiving the Web has since become synonymous with creating WARC files.

Some web archives provide limited browsing interfaces to WARC files. For example, the Internet Archive's WayBack Machine [15] allows users to retrieve and playback a historic web page using its original URL and a timestamp. Similar browsing tools have been developed and adopted elsewhere [3, 30, 114]. Browsing functions like these typically leverage CDX files [29], which point to the archived record and also contain metadata such as the URL,

timestamp, MIME type, and response code, to assist page-by-page browsing. Nevertheless, the central theme of today's web archives is still predominantly on web content collection and preservation. As a result, the archival system architecture is centered around WARC files and not optimized for research-driven analytics workflows. That archival architecture can not be adopted widely due to the high technology barrier built around the WARC standard. The browsing service is missing from many web archive collections other than large institutions like the Internet Archive. Reuse beyond the prescribed browsing pattern is also rarely supported.

This work aims to improve web archive data reuse by building a specialized infrastructure. We supplement the standard WARC preservation format and adopt a modern data format like Parquet as the underlying web archive data source for reuse cases like quantitative analysis and browsing services. Our proposed architecture includes modern industry frameworks and tools that address the computational need for large web archive collections, efficiently and scalably. We believe our architecture introduces an opportunity for web archive service providers to serve web archive data more conveniently and with lower cost. Users can benefit from the expansion of web archive services around the community to have better access and more efficient analysis of web archive data.

1.1 Background and Motivation

At Virginia Tech, in the Digital Library Research Laboratory (DLRL) and in University Libraries, web archive data sharing and reuse is a critical part of research activities and library services. The University Library, in partnership with the departments of Computer Science and Mechanical Engineering, as well as the University of North Texas Department of Library and Information Science, collaborated in a research grant, LG-71-16-0037-16, funded by the Institute of Museum and Library Services, toward building an efficient library cyberinfrastructure strategy to provide services around big data sharing and reuse. In particular, there was a focus on web archive data. The Digital Library Research Laboratory and University Libraries at Virginia Tech have collected over 1 terabyte of web pages and over 6 billion tweets, as part of their web preservation activities. These data can be used by researchers and students in various areas such as history, tourism, or political science. The web archive data collection has gained attention in many disciplines. Examples of the uses include tracking and analyzing live events such as earthquakes, political events, community activities, violence, and crime prevention. [21, 41, 67, 68, 99] The data has also been curated into the STEM learning in courses such as Multimedia, Hypertext and Information Access; Computational Linguistics; Digital Libraries; and Information Retrieval. [26, 32, 35, 39, 116]

While the web archive collections are gaining more attention and use, user's experience with web archive data has revealed issues regarding accessing and analyzing the data efficiently:

- Users can only work with the data in raw formats like WARC instead of standard

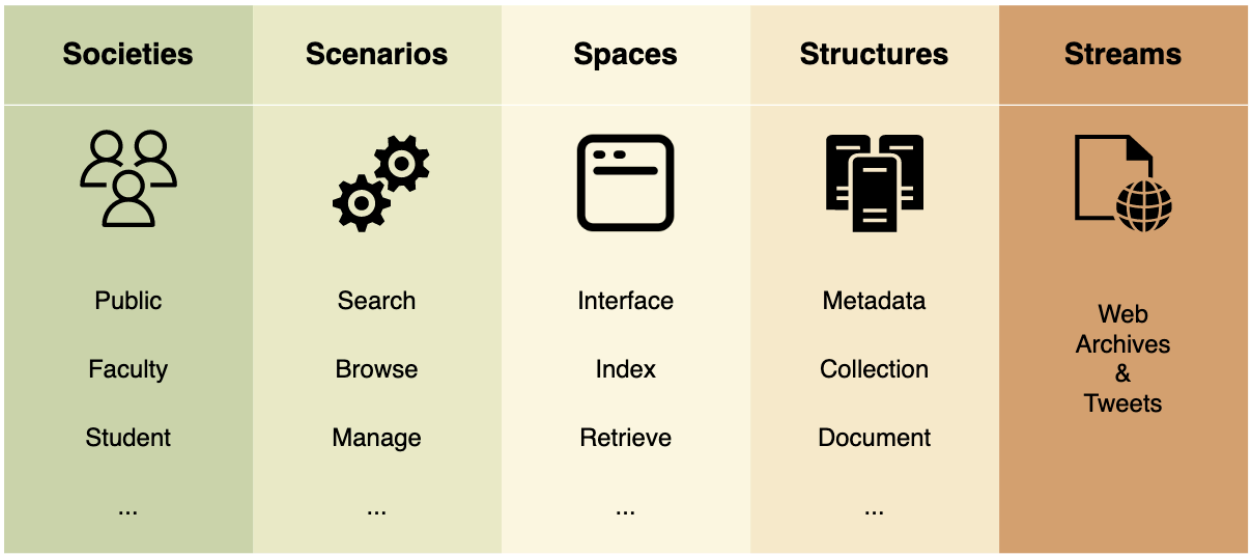


Figure 1.1: An overview of the web archive infrastructure based on the 5S theory [46]

formats like CSV, JSON, or others for database systems.

- Users can not easily browse web records if the data is not hosted through other services.
- Users might not have sufficient processing power to handle large collections.
- Existing web archive tools may not be able to meet users' data exploration and manipulation needs.

To overcome these challenges, we need a better infrastructure to aid the data-intensive science around web archives, as demonstrated in the scientific infrastructure paradigm from the Fourth Paradigm [58]. This research aims to build a digital library cyberinfrastructure that can improve the efficiency of web archive data reuse. We follow the unified formal theory of digital library design under the 5S digital library framework [46]. Figure 1.1 shows a contextualized version of the general structure of the 5S model. In this application, web archive data and social media data such as tweets comprise the majority of the digital library's content (Streams). Particularly, we strengthen the Structures and Spaces parts of the model by reconstructing the web data storage representation and designing a data reuse (Scenarios) infrastructure for web archive data practitioners (Societies) that can achieve greater efficiency and convenience.

The green part of Figure 1.2 summarizes the general web archive data service pattern in the university library. Large data sets like the web archive or Twitter data are generally stored in file systems in raw format. Some web archive data could be accessed through third-party services like ArchiveIT[1], where the original data is not locally hosted. The nature of web

[1] https://archive-it.org/

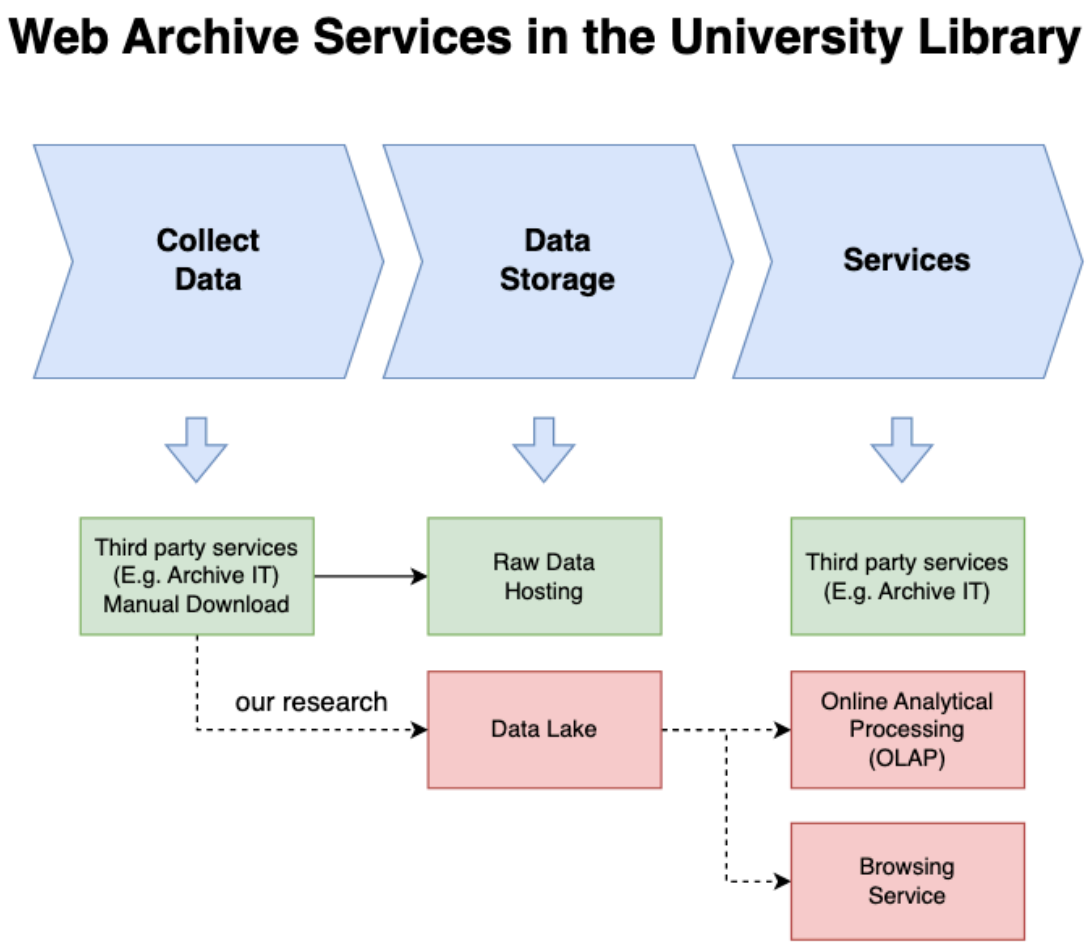


Figure 1.2: Data and service pattern in the university library

archive data makes it challenging to integrate into the existing library infrastructure: large data size and inefficient data format. Though difficult to process, the web archive data consistently draw user's interest due to the rich information. Instant access is a typical scenario where users browse individual web records within a collection. Though ArchiveIT or the Wayback Machine can provide access services for web archive records, those reply upon the service provider. There exist needs where institutions can collect and host their local collections. In addition to instant access, users typically perform extensive analytical tasks on web archive data nowadays. Depending on the size of the collection, such processing typically needs a lengthy period of computation. Furthermore, the web archive data includes more than just ordinary web page data. Social media data such as Twitter represent a big portion of current web knowledge and are typically represented in a format distinct from standard web archive data. The disparities in data representation create a barrier to the efficient and unified usage of web data.

We intend to develop an infrastructure that can handle both traditional web archive col-

lections and social media archive collections for efficient analytical processing and instant access. We hope that the infrastructure's design can maintain web archive data in a native and raw manner while being more efficient than the existing standard. Typically, this concept is referred to as a data lake[2] repository as shown in Figure 1.2. On top of the data lake, the infrastructure can provide efficient online analytical processing (OLAP[3]) for web archive data. We also want to enable efficient web hosting performance for instant data access (web browsing) under the same infrastructure design.

1.2 Research Objectives

The first objective of the research focuses on enhancing the analytical processing efficiency for web archive data under a big data infrastructure. One major bottleneck in web archive big data analysis is the underlying data storage representation, which prevents reading-intensive workloads from loading and feeding data to the processing pipeline efficiently. Through our distributed infrastructure for big data processing, we intend to collect performance metrics for existing analytical processing methods. Then, we compare the existing approaches to our proposed modifications to the data representation and processing pipeline to observe performance enhancements.

Second, we investigate the possibility of integrating a web hosting capability into the infrastructure designed for analytical processing. Despite the fact that large-scale quantitative data analysis is the primary focus of our research problem, there is still a broad interest among researchers in closely studying individual archived web pages. We should incorporate this potential into our infrastructure. Existing strategies for large-scale web hosting typically involve considerable computational and data storage resources to satisfy performance requirements. In our case, we intend to investigate a solution that leverages restricted resources while retaining good performance under the analytical infrastructure.

Thirdly, social media data, such as Twitter data, is a common resource in web archive collections. However, the data representation for social media data typically differs from the usual format for web archiving. We intend to include Twitter data into our proposed infrastructure design and increase Twitter data reuse efficiency in a manner comparable to that of standard web archive data. We aim to investigate Twitter data hosting and quantitative analytical processing capabilities.

[2] https://learn.microsoft.com/en-us/azure/architecture/data-guide/scenarios/data-lake

[3] https://learn.microsoft.com/en-us/azure/architecture/data-guide/relational-data/online-analytical-processing

1.3 Research Questions and Hypotheses

The general research question for this research follows: How can we design a cyberinfrastructure for large web archive-related data reuse? Under this general question, we pose the following research questions:

- Question 1: Can we accelerate the common analytic tasks on large web archive collections through a big data infrastructure with alternative data formats?
- Question 2: Can we provide accessibility of large web archive collections for browsing efficiently under Question 1?
- Question 3: Can we integrate social media archives like Twitter data into the infrastructure and enable efficient reuse?

We have the following hypothesis for the questions above: Using existing industry-adopted big data tools, we can design a new infrastructure for web archive access, sharing, and analysis, and with it build a more cost effective and efficient infrastructure. Under this general hypothesis, we hypothesize specifics, corresponding to the research questions:

1. The new infrastructure can provide quantitatively enhanced performance for the general web archive data-to-insight cycles.
2. At the same time, the infrastructure can also provide accessibility for web archive records for audiences that are interested in browsing the content.
3. In addition to the web pages, the infrastructure can integrate Twitter data for efficient reuse, similar to what is done for standard web archive data.

1.4 Overview

The overall structure of this dissertation includes five chapters, including this introduction. The rest of the chapters are organized as follows:

- Chapter 2: Expedite the Web Archive Data-To-Insight Cycle with Parquet Format (see also Appendix A)
- Chapter 3: Hosting Large Web Archive Collections on Raw Data with Parquet Format (see also Appendix B)
- Chapter 4: Twitter Data Analysis Integration and Acceleration (see also Appendix C)
- Chapter 5: Best Practices
- Chapter 6: Contributions and Future Work

Chapter 2

Expedite the Web Archive Data-To-Insight Cycle with Parquet Format

In this chapter, we address our first research question: *Can we accelerate the common analytic tasks on large web archive collections through a big data infrastructure with alternative data formats?* We propose to utilize modern distributed computation technologies including Hadoop and Spark to establish our web archive online analytical system. As shown in Figure 2.1, we propose to use HDFS and modern data formats as the data lake layer for the data storage; we use Apache Spark as memory processing engine to handle analytical workloads. We propose to utilize a modern file format, Parquet, to replace the existing WARC storage format to expedite the web archive analysis. In addition, our work aligns with the big data framework where the web archive collection needs to be processed in a large computer cluster. We evaluate the infrastructure by benchmarking and comparing the performance of common web archive analytical workloads in different file formats. This work is also published in JCDL 2020 [112].

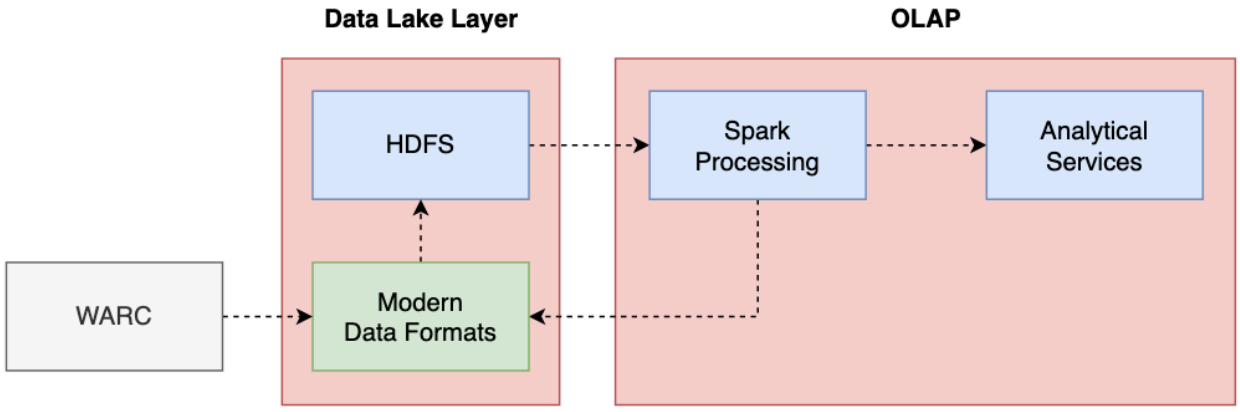


Figure 2.1: Data flow in the proposed system design

2.1 Introduction

It is difficult to make generalizations about web archives. Despite the significant growth of web archives in size, the whole web and its complete history grow even faster. Web archives therefore remain a rather thin sample. Most general purpose web archives tend to be sparse as well as shallow, capturing only a very small percentage of published versions [4, 5]. The sparsity grows even thinner when the web resource is addressed farther away from the home page. This is primarily due to the crawler-based collection method used predominantly by these archives and the associated resource, policy, and legal constraints. Given the low sampling rate of the archived web content, we need to exercise caution in generalizing and extrapolating the insights gained from close reading. In contrast, treating web archives as big data sources to extract statistics and explorative insights may be more profitable.

Reusing web archives for analytical research, however, usually requires archive users to deploy big data systems such as Apache Hadoop or Apache Spark. These systems tend to be sensitive to performance, which can easily fluctuate by orders of magnitude. Performance may therefore determine whether it is even feasible to explore certain research questions. Although a plethora of factors may impact the performance of big data systems, domain experts and big data system designers do not always align their strategies. When approaching a domain problem, big data system designers typically refrain from questioning, for a particular domain, its specific common practices. Instead, these practices are taken as matter-of-fact inputs for design. However, the domain status quo only reflects best practices extracted from past experience, and requirements that may be irrelevant in big data settings. Although keenly aware of this, domain experts do not always heed the contexts and nuances behind the drastic performance gain demonstrated by many big data systems. If no adaptation is made when using these systems, the poor initial and boundary conditions introduced by unfavorable domain common practices may penalize performance. It is in this gap that we see opportunities, particularly concerning data formats. Indeed, many de facto domain-specific data formats are being scrutinized for their performance penalty when used in big data settings [6, 54, 84, 106]. This chapter focuses on the web archival format. More specifically, we present the case for the web archiving community to move away from using WARC as the default archival format. We analyze the root causes of why WARC reduces big data analytics performance in comparison to more efficient data formats, and run controlled experiments to quantify these losses. The results also provide insights to generalize the types of archival data format we should avoid for web archives.

2.2 Related Work

In this section we present related work that helps to justify our emphasis on archival data formats.

The state of the art of processing big data exploits various patterns of the computing systems and algorithms for accessing and querying the data. For example, it is generally assumed that as the data size increases, its complexity remains largely unchanged. In other words, large datasets are comprised of structurally repetitive records with low interdependency. Large datasets can therefore be partitioned and replicated, and processed on shared-nothing commodity machines in parallel. Web archives fit perfectly into this pattern.

Although Apache Hadoop is one of the more successful implementations of this pattern, MapReduce style processing has long been used with parallel databases. Pavlo et al. [90] shows that although it takes much longer for a parallel DBMS to load raw data, once the data are in place, its query speed is much faster than Hadoop. This finding brings to the forefront the tradeoff between data loading and querying/processing. Here, data loading refers to the process of changing the layout of the original data to another one. The new layout is specified by the parallel DBMS and has been optimized for its querying performance. It naturally follows that if the original data is already laid out in the way specified by the parallel DBMS, then spending resources to reformat data would be unnecessary. Unfortunately, parallel DBMS systems, many of which were proprietary, had not thought of standardizing their data layouts as open file formats. The situation has since changed with the growth of Apache Hadoop and Apache Spark. Two standard data formats, Parquet [11] and Avro [12], have been developed from these projects and adopted as their default data persistence formats. Although not intended for this comparison, circumstantial evidence in [89] indicates that parsing data formatted in Parquet and Avro can be at least one order of magnitude faster than parsing the same data formatted in JSON.

Some domain experts have been vocal about changing domain-specific file formats for big data processing. For example, the astronomy community have had long discussions on limitations of the FITS format [54, 84, 106]. Although performance is on the agenda, the discussions tend to be surrounding qualitative features, e.g., maintaining human-readability [106]. Biologists have also complained about using HDF5 to store large microscopy data [6].

However, many researchers, particularly computer scientists working on domain problems, do not bother to advocate for data format changes. They take the status quo as is and attempt to improve performance from there. Alagiannis et al. [2] argue that with the rapid growth of research data, the performance bottleneck has moved from querying to data loading. It is therefore more advantageous to trade the query speed for the capability to perform queries directly on raw data *in situ*. If designed properly, the sub-optimal querying performance can be compensated for by the elimination of data loading. This stance persisted in Karpathiotakis et al. [66], who adapted their querying system to the ROOT file format specified by CERN's Large Hadron Collider project. Similarly, gap-filing software has been

developed to improve the I/O performance for using HDF5, netCDF [75, 117], or FITS [91] in a Spark environment. Further, many have worked towards improving the parsing performance of the inefficient yet widely adopted JSON format [71, 89]. These improvements leverage various hardware/software performance gaps [119] but can not fully eliminate the inherent inefficiency incurred by a low-performing data format. They can, however, be used to speed up the conversion from a low-performing data format to a more efficient one. A rare exception to this stoic stance comes from the ADAM project [78, 87]. There, the SAM/BAM file formats used to store large genome datasets have been replaced by a combination of Avro and Parquet formats, which helped to achieve a 50x speedup using Apache Spark.

For web archiving applications, Warcbase [72] chooses to load WARC files into HBase for querying. As such, archived data will need to persistently occupy the analytical systems, as in the "Network" pattern [115]. This is not suitable for long-term preservation. The ArchivesUnleashedToolkit [105], successor to Warcbase, also took WARC file format for granted but did not assume the existence of the accompanying CDX [29] files. Querying an ArchivesUnleashedToolkit application therefore requires repeated loading of the WARC files in full. ArchiveSpark [59], on the other hand, leverages CDX to selectively load WARC. Without sophisticated I/O scheduling, however, a full disk scan can still outperform many selective disk reads bundled together. Both the ArchivesUnleashedToolkit and ArchiveSpark perform data processing using Apache Spark. Their main difference is how they load files from the disk into memory. In our comparison, we will perform data processing using Spark SQL on top of Apache Spark. After data is loaded into memory, we assume the performance difference between Spark and Spark SQL should not be significant. Significant performance gain should therefore be attributed to the file format differences.

2.3 Reuse Web Archives as Big Data Sources

In order to evaluate performance, we must first define typical workloads against which benchmarking will be performed. Reusing web archives as big data sources has been steadily gaining momentum in research areas such as history [25, 53, 83] and communications [113]. Lin et al. observe that the frontier of contemporary historians' research focuses usually moves along about 20 to 30 years before the present time [73]. They therefore anticipate that many historians will soon start to look at the history of the 1990s and suddenly enter an age of abundance [83]: web archives have preserved unprecedentedly large amounts of historic information.

How will future web archive users, historians included, explore web archives in addition to close reading? We can only speculate, although extrapolating from problems researchers are currently exploring should give us higher confidence in our educated guesses. We identify 3 types of representative workloads on web archives (see Table 2.1), described in the following.

Take a recent research study [77] as an example. Mallapragada focuses her research on

interpreting Indian immigrants' homepages from the mid-1990s to mid-2000s; the researcher has already developed a list of URLs considered representative. We may then speculate that she would be interested in knowing how many relevant snapshots have been taken by a web archive during her time period of interest. This is a simple query that can be easily answered by filtering only existing metadata (list of URLs and timestamp range) about the archived payload without needing to access the actual payload, referred hereafter as Type 1 workload.

Mallapragada also noticed that many homepages contain links named "about me" or advertising for international calling cards. Wouldn't it be useful to be able to tell the percentage of relevant homepages that have these features? In order to answer these questions using the Type 1 query, however, we will need to query against a "link" metadata field that provides information about all links going out from a webpage. This field does not already exist, therefore needs to be extracted from the HTML payload. Type 2 workload refers to the type of queries that only filter against existing metadata (in this case the same URLs and timestamps as above), but the ultimate goal is to be able to extract information from relevant payloads (in our case, extract links from the relevant Indian immigrants' homepages). No matter what kind of data processing is to be performed, the relevant records including their payloads must be loaded into memory first. For the sake of evaluating performance on equal ground, we throw away the information extraction portion of the Type 2 workload, and only consider the performance up till the relevant records have been loaded into memory. Type 2 workload is also useful for partitioning large archival data sets into manageable chunks. It is not quite realistic for a small research team with a shoestring budget to build a data processing system big enough to query 40 petabytes of archived data from the Internet Archive all at once. Using Type 2 queries, we may filter the most relevant resources out of much larger datasets, then output them as a new but smaller data set. In this particular case, we may output a new file from the memory containing only the relevant records, e.g., all Indian immigrants' homepages captured between the mid-1990s and the mid-2000s.

We next look at another example that has involved close collaboration with our study [39, 100, 109]. Dr. Florian Zach has a special interest in analyzing state-run tourism websites and how these websites evolve to attract more tourist activities. Dr. Zach requested a composed data collection in WARC format from the Internet Archive's inventory about the state websites. Extra data were also collected through customized crawlings to enrich the dataset. The main interest of the study focuses on the evolution of the web site content throughout the period of interest. Therefore, the study involves filtering and extracting the full content from the given collection to obtain specific content like travel tips, tourism site introductions, blog posts, and reviews.

The smaller data set resulting from the above partition may then be fed into the Type 3 workload, where no further record filtering is necessary, and the main focus is to manipulate if not all then a high percentage of the records in the data set. Typically all records are loaded into memory. Typical data processing includes tasks like parsing and field extraction, text mining, and topic modeling. [53]. In our case, the useful data manipulation involves link extraction.

There exists an important difference between Type 3 and Type 2 workloads inclusive of data processing. Type 2 workloads must load into memory data extracted from much larger archival files stored on the disk. This usually involves lots of iterative random disk access, which is much slower than a sequential disk read typically used in a Type 3 workload. In light of this, if the data extracted from a Type 2 workload is expected to be used repeatedly for various purposes, it is usually cheaper to first persist the extracted data on disk and then move the smaller data set elsewhere for further processing. In other words, even if we can afford to build a query system that can hold all 40 petabytes of data, it is still more efficient to use this system only to partition the larger data set instead of directly handling batch analytics. Therefore, the suitable cyberinfrastructure pattern to reuse web archives should be the "Bridge" [115] as shown in Figure 2.2, where archival files are stored separately from where the data are to be processed.

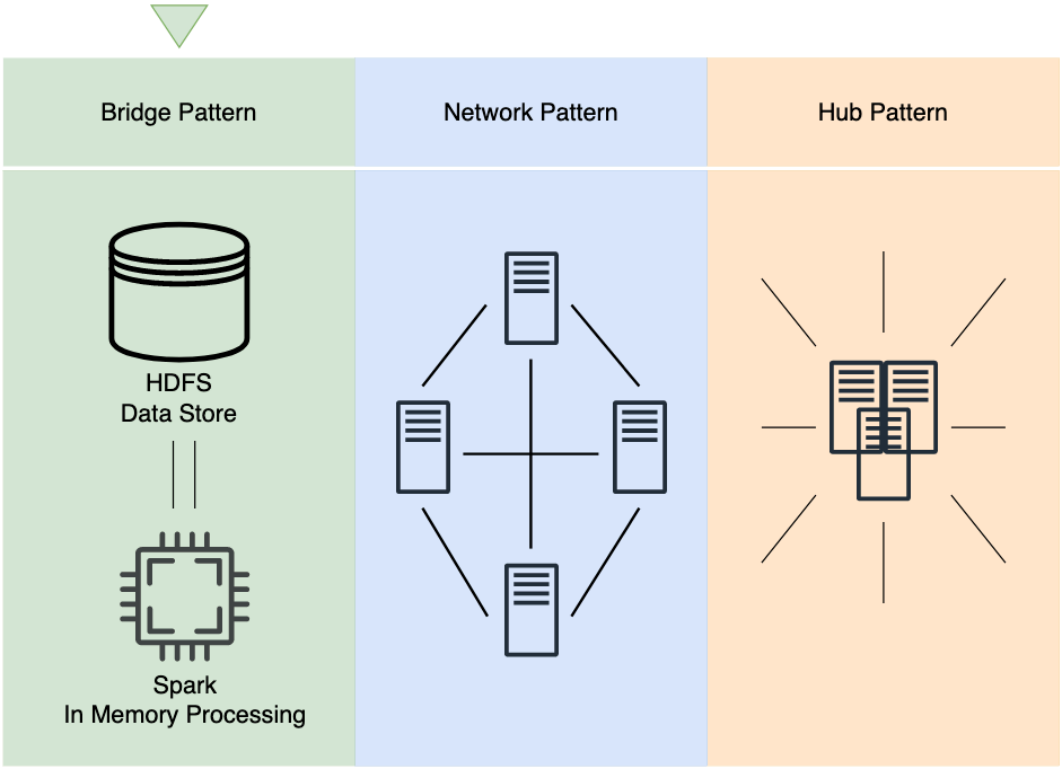


Figure 2.2: The three big data service patterns for libraries [115]. In our work, we follow the 'Bridge' pattern for web archive analytic workloads.

Under the "Bridge" pattern, however, we must assume we initiate a web archival analytics task from a "cold" system. That is, relevant data have not already been loaded into memory from disk for querying. As indicated above, the actual data loading can be a major performance bottleneck. For example, after all relevant data have been loaded into system memory, even if system A takes 1/10 time to perform the same analytics task as system B, we still cannot be sure system A outperforms B, because we do not know how much time

each system needs to load the data into memory. It is therefore critical to evaluate the full "Data-To-Insight" performance, which starts from the moment full archival data have been successfully copied to the disk of a "cold" (e.g., with empty cache) big data system, and ends at the moment useful information has been or is ready to be extracted from the memory.

Type	Workload
1	filter metadata without accessing payload
2	filter metadata with accessing payload
3	process all data without filtering

Table 2.1: Three types of workload

To summarize, this section outlines three types of workload, as shown in Table 2.1, that we believe will be representative of how researchers will be reusing web archives as big data sources. These workload types are in agreement with the filter, analyze, aggregate, and visualize, or FAAV web archive workflow previously identified [73], although our focus is more on characterising the underlying data manipulation patterns than their purposes and utilities.

2.4 WARC'S Performance Deficiencies

WARC is a text-based file format that linearly concatenates many web records. Each record is made up of a header and a content block. The header consists of multiple key-value pairs describing the record; the content block contains the payload data collected from the Web; both are encoded as text. Even non-textual payloads such as images, audio/video, and application media types are transcoded into flat US-ASCII text [47]. The WARC specification borrows its record-level structural and formatting conventions heavily from existing text-based web messaging protocols, especially RFC 2616 [44] and RFC 2822 [93]. Much of the messages exchanged during HTTP transactions may be directly inserted into WARC records without modification. It is therefore easier and cheaper to create WARC files while crawling the Web than parsing the HTTP responses and then writing into another format. However, the WARC format is not optimized for analytical workloads. In this section we characterize the root causes of its potential performance penalty. Quantitative analyses will be presented in the next section.

2.4.1 Data Structure: Linear Concatenation

Archival files are bulky and most likely have to be persisted on cheaper, slower "cold" storage media. They need to be brought into faster media, e.g., RAM, to be useful. Archival formats

determine how archived information is laid out on persistent storage. A naive data structure built using the linear concatenation method specified by WARC, will penalize the reuse performance, because it tends to be slower to traverse, retrieve, and load useful information from such a data structure into memory.

WARC stacks a large number of records consecutively without indexing. Reusing WARC files typically involves reading the whole file and parsing the records consecutively into programmable data structures such as RDDs [118] or Java/Python objects in RAM. WARC alone does not provide a mechanism to jump directly to a useful record without parsing all those before and after it. This changed when the CDX specification was introduced. A CDX file itself is another text file that linearly concatenates useful metadata, one line per record, in the hope that the metadata will help users to decide which record(s) will be useful. It also maintains the offset information for each record, which allows us to read individual records. With CDX files, many structured queries may be processed according to the following sequence: 1) parse the CDX file, 2) query the parsed information to decide which records are useful, 3) use the offsets to retrieve the useful records individually, and 4) load each record into memory and parse it, including parsing again the metadata already parsed in step 1. Even with CDX, there are still abundant opportunities to improve the analytical performance. There may be metadata duplication, where many useful records are closely aggregated in nearby blocks of storage. It is sometimes faster to first read these blocks in full into the memory and then perform the queries, rather than following the above 4 steps sequentially. However, caching, cost estimates, and query planning cannot be effectively leveraged if the access has to be either a full consecutive record scan (as in the case of WARC file only) or recursive random access (for WARC and CDX).

If we are to implement all these potential optimizations, however, the effort will largely be duplicating many years of work already implemented for databases. To leverage the prior work, all that is required from us is to adopt the data formats already specified by databases. Optimized for query performance, database file formats have leveraged many sophisticated data structures, ranging from ordered flat files, ISAM, heap files, and hash buckets, to B+ trees. Indeed, in the next section we will quantitatively evaluate analytics performance between data formatted in WARC and WARC/CDX with the same data set formatted in Parquet and Avro. The Parquet file format is designed to be used by columnar data stores while the Avro format is used by row stores. Both are open source formats, have many implementations, and already are widely used by big data systems, particularly Spark SQL.

Despite the performance penalty, WARC's data structure does carry an advantage over many others: it is very easy to append new records to existing ones. However, this advantage is only relevant for data collection. Files preserved for long term use are not expected to change over time; therefore easy appending is not an indispensable feature for an archival format.

2.4.2 Data Encoding: Text

Both WARC and CDX are encoded in pure text. This is a remnant of the upstream web messaging protocols. Still, it is worth questioning why a binary format, typically proven more efficient for both storage and processing than pure text, was not chosen for archiving. After all, web messaging protocols and archival formats serve drastically different purposes, so have different requirements. We can identify at least 3 major differences.

First, low interoperability and implementation overhead is the primary reason why web messaging protocols prefer text encoding. These protocols must be implemented by numerous software packages and systems. Each of them, acting as either a web client, server, or intermediary, is developed independently yet must be able to interact and coordinate seamlessly with others. Messaging protocols specifying more sophisticated encodings, e.g., CORBA [56] or RPC [82], tend to attract far fewer implementers, so do not achieve a critical mass, to be as successful as the Web. For web archives, however, interoperability is no longer essential since it only involves a small number of archives and their potential users. Like many other big data sources, a web archive can largely dictate the archival and/or dissemination formats as well as what is required for the potential users to access the data. Indeed, even though the support for WARC and/or CDX has been sparse for many years, they hold on as de facto web archival standard format, perhaps because the Internet Archive and many other national archives chose to preserve the Web in these formats.

Second, text encoding is a major performance concern for batch web archival analytics but not for web messaging protocols. This is because the data granularity of web messaging is typically tiny – one HTTP request or response at a time. Further, the processing load on these small messages is distributed to many machines: each web browser is only responsible for analyzing and presenting the HTTP responses it receives, while each server is only responsible for parsing HTTP requests destined for it and providing HTTP responses originated from it. Nevertheless, DDoS [40] attacks can still easily overwhelm web servers by wasting server resources on useless string parsing and database operations. In comparison, web archives disseminate data in fine granularity only under the close reading use case as illustrated by the WayBack Machine. Here, however, the WayBack Machine is responsible to disseminate responses for all archived URLs. Additionally, for batch web archival analytics workloads, we must be able to handle terabytes or even petabytes of data and all the required processing has to be shouldered by the users' analytical system. When batch processing textual files, we suspect that even tiny performance gains in either parsing, typecasting, or addressing, if multiplied by a large number, may result in significant speedups in end-to-end comparisons. This conjecture is corroborated by similar applications analyzing large JSON files or text logs. Two recent research studies estimated that they spend 80-90% of the execution time on parsing [71, 89]. In comparison, Freeman et al. [48] estimates less than 40-70% when analyzing time-series data with binary encoding.

To understand why text encoding penalizes performance, let us take a string from a WARC file, "WARC-Date: 2006-09-19T17:20:24Z", as an example. It cannot be directly used by

computers in a query, e.g., counting the number of records dated within a time range. We must first parse the whole WARC file to be able to separate one record from another and associate this string with a particular record. We also need to understand that this string is used to carry timestamp information, tokenize it to isolate the substring "2006-09-19T17:20:24Z", and then typecast the substring into an integer signifying the number of seconds from an epoch time. We also need to typecast timestamps from all records before the comparison can be performed. In contrast, if binary encoding is allowed in a database file, as is the case for both Parquet and Avro, an integer is already stored in place for all timestamps in a specific database column. We will not need to parse, tokenize, and typecast, although in some cases we still need to load the full database into memory.

Third, textual encoding is chosen by web messaging protocols also for its easy human readability. This is useful for developers of diverse systems to debug errors. In web archiving use cases, however, it almost never occurs that a user needs to directly read an archived file. Even close reading is mediated by systems like the WayBack Machine.

In summary, if analytical performance is an important factor to choose a web archival file format, maintaining text encoding as part of the format specification is rather unnecessary.

2.4.3 Data Addressing: Lack Rules and Constraints for Efficient Addressing

Closely related to text encoding, WARC's addressing mechanism is also rudimentary. Here, addressing refers to the way a computer system pinpoints a particular piece of useful information from a large archival file, for example, the timestamp of a particular record. As explained in Section 2.4.2, when the file is text-encoded, we will have to parse the whole file, be it a WARC or CDX file. This does not pose a significant performance problem for web protocols because the data granularity is typically small and each message is independent from the others. When browsing the Web, we never need to refer to the timestamp of a different HTTP transaction. Batch reusing of web archives, however, involves many such references within a large file, and parsing each of the whole files for every query is expensive. Database files provide an efficient mechanism for addressing through the use of schema. With a schema, archival metadata will be designated with various built-in or composite data types whose sizes are predetermined. For example, in Parquet all timestamps may be stored in INT64. The storage address of the nth record's timestamp can be easily computed without accessing any other field or record. Batch processing can therefore be more efficient by slashing unnecessary accesses.

2.4.4 Lost Opportunities for Combined Optimizations

When combining sophisticated data structure, encoding, and addressing capabilities, database files expose more opportunities for performance optimization. For example, Parquet may be queried so that large amounts of data accessing and processing are skipped, a mechanism known as Predicate Pushdown [17]. These opportunities are not available for WARC and/or CDX.

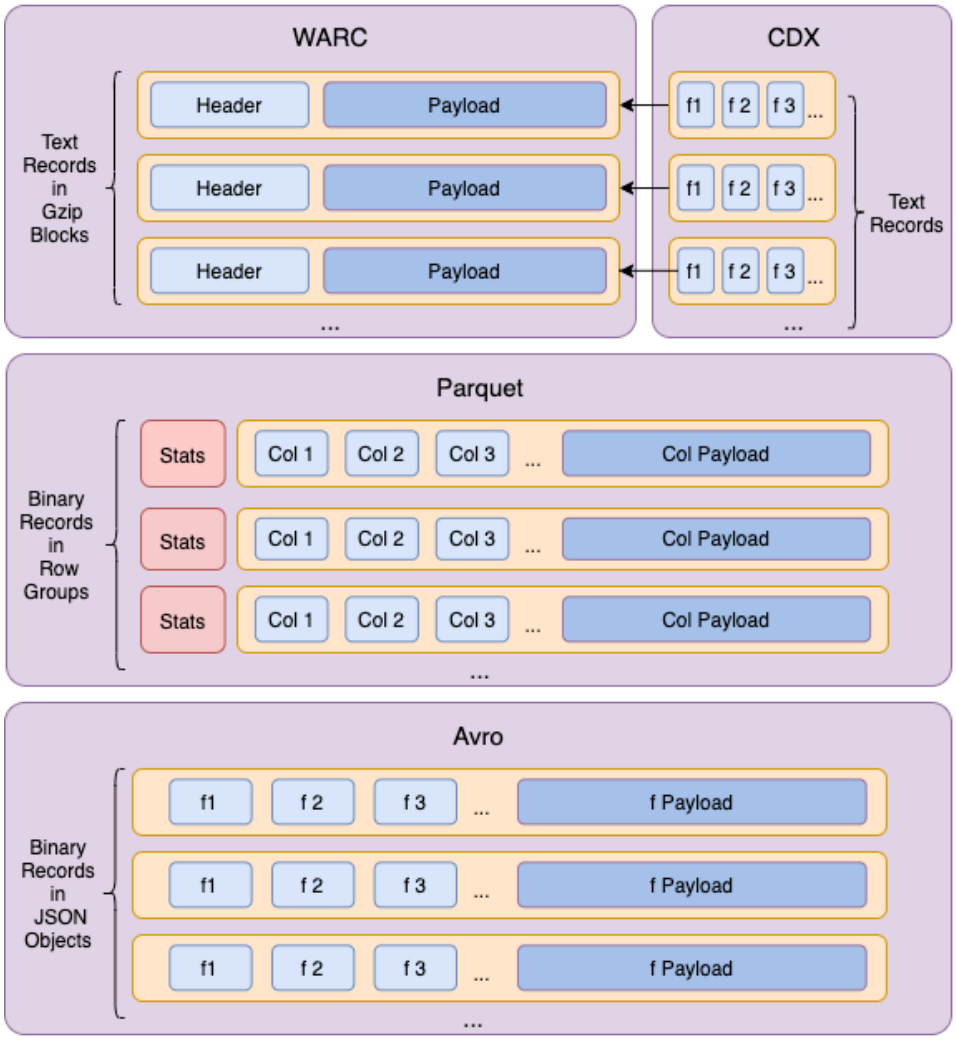


Figure 2.3: Comparing WARC-CDX, Parquet, and Avro formats

2.4.5 Alternative Data Formats and Comparisons

In this section we briefly introduce two alternative data formats, Parquet and Avro, and compare their features with WARC/CDX. Figure 2.3 schematically illustrates their data structure differences.

Both Parquet and Avro file formats are natively supported by Spark. By default, Spark will use the Spark SQL dataframe object to load Parquet or Avro data instead of traditional RDD. With dataframes, users can view the loaded data as a table in a relational database and perform SQL queries against it. Through Spark SQL's catalyst optimizer, SQL queries also achieve better performance compared with similar programming language operations against RDD. This is because Spark SQL can correctly leverage the performance optimization opportunities offered by Parquet and Avro. Figure 2.4 shows the different access mechanisms regarding how Spark would consume the web archive data in different formats. We have discussed that RDD is the fundamental processing unit that Spark uses for standardizing its internal computation object. This object requires user definition on its structure. Dataframe, on the other hand, is a new abstracted data object introduced by Spark. Dataframe objects are structured like a table in a relational database. Users could query dataframes with SQL-like queries. In addition, such queries are optimized to surpass the performance of the same implementations with RDD. In other words, data that can be directly consumed by dataframe could bring better performance on the processing stage than RDD.

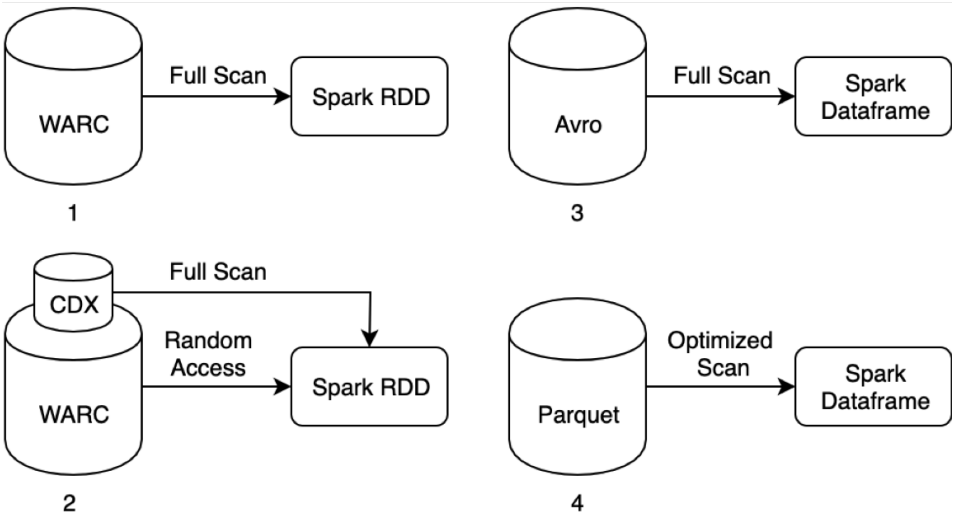


Figure 2.4: Different underlying web archive data structures and related Spark consumption methods

Parquet

Parquet is an open source columnar storage file format originally designed for the Hadoop filesystem by Cloudera and Twitter. It is optimized for fast querying, although writing Parquet files can be slow. This tradeoff is primarily attributed to its adoption of a sophisticated data structure. Following Google Dremel [79], Parquet rearranges nested data (e.g., XML and JSON) into multiple flat columns, each of which is made up of values of the same data type. For example, we can easily convert a CDX file into JSON without losing any information. We can then reformat the JSON into Parquet, which will store all timestamps into a single column of INT64 for easier and faster querying.

In addition to the WORM (write once read many) support, Parquet's columnar storage provides significant performance advantages for batch processing workloads:

- High compression. The unique data organization within Parquet can make the compression more efficient, which can reduce the data size and improve serialization performance inside the Spark framework. As shown in Section 2.5, we are able to achieve 0.9 compression ratio when compared with WARC, both with Gzip compression encoding. We also achieve 1.5x speedup on a full data scan compared with WARC as shown in Figure 2.7: Parquet (No PP) vs. WARC.

- Minimize data input with predicate pushdown. Data in Parquet is first divided into a number of row groups, and each row group is split into chunks of columns. Parquet then embeds extra information in the file, e.g., min/max stats and dictionary for column values in each row group. For queries like "count number of records with timestamp within a certain range", if the Parquet file is sorted by timestamp, the system can leverage the min/max stat of timestamp to filter out all irrelevant row groups. This is because the min/max stat of all row groups for the sorted timestamp column is continuous in a linear space. Querying systems can therefore skip loading these row groups altogether, a mechanism known as predicate pushdown.

- Minimize data input with column projection pushdown. Columns in Parquet are stored separately and only relevant columns will be accessed during the query. For example, the aforementioned time range query does not need to know anything about the payload. Therefore, like in WARC/CDX, payload data in Parquet do not need to be accessed. Furthermore, since the query does not concern any other columns except for "timestamp", all other metadata columns will not be accessed. This is known as projection pushdown. Figure 2.5 schematically illustrates the effects of both predicate pushdown and projection pushdown. In our evaluation, when using Spark SQL to query Parquet, the combination of these two pushdowns can achieve speedup of up to two orders of magnitude. Querying a Parquet file can also be around 2 times faster than querying CDX with the same data.

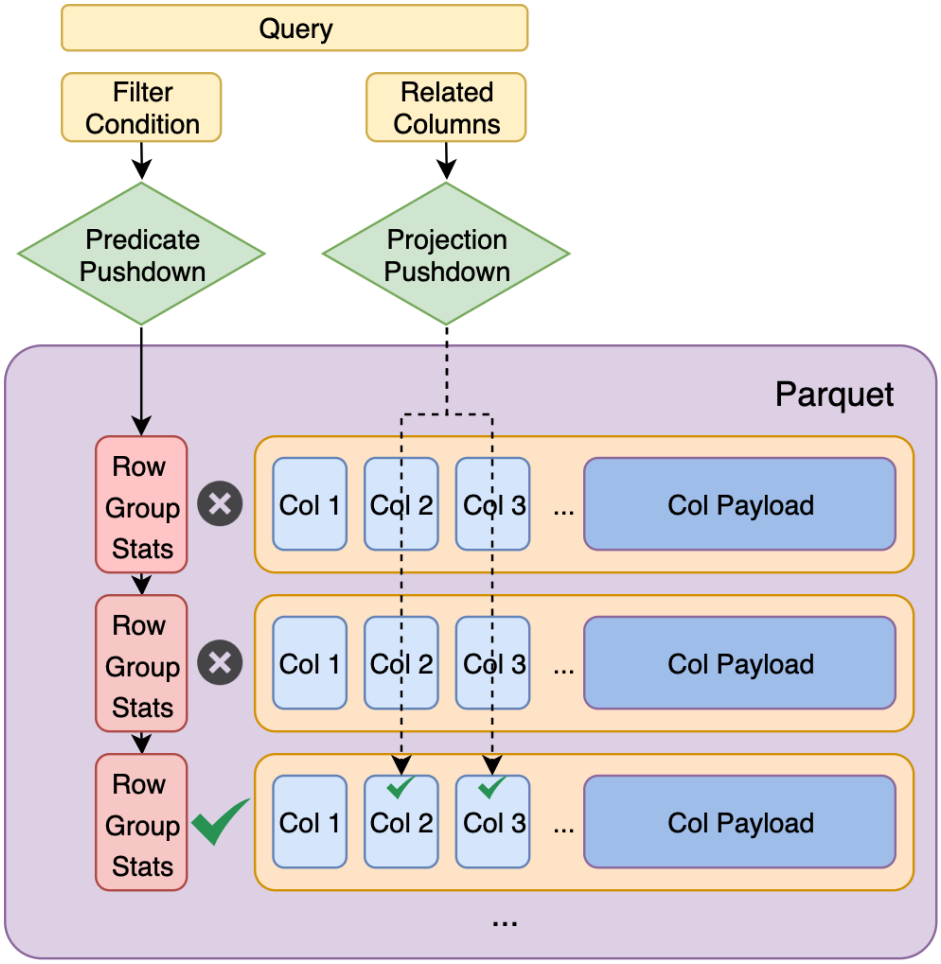


Figure 2.5: Using predicate pushdown and projection pushdown to skip reading unnecessary row groups or columns in Parquet data

Avro

Avro is another open source row-based file format, developed by the Hadoop project. Avro is optimized for data serialization and schema evolution. Similar to WARC/CDX, Avro is highly splittable and concatenatable: each record is preserved in one block with an associated sync marker (similar to offset), and the sync marker can be used for random access. Different from WARC, however, Avro adopts JSON data types and uses binary encoding. Data addressing in Avro depends heavily on the schema. Avro performs the best for batch web archival processing that needs to access lots of payload data. In full data scan, Avro can achieve 2X speedup compared with WARC and 1.6X compared with Parquet.

2.5 Evaluation

To benchmark the qualitative analysis in Section 2.4, we conduct controlled experiments. Because our main purpose is to evaluate the performance impact of the data format, we reformat the original WARC data set in different ways but they all contain the same information, albeit arranged differently. We then apply representative analytical workloads to them to trace the root cause of the performance differences.

2.5.1 Hardware/Software Setup

Experiments are performed on a shared-nothing cluster consisting of 1 master node and 6 compute nodes, inter-connected with an HP 2910al-48G gigabit Ethernet switch. Each node is a Supermicro 6027TR-DTRF server with 2 x 8-core Intel® Xeon® E5-2660 CPU, 64 Gb Memory, and 4 x 4 TB SATA hard disk. The cluster runs Cloudera Distribution Hadoop (CDH) 6.3.0, with Hadoop 3.0.0 and Spark 2.4.0. Data files are copied to HDFS. All experiments are executed in a Spark Read Evaluate Print Loop (REPL) environment through an Apache Zeppelin notebook. We use the following Spark configurations for all Spark jobs: yarn-client mode, 5 executor cores, and 30 Gb executor memory.

	WARC	WARC-CDX	Parquet (String TS)	Parquet (INT64 TS)	Avro
Size (TB)	0.985	0.998	0.917	0.914	1.321

Table 2.2: Data Size of different data types. "String TS" indicates the Parquet data with string type timestamp; "INT64 TS" indicates the Parquet data with INT64 type timestamp.

2.5.2 Data

We use Common Crawl's web archiving data crawled from May 20 to 23, 2018. The data set consists of 1219 Gzip compressed WARC files totaling 0.98 TB, and contains 53,324,440 records. The WARC files are organized by crawling time, each containing records crawled from a mutually exclusive time span.

We then reformat the WARC files to yield the following five datasets for comparison: 1) the original WARC files; 2) case 1 plus CDX index files built against all the original WARC files; 3) Parquet files containing the same information as case 1, with most columns in String type; 4) the same as case 3 but the Timestamp column in INT64 Timestamp type; 5) Avro, with the same column data types as in case 4 but the Timestamp column in INT96[1].

[1]The default timestamp format in Spark is in INT96 encoding. As of the time we conducted the experiments, the INT96 encoding had a bug within Spark development that prevents the predicate pushdown mechanism. So we force the encoding to be INT64 when we write the Parquet data through Spark. However,

We use a modified ArchiveUnleashedToolkit for reformatting WARC to Parquet and Avro. Settings for Parquet/Avro file writing are included in the example code repository on Github[2]. The reformatted Parquet and Avro data are Gzip compressed, the same as WARC files. Table 2.2 shows the data sizes for different data formats.

2.5.3 Workloads

We perform the following 6 analytical tasks, with Task 1 to 5 run against the 5 datasets specified in Section 2.5.2, and Task 6 run against 5 filtered datasets created from the previous 5 in the same formats. The filtered dataset is about 5 GB when formatted in WARC.

- Task 1: Count the number of records.
- Task 2: Retrieve metadata from records filtered by a given time range.
- Task 3: Retrieve metadata from records filtered by a list of URLs.
- Task 4: Retrieve full records filtered by a given time range.
- Task 5: Retrieve full records filtered by a list of URLs.
- Task 6: Retrieve full records without filtration, extract pure text from all payloads, and run topic modeling over the extracted text records.

Queries and example codes used in the experiments can be found on Github[2].

Tasks 1 to 3 are Type 1 workloads that only need metadata to complete. Tasks 4 and 5 are Type 2 workloads that will load all filtered records into the memory. Task 6 is a Type 3 workload that will load all records in the dataset into the memory.

For Tasks 2 to 5, we also use different time ranges or URL lists to yield different levels of selectivity from the dataset. A selectivity of 0.001 means the query has yielded 0.1% of the total number of records. When we vary the time ranges on Task 2, however, the results do not change with selectivity. We therefore do not plot the results against selectivity.

For Type 1 workloads (Tasks 1 to 3), the query results only include metadata. For Type 2 workloads (Tasks 4 and 5), the query results include the full record including payload content.

The Type 3 workload (Task 6) involves retrieving the payload content from the collection, extracting pure textual content from the payload, and applying the topic modeling algorithm

we do not find an option in Avro writing that can change the timestamp encoding. Thus we kept the INT96 encoding with Avro conversion. The only difference between INT96 and INT64 is that INT96 includes a nanosecond representation in addition to INT64.

[2] https://github.com/xw0078/WebArchiveWithParquetAvro

to derived text content. We use the Latent Dirichlet Allocation (LDA) algorithm from SparkMLib [80] for topic modeling. In our experiments, we use a subset collection of about 5GB to represent a derived web collection. This subset collection is a simple selection of certain files from the whole collection. LDA is a computational intensive and time consuming process when applied to a large dataset. With 5GB size, we can control the max time for the experiment, so it is limited to be in hours.

2.5.4 Results and Analysis

As shown in Table 2.3, Table 2.4, and Figure 2.6, for all Type 1 workloads, Parquet performs the best. This may be attributed to the columnar data structure that allows Spark SQL to skip loading irrelevant columns, which significantly reduces the disk I/O. In Task 2, the difference between Parquet with no predicate pushdown (No-PP), where Timestamp is encoded in String, and Parquet, where Timestamp is encoded in INT64, illustrates the performance gain from the binary encoding as well as Predicate Pushdown. Parquet and WARC-CDX performance are significantly better than Avro and WARC. This is because when queries only concern metadata, only the CDX file and the relevant column(s) in Parquet will be accessed. In contrast, all data contained in WARC and Avro will need to be loaded into memory.

	WARC	WARC-CDX	Avro	Parquet
Average Runtime (sec)	2891	33	1245	15

Table 2.3: Task 1: Count the number of records. The timestamp field type is irrelevant to this task; we use the Parquet data with string type timestamp for this experiment.

	WARC	WARC-CDX	Avro	Parquet	Parquet (No-PP)
Max	3366	36.70	1493	17.07	40.14
Med	3357	35.28	1450	15.86	35.43
Min	3354	34.51	1427	15.49	31.59

Table 2.4: Task 2: Retrieve metadata from records filtered by a given time range (runtime in seconds). 'Parquet (No-PP)' here represents the Parquet data with string type timestamp field, where the predicate pushdown is not enabled (No-PP).

As shown in Figures 2.7 and 2.8, Parquet still demonstrates strong performance in Type 2 workloads. At low selectivity, the effects of Predicate and Projection Pushdown are more evident, allowing it to achieve about 2 orders of magnitude of performance gain against WARC. WARC-CDX is only faster when the selectivity is very low. Since Type 2 workloads need to load payloads from many records into the memory yet WARC-CDX uses recursive random disk access, its performance quickly degrades and will continue to worsen as selectivity goes higher. Figure 2.7 shows that when the selectivity is above 25%, WARC-CDX's

performance is worse than reading all WARC files in full. Using the CDX index becomes counterproductive. In contrast, the Parquet performance degradation converges to Parquet (No-PP).

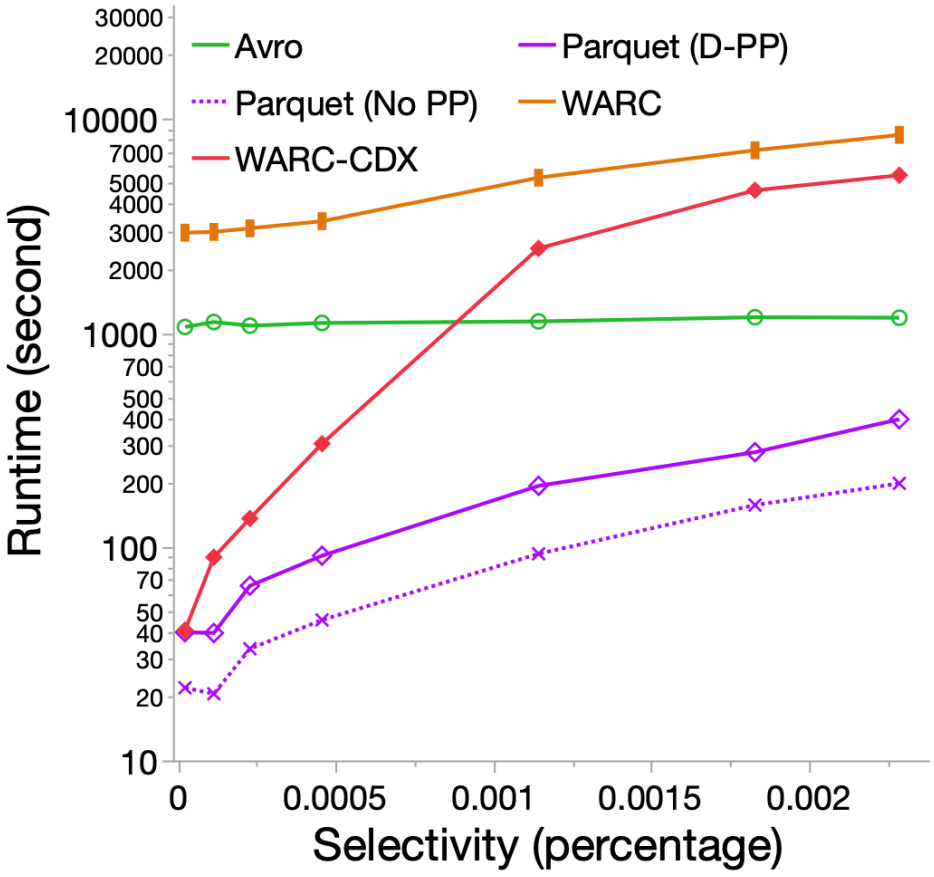


Figure 2.6: Task 3: Retrieve metadata from records filtered by a list of URLs. 'Parquet (D-PP)' and 'Parquet (No-PP)' here represent the Parquet data with string type timestamp field. The timestamp is irrelevant to this task. 'D-PP' here indicates that we use the domain first filtration as a one step optimization before full URL matching with predicates pushed down to the domain level. 'No-PP' here represents full URL matching without predicate pushdown.

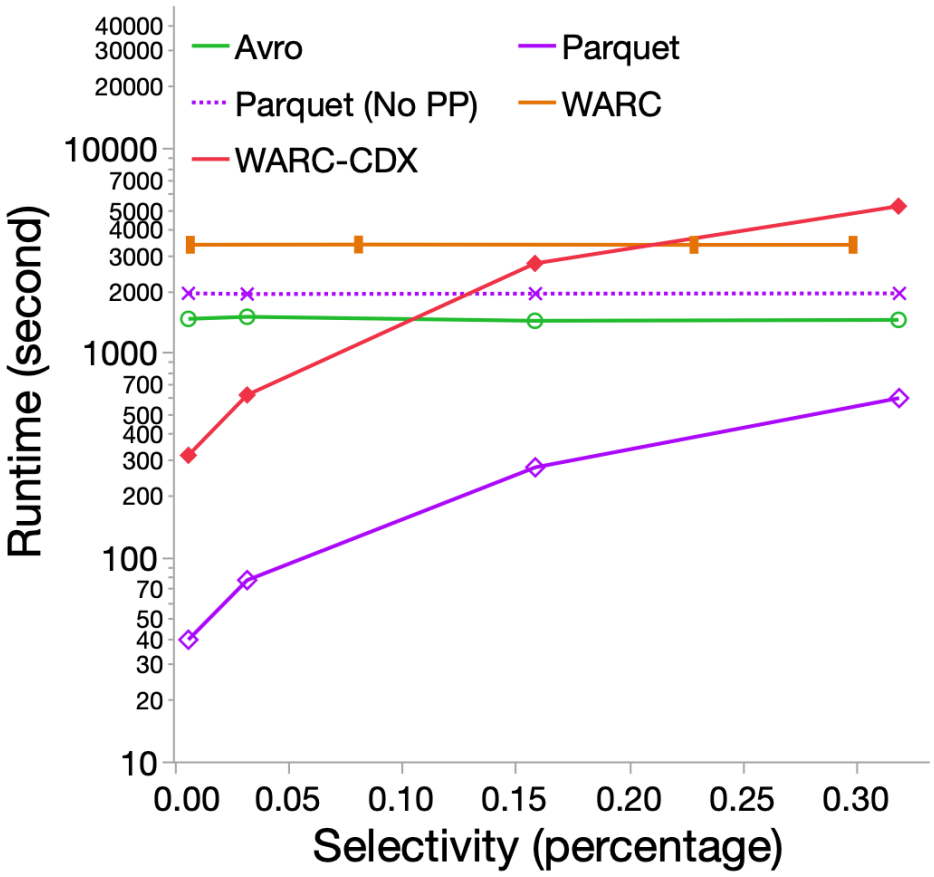


Figure 2.7: Task 4: Retrieve full records filtered by a given time range. 'Parquet (No-PP)' here represents the Parquet data with string type timestamp field with no predicate pushdown enabled. 'Parquet' represents the Parquet data with INT64 timestamp field with predicate pushdown enabled.

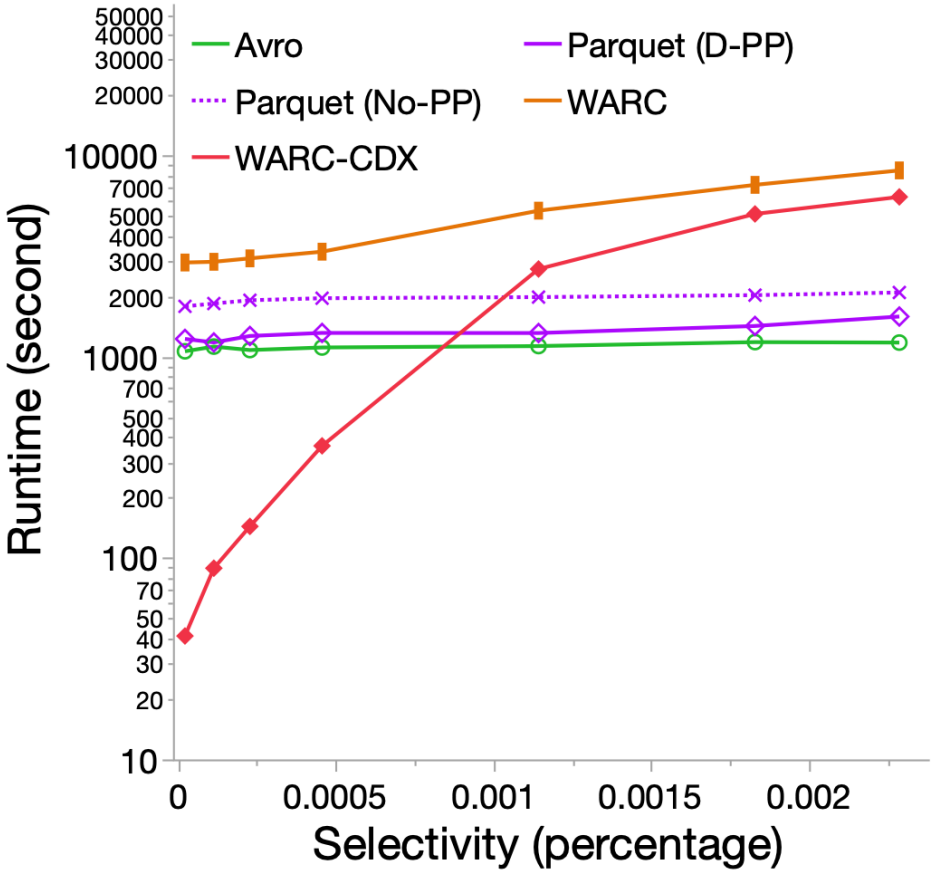


Figure 2.8: Task 5: Retrieve full records filtered by a list of URLs. 'Parquet (D-PP)' and 'Parquet (No-PP)' here represent the Parquet data with string type timestamp field. The timestamp is irrelevant to this task. D-PP here indicates that we use the domain first filtration over URL searching as a one step optimization with predicates pushed down to the domain level. No-PP here indicates processing without predicate pushdown.

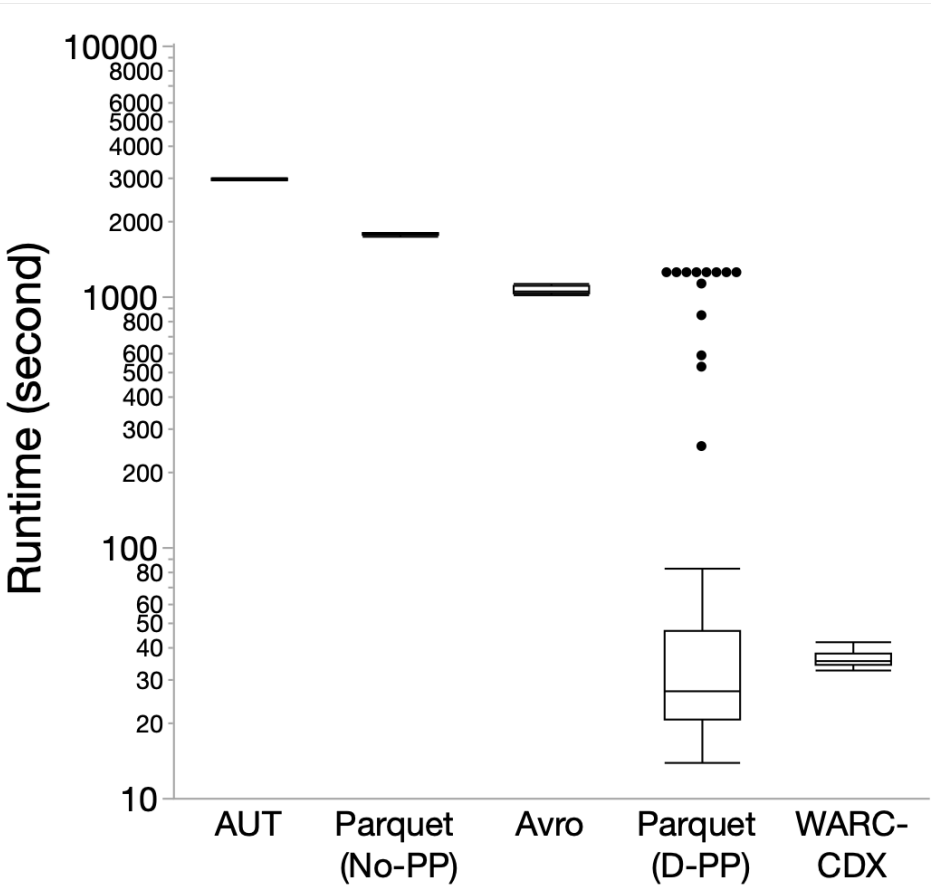


Figure 2.9: Task 5 with a single URL. 'Parquet (D-PP)' and 'Parquet (No-PP)' here represent the Parquet data with string type timestamp field. The timestamp is irrelevant to this task. D-PP here indicate that we use the domain first filtration over URL searching as a one step optimization with predicates pushed down to the domain level. No-PP here indicates processing without predicate pushdown.

	WARC	WARC-CDX	Avro	Parquet
Average Runtime (sec)	2115	5450	466	484

Table 2.5: Task 6: Topic modeling. The timestamp field type is irrelevant to this task; we use the Parquet data with string type timestamp for this experiment.

2.5.5 Transactional Workload

While our experiments clearly demonstrate the WARC-CDX case's performance degradation for batch data processing, we are also interested in gauging its advantage on small, transactional queries. Figure 2.8 shows the result of Task 5 filtered against a single URL. This is similar to the workload of the WayBack Machine. On average, Parquet with Predicate Pushdown on domain name (D-PP) is actually faster than WARC-CDX. However, in rare cases, Predicate Pushdown does not work very well. In these cases, the different captures of a single URL may have been evenly distributed in many row groups, forcing the application to read all these row groups, thus leading to performance degradation.

Another point to note is that the Parquet file we created sorts the row groups by Timestamp. Predicate Pushdown therefore works better on Timestamp filtering than URL filtering, as shown from Figures 2.7 and 2.8. If URL filtering performance is more important, we can always re-sort the row groups differently, e.g., based on domain name, although there is a limit on how much we can optimize for all types of queries. Nevertheless, as Figure 2.6 clearly illustrates, even sub-optimal performance from Parquet can be comparable to or outperform WARC-CDX on transactional workloads.

2.6 Conclusions and Discussion

As we point out in Section 2.4, WARC's resemblance to formats involved in web messaging allows crawlers to defer many data loading, parsing, conversion, and validation workloads during the data collection stage. However, if we choose to also use WARC as the archival format and eliminate such processing altogether, the end users of the archive will have to shoulder an extra processing burden at the reuse stage. Our evaluations provide evidence that WARC carries significant performance penalties for batch data processing workloads, up to two orders of magnitude slower than that of more efficient formats. We therefore call for the web archiving community to consider adopting alternative archival formats.

On the other hand, our recommendation does not necessarily imply the end of life for WARC. In terms defined by the OAIS reference model [28], the WARC format remains valuable when collecting data to build Submission Information Packages (SIPs), especially when resources are limited during the crawling. But more efficient formats should be considered for Archival Information Packages (AIPs). The format conversion from SIPs to AIPs may be carried out offline when computing and storage resources are no longer a bottleneck. Indeed this is already happening. A number of WARC derivative formats have been proposed to store post-processing results. For example, CDX/CDXJ [31] are plain text formats used to store HTTP headers [29]; WAT is a JSON based format used to store WARC metadata [50]; WET is again a plain text format used to store extracted textual elements [81]. More recently, Common Crawl even directly used Parquet to store metadata extracted from WARC

[85]. The weakness of this approach, however, is its dependency on an inefficient storage format. Because ArchiveSpark [59] is the only published example we are aware of that leverages a derivative format in batch web archive analysis, we use it as a surrogate to evaluate the performance impact of WARC derivatives. The results show the performance is only comparable to more efficient formats for transactional workloads. For Type 2 and 3 batch workloads, because WARC derivative formats can only store record-level offsets instead of directly managing the data storage, the performance is throttled by the addressing mechanism that has been demonstrated by our evaluations to be inefficient. We anticipate this trend to hold true even if the metadata are stored in Parquet as in [85]. Even for Type 1 batch workloads where the absolute difference between querying a pure text metadata file and querying a database file is not as pronounced, the 2X to 3X relative difference can still be significant, given the growth of web archives. It is therefore not sustainable to continue storing metadata in pure text formats, as is the case for CDX, WAT, and WET.

As with many other discussions around data formats, arguments can be made to emphasize the simplicity, flexibility, and durability of a text-based format such as WARC and its derivatives against a binary format. However, many such arguments need to be put in the context of big data and its usage. Not only does human readability factor far less significantly than machine efficiency with the data growth in size, many of its perceived advantages have also been rendered irrelevant. For example, almost all archived WARC files we encountered have already been compressed and converted into binaries in the archival storage. The notion that a human-readable text is more durable because it can sustain benign neglect for an extended period of time, in our view, lacks quantitative support. Qualitatively, contrary arguments can also be made that free-form natural languages appear to be harder to decode and process than machine languages, whose semantics and syntax are mechanical and thus simpler. Specifically, a database format can be decoded with the associated schema, and the schema can be extended to include unexpected elements in the future.

Moreover, neither Parquet nor Avro depends on the existence of HDFS and/or Spark to function as an archival format. They simply offer more optimized layout and hooks to leverage data sharding, replication, and parallel processing provided by these systems. Even without taking full advantages of these optimizations, the performance is not necessarily worse off than analyzing flat textual files, which offer little opportunity to speed up queries. The support for open database formats can also be more available and sustainable than that of a niche text format. Moreover, the loss of abandoning many years of work around WARC can in turn be compensated by leveraging a much larger body of established work on databases, distributed systems, and parallel processing. The performance gain will open the growing volume of web archives for more productive and effective reuse.

Our recommendation is also in line with academic and industrial trends in big data processing. While flat text log formats continue to be used to collect data, and JSON is used by APIs to disseminate data, they have long been considered a major bottleneck for data warehousing and analytics. Much work has been done and new techniques developed to speed up the parsing and loading of log files and JSON, as well as to convert them into

more efficient formats for batch processing. A recent example from Uber [101] describes how the company incrementally converts its JSON source data into Parquet and Avro in a 100+ Petabyte data lake with minute level query latency. In order to achieve similar performance for a web archive, we think it is necessary to move away from using WARC as the archival format.

This chapter is supplemented by Appendix A, which provides more implementation details for the accompanying experiments.

Chapter 3

Hosting Large Web Archive Collections on Raw Data with Parquet Format

In this chapter, we address our second research question: *Can we provide accessibility of large web archive collections for browsing efficiently under Question 1?* Chapter 2 has established a quantitative analysis computational infrastructure for web archive collections with a new file format Parquet and computational frameworks including Hadoop and Spark. This chapter proposes a web archive hosting infrastructure for browsing above Chapter 2. As shown in Figure 3.1, we propose using direct access between Spark and Parquet formatted data to provide instant access to web archive records that can be delivered to web servers efficiently for web browsing. We also leverage high-performance index storage with HBase to provide rapid web archive record navigation through URL searching. We evaluate the infrastructure through standard load testing workloads for web servers to benchmark the performance of our demonstration websites.

Web browsing is considered as a classic use case in online transactional processing (OLTP) systems. However, the web archive use case does not aim to provide a service with optimal throughput and low latency. Instead, we consider the web browsing of a web archive as a supplemental tool with other analytical processing. Thus, we still include this component under the OLAP pattern.

3.1 Introduction

Web history is now an important part of the record about human history. However, the nature of web development is primarily focusing on information dissemination with the latest content. Few web creators expend effort to preserve their work, so such memories can completely vanish from the internet. To combat the loss of our web history, web archiving practices are getting more attention worldwide. Many web archiving initiatives around the world have already taken action and collected a vast amount of web data [33, 52] since the 1990s. One of the earliest and most prominent web archive initiatives, the Internet Archive (IA), had already captured more than five hundred billion web pages across the Internet [16]

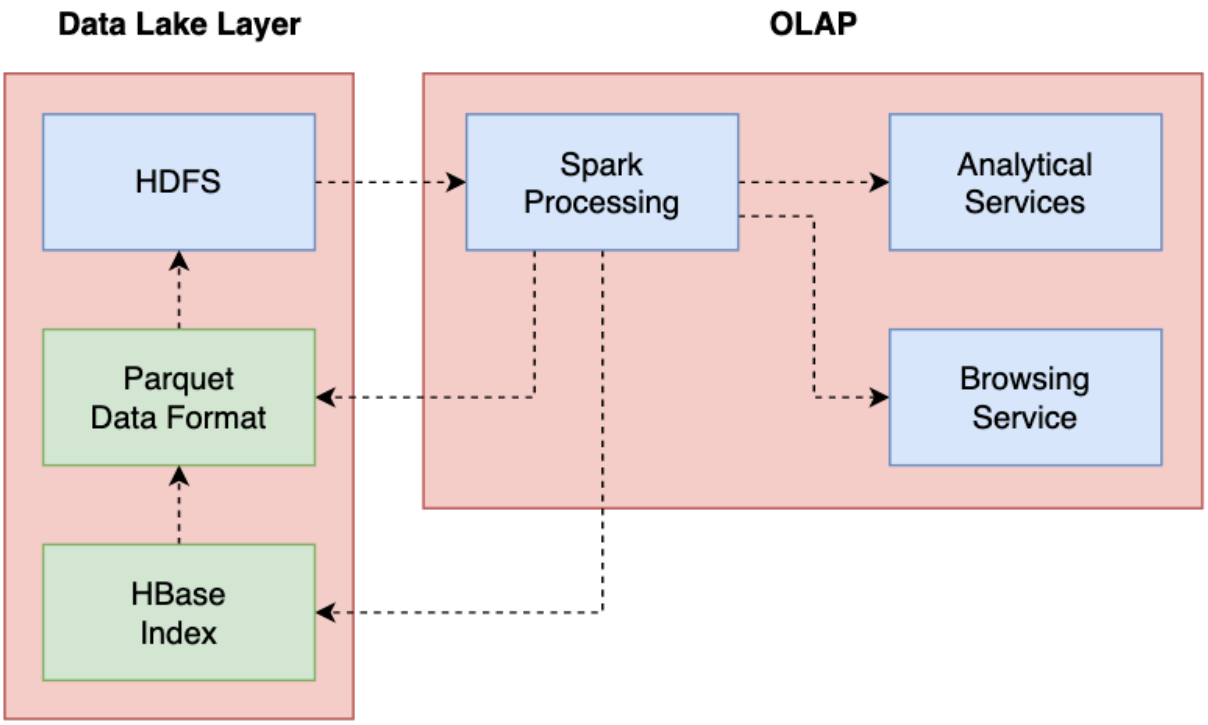


Figure 3.1: Data flow in the proposed system design

as of 2021. Though large organizations like IA have preserved a vast amount of information, it only composes a tip of the iceberg from the whole Internet. Most of the web archive initiatives are smaller organizations like libraries and universities. There are abundant web archive collections that are collected from such organizations. However, such organizations usually can not provide equivalent content delivery systems relative to the Internet Archive. This is because web archiving activities can involve high computational requirements to collect, store, and serve the information at scale. The computational resources and technology supports for smaller organizations are far from those of big web content providers like Twitter or Facebook.

The web archive content generally has been collected through web crawling activities. The community has standardized the web archive data preservation format as WARC [62] format to expand the web archiving efforts. Since the web archive data is usually not used for immediate consumption, the standard web archival format is designed to be in an aggregated form with plain text. Such a data format design makes the web archive collecting and sharing processes more convenient. However, in addition to data preservation purposes, web archive reuse is now playing an important role as well. The vast amount of knowledge preserved in web archive data has attracted audiences to explore and analyze web history more deeply. The most common way to access web archives for audiences is through directly browsing the web archive records in a web browser. Large web archive content providers like the

Internet Archive have built an access service, the WayBack Machine, to host the majority of their preserved content and make that available for users to browse individual records. Similar services are also provided elsewhere [3, 30, 114]. Such services usually leverage web archive index files like CDX [29] to improve access to WARC content. However, many web archiving initiatives lack technical support to construct such a service as the WayBack Machine. Therefore, the Internet Archive has developed a service called Archive-It to provide an automatic service for web archive collecting, storing, and serving with a version of the WayBack Machine. Users of Archive-It can create their own web collections, which would capture unique parts of the web, preserving memories and sharing them through this service.

In addition to web archive browsing, quantitative data analysis is also popular around researchers who desire to mine the web archive data to provide insights. As an example, large web archive collections have contributed to phenomenal artificial intelligence developments like BERT and GPT computational language models [24, 37]. Such large scale data reuse cases usually involve collections like those from Common Crawl, a web archive initiative that provides large-scale web archive datasets in raw format. The web archive data from Common Crawl is hosted on the Amazon Web Service (AWS), where audiences can download the data freely. Host web archive data in raw format is also popular among small web archive initiatives, because this is a low-cost method to deliver the data to audiences. The Internet Archive also provides large datasets upon request. However, such requests sometimes require extra cost when IA needs to construct the dataset from its enormous collection.

In academia, both browsing and quantitative usage are essential tools for the researcher to conduct data analysis. However, there are limited opportunities for such audiences to have the ability to access both of these services over large web archive collections at the same time. This is because the general computer system architectures for web archive browsing and quantitative analysis are quite different, making it challenging to combine both services in one playground. Browsing services usually involve a traditional web hosting architecture where optimized database systems play a significant role in providing high-performance service. Though the WayBack Machine builds its service over the raw WARC data, the full picture of its architecture remains unknown. Quantitative usage, on the other side, usually involves data manipulation of raw data in a batch mode to enable efficient processing.

If we can build an architecture that can facilitate both browsing and quantitative reuse services, researchers will be empowered to derive valuable insights from web archive collections more efficiently than with the existing research model. To address such challenges, we propose an architecture to host large web archive collections for browsing on raw data with the Parquet format. This architecture is built on top of a quantitative architecture for web archive data. The Parquet [11] data format was invented for large scale data warehouses, facilitating various processing needs over large amounts of data. The nature of Parquet formatted data provides a nice data slicing in its internal structure through row groups. We can leverage the row group level data ingestion mechanism to provide instant record access with minimal scanning in the raw Parquet web archive data. In Chapter 2, we have demonstrated the advantages of using Parquet formatted web archive data in the quanti-

tative usage scenarios in a Hadoop-based distributed computational environment [112]. To advance further, we expand the usage of Parquet formatted web archive data to provide a browsing service under the Hadoop environment and test its performance. We leverage the efficient direct access from the Spark processing framework and Parquet formatted data to provide consistent performance on raw data scan and delivery for web servers. We also take advantage of the high-performance HBase database system to boost the performance of URL searching, enabling efficient web archive records navigation.

We benchmark our proposed architecture with standard load testing for web sites. The benchmark results show that our proposed architecture can maintain a consistent and reasonable performance for web page hosting while incurring a low cost for the computational resources. Our study introduces an opportunity for web archive initiatives to leverage large web archive collections for both browsing services and quantitative analysis works, at scale. We believe such an architecture design philosophy can be adopted by the web archive community and expand the content delivery services for web archive audiences in a broader way.

3.2 Web Archive Hosting Services

Reading and analyzing web information over time can offer valuable information. Thus, access to individual web archive records is an essential part of web archive-related services to deliver content to audiences. The Internet Archive has established an astonishing amount of web archives covering a wide range of websites where users can find and browse their desired web archive page with a response time in seconds. Not limited to its own interests, Internet Archive also offers the opportunity for other web archive initiatives to collect and serve data of interest through their established technology through the Archive-It service. Other than the WayBack Machine and Archive-It, web archive hosting services with a similar scale can hardly be found on the Internet.

However, tremendously valuable web archive content also is being collected by individuals and institutions around the world, and such web archive records might not have a proper infrastructure to deliver the content to audiences. Many smaller web archive initiatives only store the raw web archive data in a local system and share it through raw data files, where users could download the data and use them locally. Such a web archive service model requires a high level of computational skill from the audiences to reach the insights from the web archive data. The absence of a browsing capability for many web archive collections also prevents audiences from exploring the data efficiently. Why is a suitable browsing service not generally adopted within the community? One primary concern is that such a service requires dedicated computational resources, which leads to a high cost for building such a service. Since a web archive collecting service has already consumed significant computing resources, many institutions do not have extra resources to build other services around the data. In addition, there is no architectural standard for web archive hosting

services. Such architectures can involve complicated computational system design, causing a technology barrier for service providers. For convenience, the web archive data can certainly be consumed by conventional web hosting systems. However, such systems are designed for high performance web hosting under a high load scenario with a relatively small number of web pages. On the one hand, web archive browsing activities generally have a much lower access frequency, but on the other hand, the scale of the underlying data size could be very much larger than for common web sites. However, dedicated web hosting systems for web archives would usually not be worth the cost to maintain. Fortunately, open-source software like Pywb enables web archive hosting in a convenient one-stop environment; it addresses the limitation on providing scalable services with a large amount of data. Unfortunately, the standard WARC data storage introduces an extra burden for services to consume and deliver the data, because the format is designed for long-term preservation instead of analysis/access efficiency and flexibility. Though we can gain some hints from the design of the WayBack Machine, there are insufficient details available to replicate such a service for a broader adoption.

In Chapter 2, we observed multiple benefits from the Parquet data format in various web archive workloads. We think the same concept can be applied to a web archive data browsing service. We will illustrate our proposed infrastructure for web archive browsing based upon Parquet formatted data in later sections. In the next section, we will examine the related works around web archive browsing services, low-cost infrastructure with Parquet formatted data, the Apache Spark framework, and HBase storage.

3.3 Related Work

In this section, we introduce related works around the web archive hosting services and help justify the need for an infrastructure design as we are proposing in this work.

The WayBack Machine is currently the largest web archive content provider for audiences to view web archive records over time. According to the Internet Archive's developer sam and raj [98], the core infrastructure of the WayBack Machine works with a distributed computation environment where it utilizes a highly efficient index store based on Redis [27], and WARC format based web archive data files in the file system. Nevertheless, details of the WayBack Machine's infrastructure remain unknown, making it hard to learn and replicate the architectural design. On top of the WayBack Machine, the Internet Archive also provides the Archive-It service for many web archive initiatives to collect and deliver unique web archive content through their existing technologies. The initiatives include large organizations like the EU Web Archive, Library and Archives Canada, and many university libraries [52]. Though centralized services like the WayBack Machine and Archive-It have been delivering a large amount of high-quality web information, many other web archive collections are collected and scattered around smaller organizations in the web archive community, and lack a content delivery system. Users may also seek paid web archive services for convenient

data access like Stillo [104], Pagefreezer [88], or even free lancers [45].

Some developers and researchers have worked to empower the community with the ability to deliver web archive content independently. Pywb [114] is open-source software partially meeting the need for high-fidelity web archive replay services, where users can host their local web archive collections in WARC format with an integrated web server from the package. Audiences could have a similar experience to the WayBack Machine from Pywb. However, unlike the WayBack Machine, Pywb is not designed to be scalable; it is hard to make it host large web archive collections. Alam et al. introduces the InterPlanetary WayBack (IPWB) as an effort toward building a web archive replay service in a decentralized fashion [3]. IPWB takes advantage of the InterPlanetary File System (IPFS) [18], a peer-to-peer (P2P) distributed file system, as the storage layer for hosting WARC formatted web archive collections. From such an infrastructure, a user could access any web archive collections in the IPFS network. The author of IPFS, Benet, envisions that IPFS can enable a web environment that can permanently preserve the historical information. However, the success of the data preservation effort from such a decentralized mechanism heavily depends on user participation. If a publisher stores some data on IPFS that is rarely accessed, like web archives, the data could still be lost if the publisher loses the data on their end[18].

Besides the work that is directly connected with web archive hosting services, several works have been focusing on the data access and analysis of web archive data in a scalable way. Lin et al. introduces Warcbase [73], an infrastructure design for big web archive collections, to aid content access and analysis through separate big data tools including HBase and Spark. As a component for high performance content delivery, Warcbase proposed to use HBase as one source for web archive applications that require high query performance. In this case, the web archive data need to be persistently stored in HBase to provide related service as in the "network" pattern [115] shown in Figure 2.2. Though the system could gain performance advantages from the database technology, such an infrastructure introduces an extra burden on the data storage, with duplication of data. This is because the service system is not suitable for long-term preservation. The successor of Warcbase, ArchiveUnleashed-Toolkit (AUT) [95], abandons the HBase part and focuses on an analytical infrastructure with Apache Spark and WARC-based web archive data. This indicates that the storage concern is an important factor with large web archive collections in the community. Holzmann et al. implement a web archive framework with Apache Spark, ArchiveSpark, that can efficiently leverage the CDX index data with WARC files. This work demonstrates an efficient data access and analysis model with selective content through index filtering [59]. This is also the major difference between ArchiveSpark and AUT.

There is a middle way where web archive researchers can leverage the power of quantitative and instant access at the same time. In this work, we want to fill in the missing parts of the puzzle where users can gain the capability to browse web archive records from the same analytical system that also ensures high performance on quantitative analysis.

3.3.1 Access Web Archive for Browsing

In this work, we built a demonstration web archive hosting service to show and benchmark the performance of our proposed infrastructure. The first step when a user wants to explore the web archive sample is to find the link or URL for the desired content. This step typically involves searching for a URL address. Figure 3.2 shows the demonstration web page where a user can search a URL from the search box to find web archive records from different timestamps. After searching, a user will then receive a list of candidates captured from

Figure 3.2: Demonstration web page for searching a URL from the web archive collection

different times. Each record represents an identical copy of this URL at a given time when the page was archived. Figure 3.3 shows the demonstration results returned for the searched URL. Finally, a user can click the desired archive version from the returned list, to access

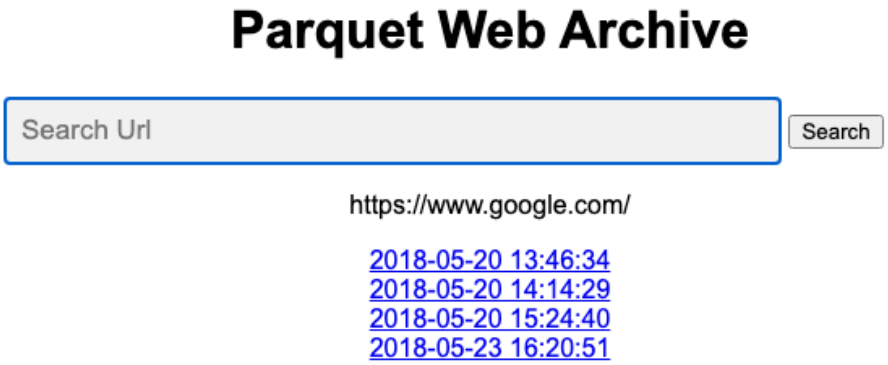


Figure 3.3: Demonstration web page for the response from a searched URL. In this example, we searched "https://www.google.com/" and the timestamp list below shows all the matching records in our collection.

the actual web page for browsing.

The system first needs to find all matching records for a searched URL from the web archive collection from the server-side. Then, for each web page's complete content request, the system must extract the entire content (payload) from the web archive collection. The challenge is that we need an architectural design with less operational cost than a traditional

web hosting infrastructure on top of the raw web archive data. The following section demonstrates our proposed web hosting infrastructure design, where the system operates on raw Parquet formatted data to provide consistent web archive record access.

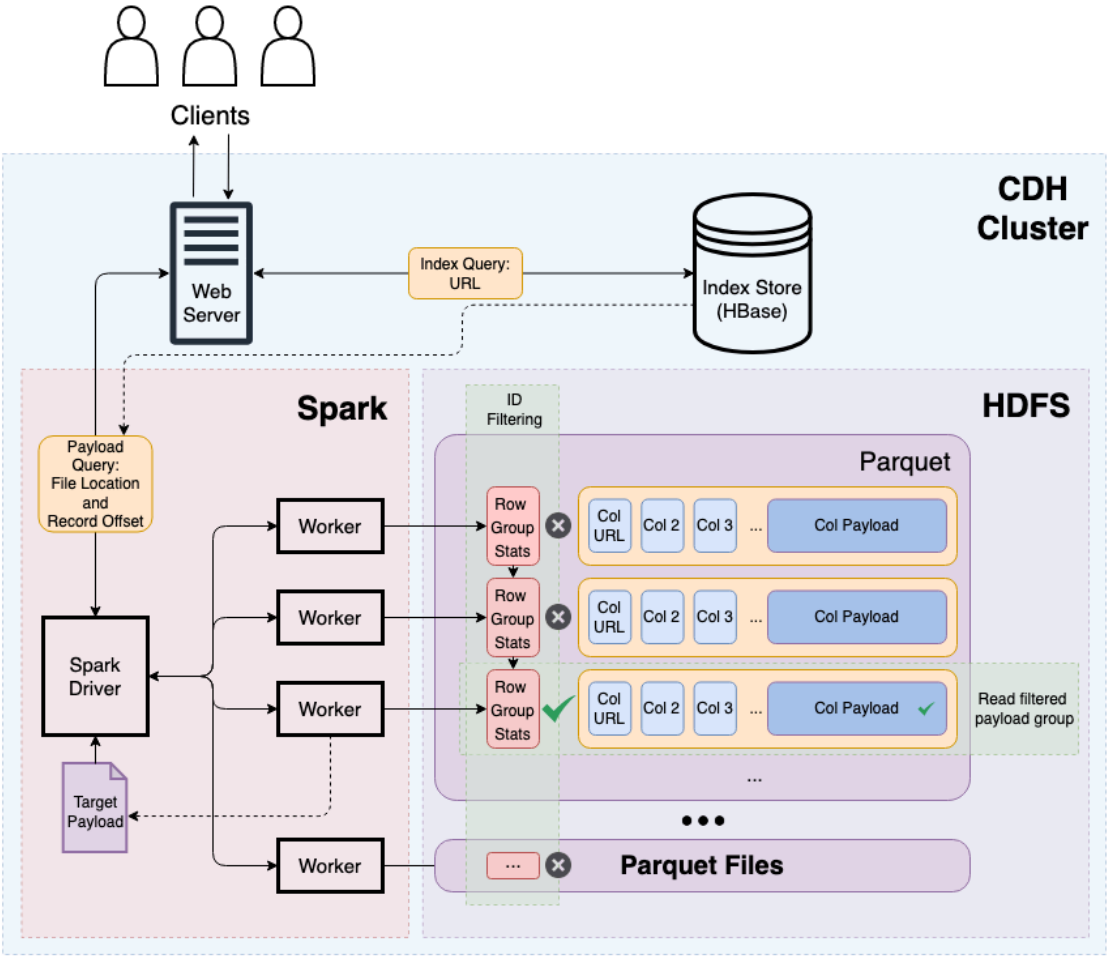


Figure 3.4: Infrastructure design for large web archive collection hosting service with raw Parquet formatted data

3.3.2 Web Archive Hosting Infrastructure

Our work aims to build an efficient, scalable, and easy to manage infrastructure for web archive hosting services with online browsing capability. This section demonstrates the overall infrastructure design for hosting large web archive collections with raw data. As shown in Figure 3.4, our proposed infrastructure design relies on Hadoop-based distributed computational environments. We utilize the following components in the Hadoop environment to accomplish our goal: HDFS, HBase, and Spark. We take advantage of the direct

access to Parquet formatted web archive through the Spark processing engine, which enables content extraction from the raw data. In addition, we leverage the high-performance searching database HBase to maintain index information where HBase will be frequently queried from the client end, which brings superior performance on index searching and access for web archive URLs.

3.3.3 Parquet Web Archive Data

We use Parquet formatted web archive data as the underlying data source to deliver the content for web browsing. We consider the Parquet formatted web archive data an equivalent to its original WARC data, in terms of preservation. In addition, Parquet formatted data can bring multiple benefits for further data consumption and services. We have demonstrated such advantages in common quantitative analytical works around the web archive in Chapter 2. The quantitative workloads in Chapter 2 primarily benefit from the data ingestion efficiency of the Parquet data format. We think the same concept can be transferred as part of the core method to leverage Parquet formatted web archive data for a browsing service. A Parquet file is sliced into multiple row groups in its internal structure, vertically. Each row group also carries the statistical information related to the fields inside this row group. One primary statistic we are looking for is the range of a specific field. If a field in Parquet data is ordered and continuous, the range statistics of this field will also be ordered and continuous. Such an ordered field can be a good candidate for data selection and scanning because the system can leverage the statistical data to identify related row groups and only consume those row groups for further processing through the predicate pushdown mechanism. This mechanism can avoid substantial data ingestion of the whole file, and avoid full scanning. We can leverage this mechanism to provide consistent data reading performance for queries related to web browsing. Thus, we generate an incremental identifier for all of the records, given a Parquet file. In the web hosting infrastructure, for each web page query, Spark will use the filename and associated identifier to scan and identify the target record. Then, the system will fully consume and scan one row group block of data to find and deliver the content. All queries of the same type submitted to our infrastructure can be automatically handled through Spark's job scheduling feature, where such queries can be processed simultaneously. As explained in Section 3.4, we benchmark this infrastructure design with load testing and show its performance.

URL Scanning Disadvantage

In addition to a query for a specific web page record, a URL searching query is another important type of query for web archive content navigation. However, Parquet formatted web archive data from sources like Common Crawl can inherit a significant disadvantage if we perform the query over the raw data.

The benchmark results from Chapter 2 demonstrated that the performance of a single URL retrieval task in Parquet varies for the targeting URL when the system treats all Parquet data as a whole collection: some search can provide superior performance while some may incur significantly poorer performance. This drawback primarily comes from the approach we are taking in Chapter 2, where we use the system to treat all web archive Parquet data as a whole collection and try to search the content in the complete data pool. In Parquet, URLs are represented as a numerical format in each row group block. The row group statistical data includes a range index for all of the URLs inside. Once a URL search is initiated, the system would first leverage the row group statistics of URL range to locate row group candidates. Then, an entire row group level search will find the exact target. In this case, we can see that the numerical distribution of URLs on the row group level significantly influences the performance of the single URL search. A URL that falls into the typical numerical range would trigger a more prolonged search due to more row group scan while numerical outliers would bring better performance due to less row group scan. In this case, the upper bound of the URL search is a full URL list scan across the whole dataset. The lower bound of the URL search can be as fast as the speed of retrieving data from a single row group. The hosting service can not rely on such unstable performance where users may experience minutes or more waiting for the response. Though the URL searching deficiency in our Parquet web archive data can be resolved by re-ordering the records inside the Parquet data based upon the URL field, such ordering would be in conflict with the existing ordering based upon crawling time.

To overcome this issue, we can use an optimized database to handle this specific workload. Such a database only needs to store minimal information related to the workload, and so imposes minimal extra storage cost. Here, we choose HBase as the solution for high performance URL scanning.

HDFS Storage

The Hadoop Distributed File System (HDFS) [23] is an open-source file system designed to provide highly scalable storage in a distributed computing environment. The brief concept of HDFS is dividing data into multiple replicated blocks and storing them across the nodes of the computing cluster where processing units can have high-throughput access to the data across all nodes in parallel. We use HDFS as the fundamental storage layer for the Parquet web archive data where processing engines like Spark directly consume the data.

3.3.4 HBase and High Performance URL Scanning

HBase is an open-source distributed database designed for large table access. HBase is built on top of the HDFS system and leverages all the benefits from HDFS. HBase is designed to provide high-speed random access to data, making it an ideal solution for queries like web page URL search. In HBase, the internal RowKey shard/partition design can bring optimal performance to tasks such as URL searching. The row key design is a significant contribution that can influence the performance of HBase searching. Figure 3.5 shows our row key design for the web archive record: SURT [51] URL followed by a vertical bar separator and then timestamp. Given a URL query, we first convert the URL to SURT URL. SURT URL is a URL form that is more friendly for grouping and sorting compared with standard URL forms. Thus, in HBase, SURT is a better choice as part of the row key design. To search a URL in our HBase design, we use an HBase row prefix scan to match the SURT URL part with the vertical bar end. We use the vertical bar as the separator because the vertical bar is not included in the reserved character list for a URL/URI by the definition of RFC 3986 [19]. Table 3.1 shows the metadata schema we maintain in HBase. As mentioned earlier in the Parquet data section, the ID and filename fields will be further consumed by Spark to extract content for specific web page queries. We will show the detailed performance with HBase in Section 3.4.

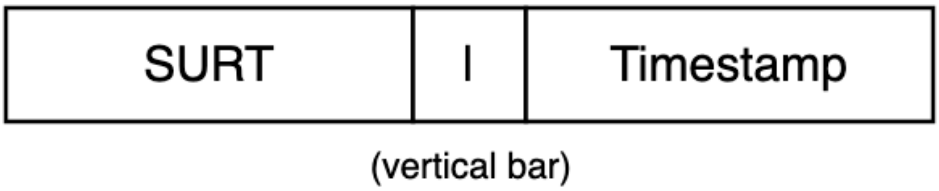


Figure 3.5: HBase row key design

3.3.5 Spark Processing Engine

Spark is a popular open-source in-memory distributed data processing framework designed for large scale data processing. Spark takes advantage of the speed of random-access memory (RAM) to provide efficient processing. The design of Spark brings a flexible, scalable, and efficient data processing experience for general data accessing and analysis. Previous efforts including ArchiveSpark [59], Warcbase [73], and ArchiveUnleashedToolkit [96]. All have adopted Spark as the processing engine to manipulate web archive data in WARC format to aid the analysis over large web archive collections. Spark is known for ETL (extract-transform-load) type workloads and large-scale data analytic jobs. However, online web services are rarely connected with Spark practices. We haven't found any Spark application that is related to web hosting and web browsing. However, we think Spark introduces a unique opportunity for web archive hosting services with Parquet data.

Field	Type
ID (sort order)	Long
Filename	String
SURT	String
Timestamp (sort order)	Timestamp
Original URL	String
Domain	String
MIME	String
HTTP Response code	Long
WARC Payload Digest	String
Redirect Url	String
Meta	String
Payload	String

Table 3.1: Parquet web archive data schema

In our application, Spark works as a back-end server that handles all of the web page requests, involving long-lasting working sessions and high loads of data reading tasks. We use Flask [55], a Python based web server framework, and PySpark, to build the communication between the web server application and Spark processing. We maintain a Spark session purposely for handling the web browsing related queries. For each query, Spark will use the filename and identifier information from the query to identify and extract the record from the Parquet web archive data on HDFS.

The performance of a Spark application can be significantly influenced by parameter settings. These parameters can be tuned to the resource allocation, job scheduling, and other internal mechanisms. Without proper settings, our Spark application could suffer significant performance drawbacks. With the default setting, our preliminary experiments reveal that our Spark application could suffer performance degradation and even service crash under a high load of testing. As an important contribution from this work, we test and find proper settings in Spark to fulfill the purposes of a hosting web archive service, with high quality performance. Our experiments involve settings for the following: application cleaner, execution unit resource allocation, and job scheduling.

Application Cleaner

During the early phase of this research, we found a fatal issue where our Spark application for web archive hosting always crashed during our maximum load testing. We found that Spark's default application cleaner mechanism leads to such a phenomenon. The application cleaner is a service to clean up internal resources within Spark that can reduce the memory requirements for long-running Spark applications, which involves Java run-time

Strategy	Cores	Memory (Gb)	Max Executors per Node	Max Executors Full Cluster
Very Big Executor	30	42	1	6
Big Executor	10	14	3	18
Balance Executor	5	7	6	36
Small Executor	2	2.8	15	90
Available Resources per Node	30	42	/	/
Available Resources full cluster	180	252	/	/

Table 3.2: Executor design

garbage collection. In the default application, the cleaner thread would run in a fashion that blocks clean-up tasks, which means each clean-up task would run upon confirmation from the previous one. In our Spark application for web archive hosting, this blocking leads to insufficient memory cleaning and eventually breaks the application. Our experiments show that if we change the setting for the application cleaner thread to operate in non-blocking fashion, where clean-up tasks can execute independently, our Spark application can avoid crashing and can maintain stable performance over long-time operations.

Execution Unit Resource Allocation

In Spark processing, each job is assigned to an executor. Each executor takes a certain amount of computational resource from the host machine. Among the settings, we can define the number of CPU cores and system memory used for one executor. We can also define the number of executors in a Spark application. The resource allocation for the executor is crucial and can significantly influence Spark applications' performance. Our experiments design four types of executor resource allocation strategy: small executor, balanced executor, big executor, and very big executor. Table 3.2 shows the details of the executor strategy we use in our experiments. We also conduct experiments on limiting the number of executors to ascertain the scalability of the performance.

Job Scheduling

Job scheduling is an essential mechanism in Spark to handle different Spark jobs in parallel. In our web archive hosting service, many payload retrieving tasks are queued in Spark when many users use the system simultaneously. In Spark, there are two job scheduling methods: FIFO and FAIR. FIFO stands for first in, first out. The job queue would always give the first job priority on allocating resources. FAIR scheduling instead assigns jobs in a "round-robin"

fashion, and all jobs get roughly equal resource allocation. We compare both methods in our experiments to determine which is better.

3.4 Evaluation

We conduct controlled experiments to benchmark the performance of our proposed infrastructure design under different settings for the web archive hosting service. Our primary purpose is to evaluate the server-side performance of our service given the scenarios demonstrated in Section 3.3.1. We use the standard server load testing method to benchmark the server performance by swarming the server with simulated simultaneous users. The load testing benchmark has two parts: one for testing the performance of the URL searching page, the other for full content extraction performance when visiting the web archive record.

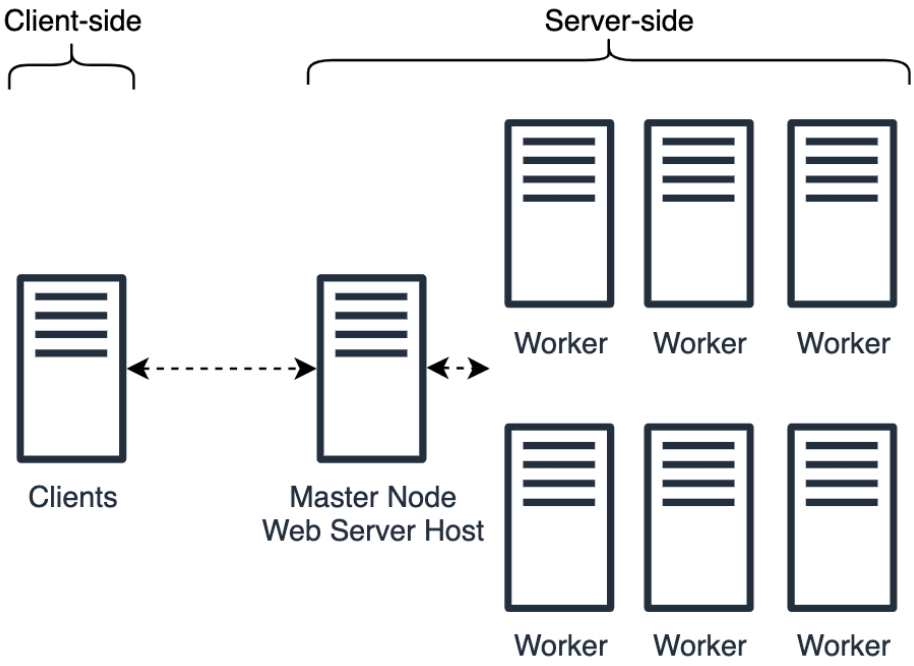


Figure 3.6: Hardware demonstration

3.4.1 Hardware Setup

Figure 3.6 shows the general hardware infrastructure for our experimental environment.

On the server side, we use the same systems as in Chapter 2, in Section 2.5.1, a CDH cluster in version 6.3.0.

On the client-side, we use a single Linux machine with Intel® Xeon® E5-2687W CPU, 128 GB Memory. The system runs the Ubuntu 18.04 operating system. Both client and server machines run within the Virginia Tech campus network. The physical locations of the two systems are within the campus, and the distance is no more than 1 km.

3.4.2 Software Setup

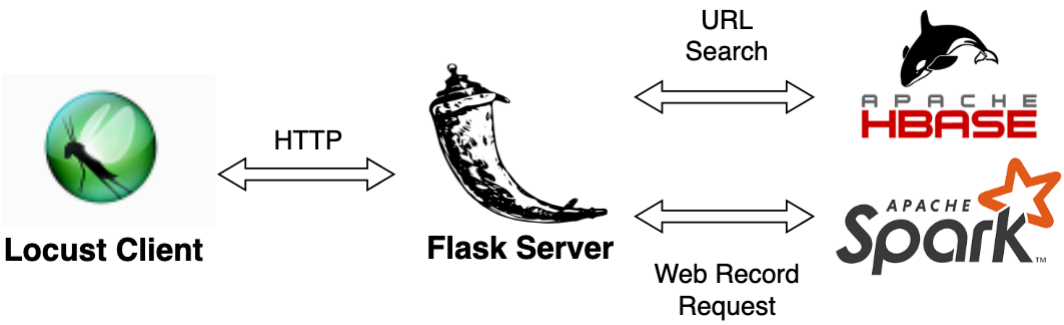


Figure 3.7: Application stack for our demonstration application

On the server-side, we use Flask [55], a minimal Python-based web server framework to handle our web hosting service in this experiment. Within Flask, we specifically use the Gevent [49] networking library as the server backend. The connection between Flask and HBase is handled through the HappyBase [57] library. The connection between Flask and Spark is through Python support from PySpark.

On the client-side, we use Locust [76], a Python-based load testing framework, to handle the load testing benchmarks for our experiments. Locust can generate a predefined number of simulated users on the client machine during load testing, and send requests to the server-side. During the testing, Locust will handle the information collection for essential metrics such as average requests per second (RPS) and average response time, where we use these metrics to evaluate the server performance.

Testing Load

We control the number of simulated clients to stress the server for each load testing experiment. We use a one-hour time window to consistently monitor the server's performance so that we can get stable results. To examine the server's maximum capacity, we linearly add more users in each experiment. We build two separate request pools with 10000 requests for URL searching and specific web record browsing. All of the requests are sampled from the raw data collection randomly. During the load testing, Locust would first randomly pick one request from the pool and then send the request to the server for response. Given a pre-defined number of clients, Locust will linearly spawn the clients up to that number,

with a rate of 100 for URL searching and 10 for web record browsing. Once the testing load reaches the defined number of clients, Locust will wait for the response for each request before sending the next one.

3.4.3 Data

We use the same dataset as we mentioned in Chapter 2, in Section 2.5.2.

3.4.4 Results and Analysis

URL Searching Performance

Figure 3.8 shows the URL searching benchmark results. We compare the performance of a URL searching page with a static page where the static page has the same content as the URL searching page without actual searching. The static page performance benchmarks demonstrate a baseline of our web hosting service under three different loads with 100 clients, 200 clients, and 300 clients. Our system can achieve over 100 RPS performance with a relatively stable response time for URL searching performance under the same load as the static page benchmark. The URL searching performance primarily relies on the HBase searching response time. We can see that HBase can produce high-quality URL searching performance that confirms our expectations. We believe this part of the service can be replaced by many other high speed database systems as well, such as Redis, that is used by the WayBack Machine. The convenient property of HBase is that it is an out of box tool in the Hadoop environment that can be easily deployed and used.

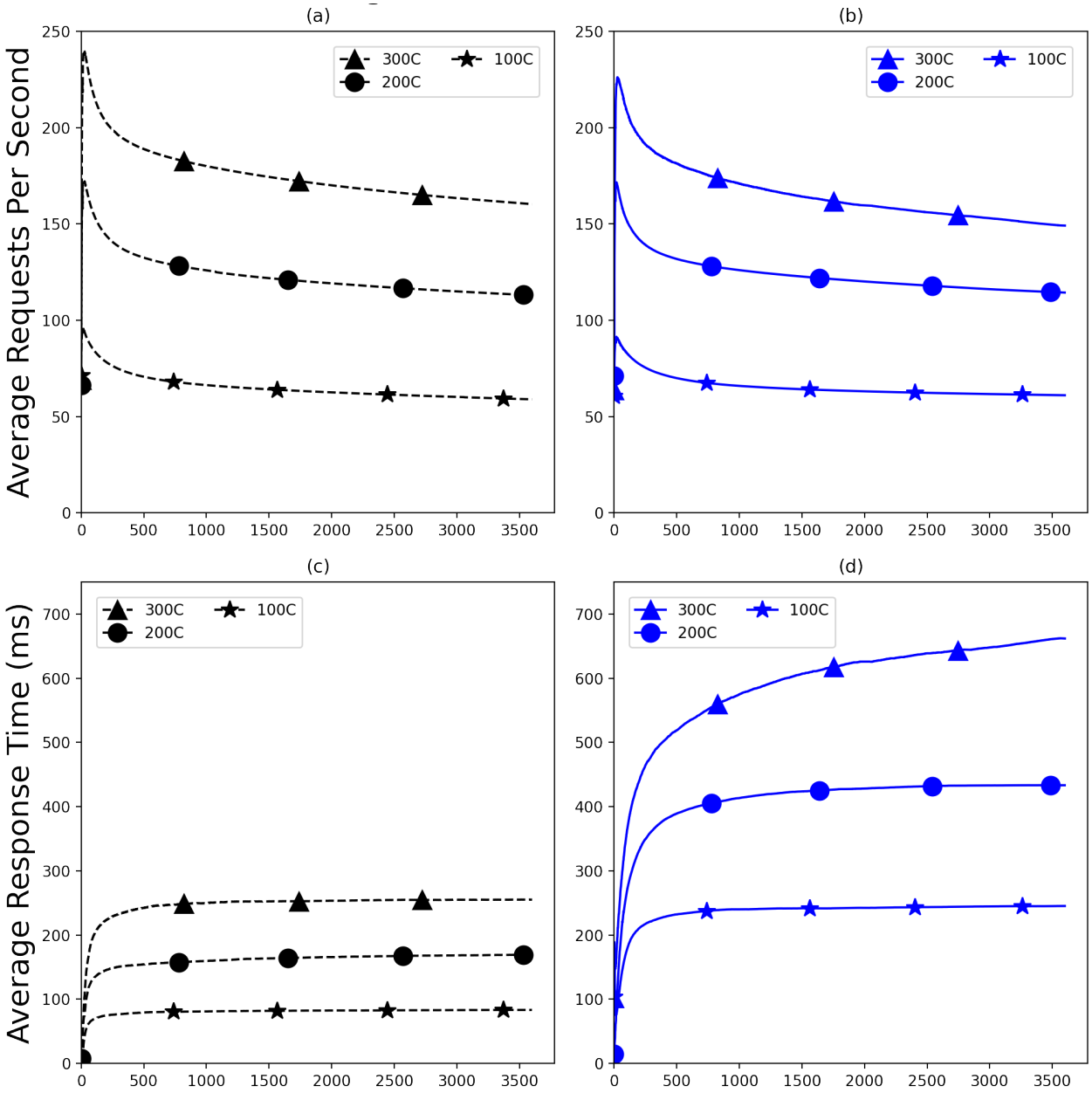


Figure 3.8: URL searching page performance compared with static page performance without searching

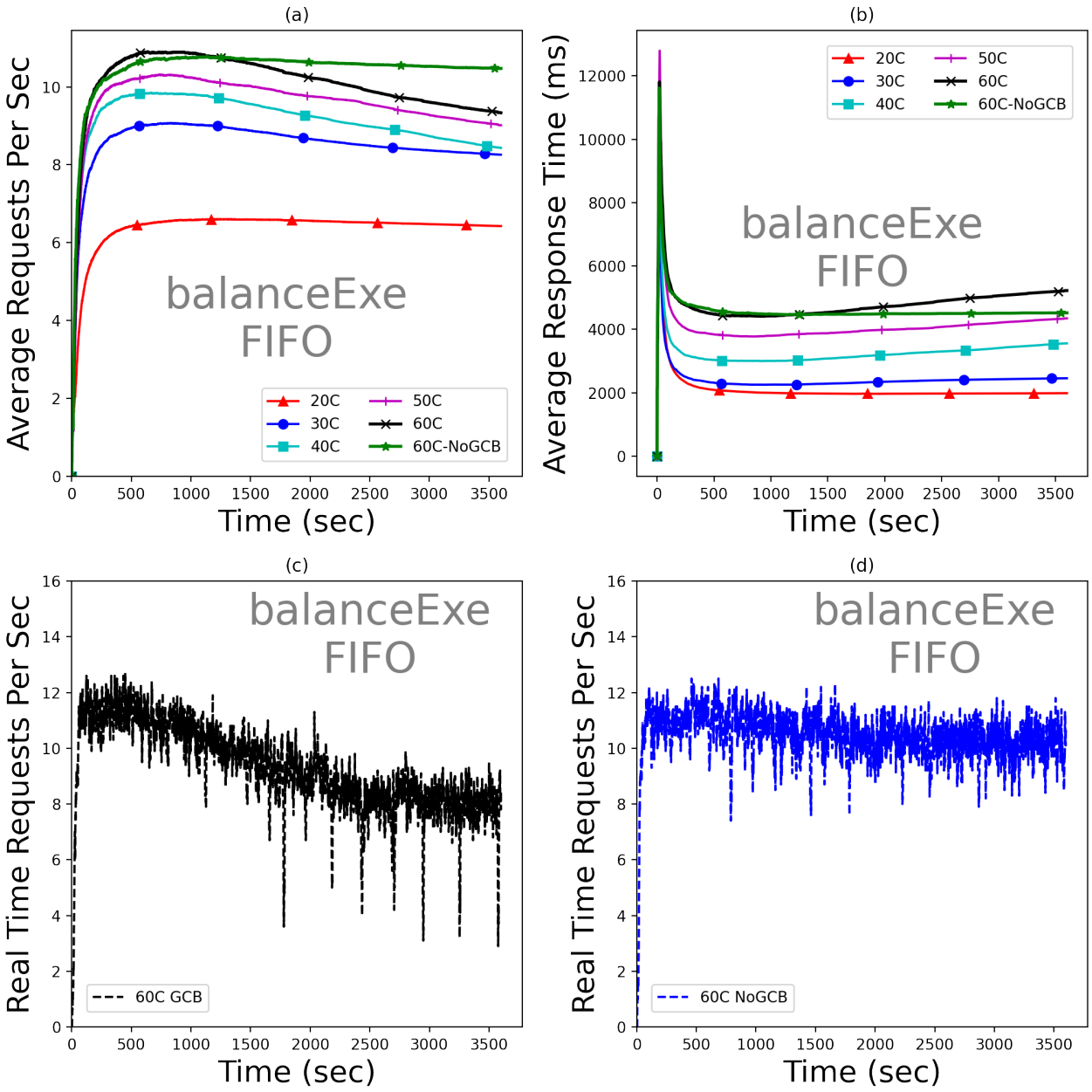


Figure 3.9: (a) shows the full content visit RPS performance with balanced executor strategy and FIFO job scheduling under different testing loads. 60C in the figure represents a 60 client testing load. The 20C, 30C, 40C, and 60C experiments all have the default Spark cleaner thread blocking enabled. 60C-NoGCB represents the experiment with no cleaner thread blocking (NoGCB); (b) shows average response time performance in the same setting as (a); (c) and (d) show the real time RPS performance for 60C GCB and 60C NoGCB.

Web Record Browsing Performance

Under default Spark settings, our experiments show that the server's performance can achieve maximally around 10 RPS with a response time of around 5 seconds. Details are given in Figure 3.9. We can see that the system can stably handle loads with 20 clients or 30 clients. With a higher load, we can see the server's performance would continuously decline over time, indicating that the server is overloaded and queued requests can not be processed in time.

In addition to the performance decline, we also find that if we let the server continue running over time, it would crash at a certain point due to a memory leak issue. We have discussed this scenario in Section 3.3.5, where we noted that the application cleaner blocking thread is responsible for such a problem. The plot for "60C-NoGCB" in Figure 3.9 (a) shows the performance of the server after we disable the application cleaner blocking. We can see that the server performance became stable at around 11 RPS with 60 clients load without a downgrade, and it would not crash after a long-time running. Figure 3.9 (c) and (d) shows a more detailed real-time RPS metric for the 60 clients loaded with GCB and without GCB. The server can achieve a stable 5 seconds response time under a 60 client load. At this point, we think our proposed Spark server infrastructure is producing a promising result on delivering the web archive hosting service, considering that both RPS and response time for the service can achieve a reasonable performance for a typical campus level of usage. We explore further improvement in the rest of the evaluation by tuning the Spark run-time settings.

The next crucial Spark tuning is the executor resource allocation strategy. Figure 3.10 shows the RPS and response performance of the Spark server under the four executor strategies. Among all the strategies, the big executor strategy achieves the best performance with around 3 percent performance gains compared with the default balance executor design. In this case, we conclude that our web hosting service favors a larger executor strategy over the small one. However, in the very big executor strategy, we take an extreme scenario where one worker node only contains one executor allocated with all the available resources. We see that the big executor plan outperforms the very big one. This finding reveals that the relation between executor resource and performance is not linear.

Figure 3.11 shows the differences between the two job scheduling methods in Spark. Under the same load, the FAIR scheduler performs around 6 percent better than the FIFO scheduler and 10 percent better than the default setting.

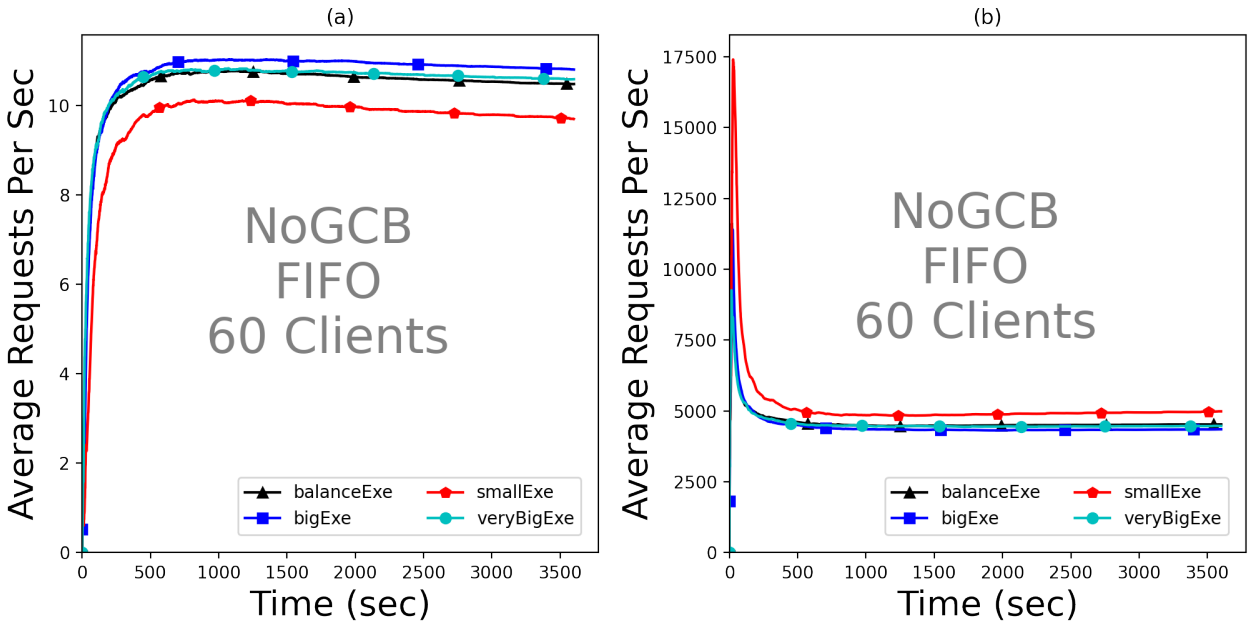


Figure 3.10: (a) shows the full content visit RPS performance with different executor strategy and FIFO job scheduling and NoGCB; (b) shows the average response time performance in the same setting as (a)

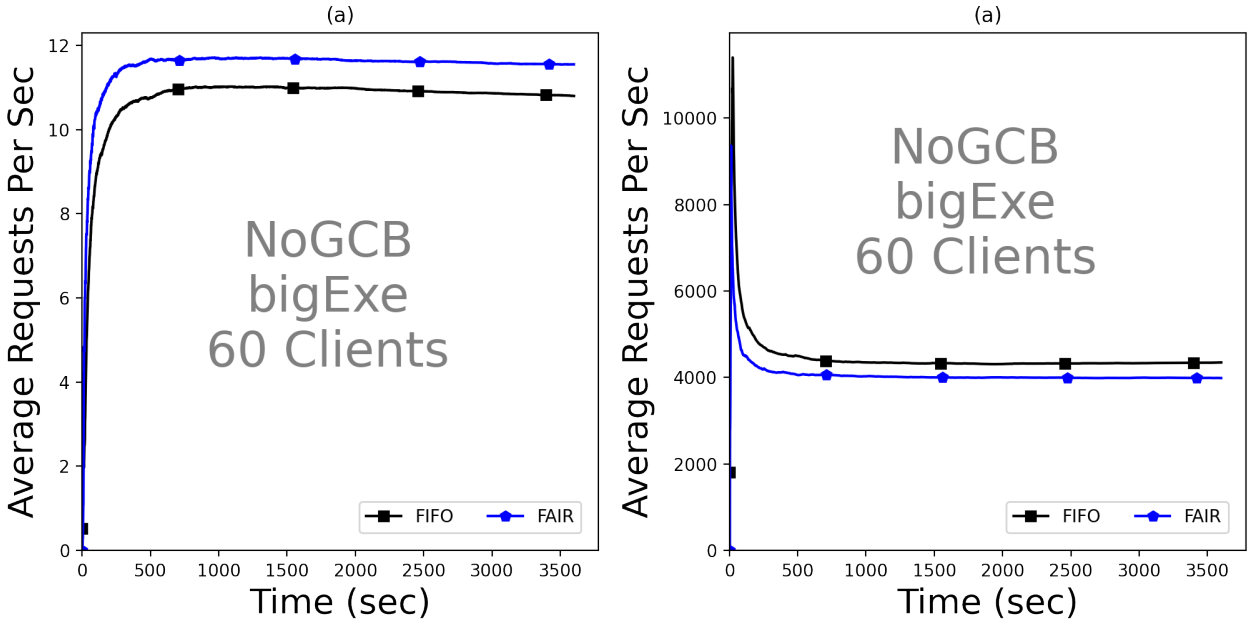


Figure 3.11: (a) shows the full content visit RPS performance of the two job scheduling methods with the big executor strategy and NoGCB; (b) shows the average response time performance in the same setting as (a)

By default, for all the experiments above, Spark would try to take as many resources as possible for the load testing with the growth of testing load and eventually allocate all available resources to the Spark application. In our service, we visualize it as a hybrid environment where other types of workload or analysis could happen simultaneously with the hosting web service. We may not want our system to take all of the resources dedicated for the hosting web service. Thus, we conduct experiments to see the scalability of the Spark application with a limited number of executors across the cluster. Surprisingly, the server's maximum performance does not linearly scale with the executor numbers. The optimal setting we have observed so far is: big executor strategy, FAIR scheduler, and NoGCB.

As shown in Figure 3.12, the experiment with nine limited executors across the clusters can achieve around 22 percent performance gain compared with the total 18 executors across the cluster. Our cluster monitoring finds that the system maintains a higher disk read throughput under nine executors than other settings. Due to the fact that our Spark application is reading intensive, the reading locality becomes a crucial factor to influence the performance. The higher read efficiency in this setting indicates that such a setting can provide a better read locality. In the scenario where there is an excessive number of executors, many jobs in Spark will be assigned to executors with less locality. Executors with less locality need to obtain and cache data from the source that is not local for this executor, and then provide the required processing. In this case, the optimal executor number for the system depends on hardware factors such as disk performance, processor performance, and network performance. Different systems would need experiments to determine the best settings.

In the scenario with inadequate executors, comparing with the best performance case, we also observe that the system can maintain nearly 80 percent of performance with only three executors. Imagine the system is loaded with other heavy analysis tasks. The service can still maintain a relatively solid performance with minimal resources allocated, showing that our proposed infrastructure can work consistently in a dynamic working environment with different loads.

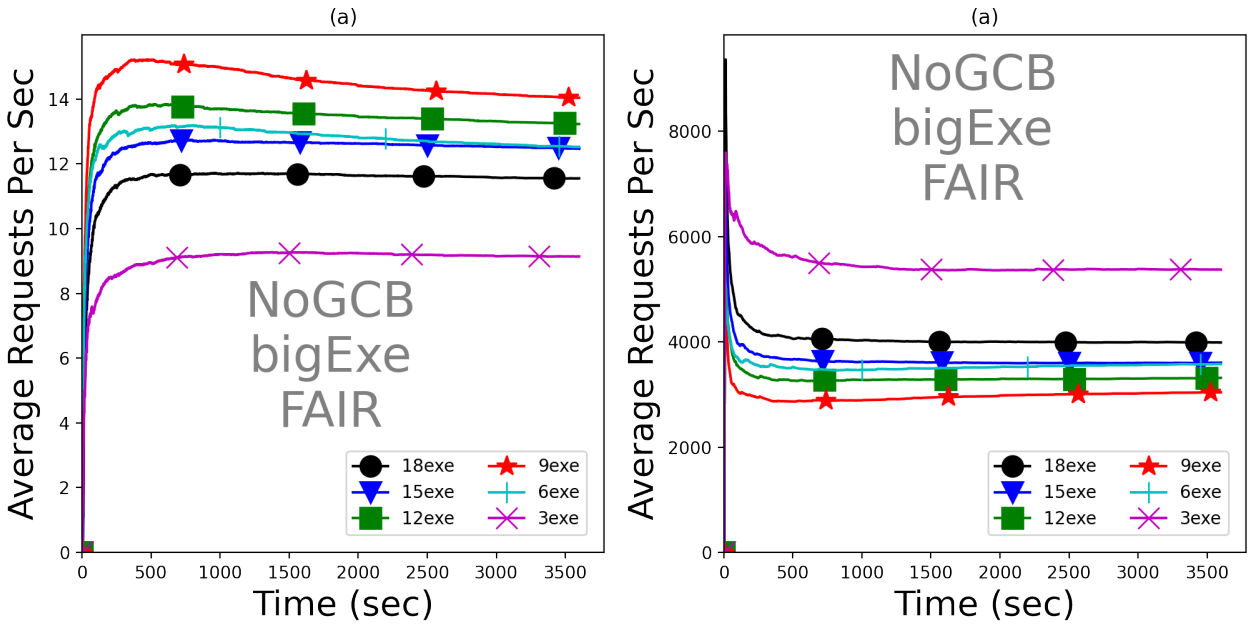


Figure 3.12: (a) shows the full content visit RPS performance of different limited numbers of executors with the big executor strategy, FAIR scheduler, and NoGCB; (b) shows the average response time performance in the same setting as (a)

3.5 Conclusions and Discussion

In Chapter 2, we discussed that use of the WARC format can bring extra processing burden for web archive reuse. We demonstrated the performance advantage of Parquet formatted web archive data under the Hadoop and Spark big data processing framework. In this chapter, we further expanded the usage of Parquet formatted web archive data to establish an infrastructure for a web browsing service through raw data access on the file system. Our evaluation demonstrated that our infrastructure can bring a consistent and stable browsing service over the raw Parquet web archive data. In addition, our infrastructure maintains a low level data replication with only index information stored in HBase. What's more, our infrastructure shares the processing components that we use for quantitative data analysis. The browsing service and quantitative analysis can co-exist in the infrastructure. The browsing service can also maintain a stable performance even when most resources are allocated to other workloads.

In Chapter 2, we recommended that a transition from Submission Information Packages (SIPs) format like WARC to Archival Information Packages (AIPs) format like Parquet can significantly improve the computation efficiency around web archive data. Here, we show an additional application where such transition brings benefits when building a browsing service with large web archive collections. The compatibility of modern data formats like Parquet is

a strong advantage for its multi-use capability. In our work, we have built our infrastructure mainly with Spark and Parquet. Many other big data frameworks like Apache Flink, Presto, or Apache Hive can all leverage such a data format efficiently in their processing. In other words, web archive initiatives can explore more opportunities to construct their services for different needs without a development and processing burden. We believe that web archive initiatives can benefit from such a infrastructure design philosophy to expand their web archive related services. Such an expansion of web archive services can accelerate the reuse and discoveries from our web history.

This chapter is supplemented by Appendix B, which provides more implementation details for the accompanying experiments.

Chapter 4

Twitter Data Analysis Integration and Acceleration

This chapter addresses the final research question: *Can we integrate social media archive like Twitter data into the infrastructure and enable efficient reuse?* In Chapters 2 and 3, a computational infrastructure for analyzing and storing web archive collections in Parquet format has been built. This chapter proposes the incorporation of historical Twitter data sets into our proposed infrastructure to support Twitter data analytics. Twitter data preservation lacks a uniform standard and has a more complicated data structure compared to WARC. Therefore, we propose a uniform data format for Twitter to define all of our Twitter collections. Similar to the web archive data, as depicted in Figure 4.1, we propose converting our existing Twitter data collections to the Parquet format with a consistent schema for optimal storage and processing.

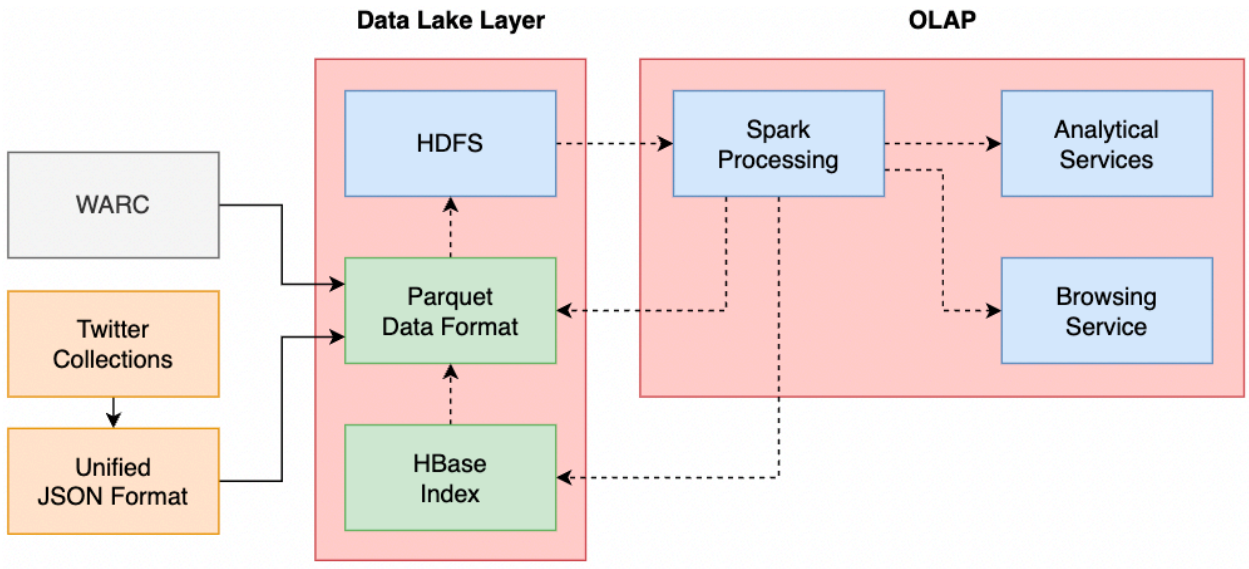


Figure 4.1: Integrate Twitter data into the infrastructure

4.1 Introduction

The emergence of social network platforms has a remarkable effect on the rate of human social contact on the Internet. Simultaneously, inadvertent online footprints and content are being rapidly generated. Twitter Inc. is one of the leading social networking sites that emphasizes rapid communication through the creation of brief messages known as 'tweets.' Twitter restricts user posts to 280 characters (140 before 2017). This format facilitates the efficient generation of content. Users can also acquire more information in less time using this mechanism. Twitter plays a vital function as a media source for numerous organizations, in addition to serving as a conduit for messages from normal users. According to corporate data, there are more than 300 million Twitter users as of 2021, and they produce more than 500,000 tweets each minute [102, 103]. Twitter's vast amount of user-generated content has propelled the area of quantitative social media analysis. Such analysis has proven useful in numerous fields, including the financial sector, disaster analysis, and political events.

Twitter data gathering has been an integral aspect of the web archive preservation activities at Virginia Tech. In addition to the normal web archive collections, the Twitter archives constitute a substantial portion of our digital knowledge base. The archive of the university library has more than six billion tweets. Natural disasters, political events, crimes, and pandemics are among the topics covered. Our Twitter data archive is approximately one terabyte in size. In contrast to regular web archive data, Twitter data is typically gathered and structured differently. Conventionally, Twitter data collectors crawl and parse the website's content using web crawling techniques. Instead of storing the complete page's information, collectors often isolate each tweet's primary content and store it in a predetermined structure. Modern Twitter data collection tools, such as Social Feed Manager (SFM) [74], have arisen to aid researchers in the systematic collection of Twitter data. Twitter data may be stored in plain text, CSV, or JSON formats [92], or in database systems like MySQL [70]. There is one instance in which tweets can be collected similarly to web archives: Social Feed Manager aids in the preservation of Twitter data by utilizing the web archive standard WARC format. Consequently, distinct Twitter data collections may necessitate distinct tools and methods for consumption. Such fragmented representations of Twitter data in distinct collections make research inefficient.

This study proposes an uniform tweet schema that organizes tweet data from various sources for effective usage. In addition, as an increasing number of studies utilize Twitter collections on a large scale, computing efficiency becomes a concern, since such analyses could take a very long time. As is the case with regular web archive collections, we believe these issues can be remedied by including Twitter data in our web archive big data infrastructure. The second and third chapters illustrate the efficacy and scalability of our contemporary networked computing infrastructure while working with ordinary web archive data. This file-based storage and processing infrastructure is also intended to be adaptable to other data types. Using a consistent file-based storage and processing pipeline, we can enhance the data processing efficiency of the Twitter collections by applying the same idea. Conse-

quently, we propose storing Twitter data using a uniform standard (schema) for file storage format. Such a structure is easily applicable to file-based storage formats like JSON. In addition, we propose converting the JSON data to the Parquet data format, comparable to the WARC-Parquet conversion. Twitter data is primarily derived information in an organized fashion, as opposed to the unstructured nature of web archive payload material. In terms of data compression and serialization, the columnar data format, such as Parquet, is more advantageous for data of this type, which eventually contributes to more efficient processing. This study evaluates the performance differences between Twitter data in JSON and Parquet formats. We also investigate the ideal Twitter data storage and processing configuration for the Spark processing pipeline.

4.2 Related Work

In this section, we describe relevant work that serves to justify the integration of Twitter data with our proposed infrastructure and system for big data analytics. A preliminary study by DLRL's Matthew Bock [22] investigates an infrastructure design that leverages the Hadoop ecosystem to facilitate the general reuse of Twitter data with frameworks such as HBase, Spark, and the Avro data format. The work of Matthew Bock highlights the potential of a big data framework in terms of data processing efficiency, where the combination of the HBase and Avro file formats can give a solid synergy. This study emphasizes file-based data storage and related workloads more than Matthew Bock's work. Instead of Avro, we propose using the data format Parquet, which is compatible with the current web archive processing infrastructure. The processing efficiency issue is not new to the world of Twitter data analysis. In a growing number of scientific studies involving massive Twitter datasets, the data are getting more difficult to consume and analyze. Researchers have adopted commercial big data infrastructures as methods to enhance the scalability of Twitter data's analytical capabilities. Bhardwaj et al. used Apache Hive, a data warehouse system within the Hadoop environment, to conduct a case study utilizing Twitter data [20]. Hive offers scalability to the processing of Twitter data. However, it is unclear which precise question was assessed in this study, so no performance insights can be gleaned. Nodarakis et al. proposes to use Hadoop MapReduce distributed processing and new algorithmic enhancements to accelerate the performance of Twitter sentiment classification [86]. Rodrigues and Chiplunkar proposes a general infrastructure design with the Hadoop ecosystem, Apache Pig/Hive, for Twitter data real-time ingestion and analytical workloads, including the discovery of trending hashtags and sentiment analysis [94]. Similarly, Kamal et al. embrace the concept of using Apache Hive as a layer of intermediate storage to accelerate opinion mining with Twitter data [65]. Farhan et al. compare the speed of the Hadoop MapReduce framework with Apache Spark when analyzing Twitter data with JSON storage. [42] The work reveals that Apache Spark is significantly superior to the MapReduce framework, validating our architectural design.

It is evident from the articles examined here that many academics prefer Hadoop and Spark

processing over Twitter data processing. In addition, this evaluation indicated that the majority of studies have only demonstrated a general improvement when applying the new infrastructure to Twitter data for simple tasks. Several difficulties remain: (1) Specialized data stores such as Hive continue to introduce issues such as data redundancy, lack of data manipulation flexibility, and mobility. (2) There is no comprehensive benchmark for typical Twitter analytic workloads. (3) Modern data formats such as Parquet are carefully designed for the framework to eliminate the storage layer bottleneck, which has not been investigated. In order to overcome the aforementioned concerns, we study further, on top of Apache Hadoop and Spark, the storage layer bottleneck of Twitter data analysis workloads with JSON and Parquet file storage.

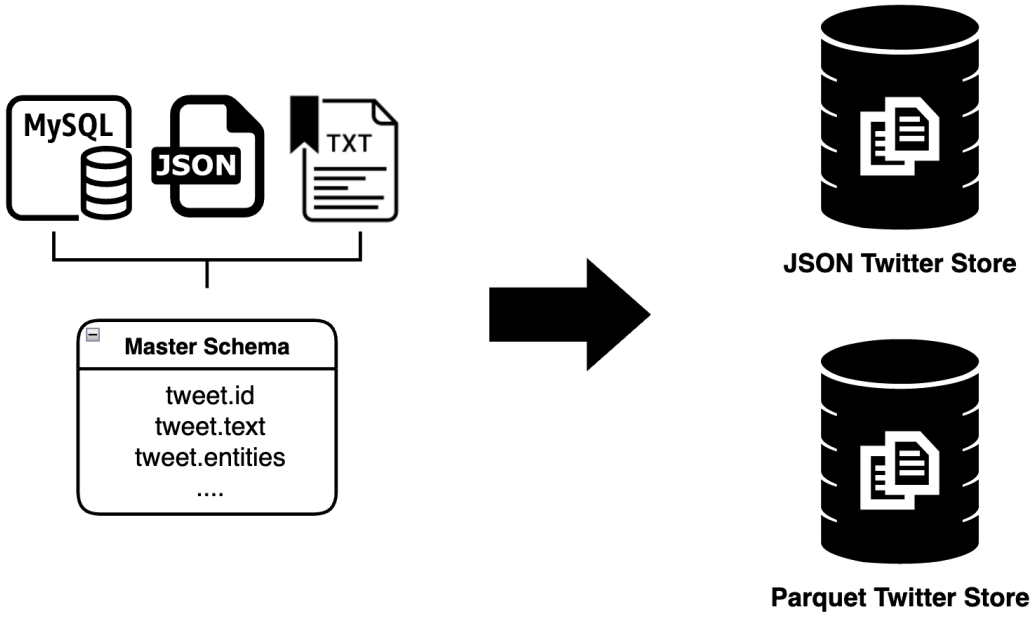


Figure 4.2: Apply master Twitter data schema to JSON and Parquet file formats

4.3 Unify Twitter Data Storage

We suggest consolidating and conserving Twitter data in a uniform schema, which is a superset of the original storage sources. With the unified schema, we are able to save data in file-based formats comparable to those used for regular web archive preservation. In lieu of JSON or database systems, we propose using the Parquet data format as an efficient file-based format for Twitter data preservation on top of the unified schema.

4.3.1 Segmented Twitter Data

Numerous Twitter data collections are fragmented in multiple storage standards, preventing the collection as a whole from being utilized in an efficient manner. Different data gathering pipelines, including Social Feed Manager (SFM) [74], yourTwapperKeeper (YTK) [1], and DMI-TCAT [61], are the root cause of this issue. All of these tools have been useful for the capture and storage of Twitter data. However, the storage format of the collected data is inconsistent. All of these technologies employ their own non-standard internal storage structure for storing raw data, and then provide export alternatives including JSON, CSV, and plain text. However, even using the same export format (e.g., JSON), the schemas of the exported data still differ, creating an obstacle for uniform data consumption.

4.3.2 Unified Schema

To address the issue of data consistency with Twitter data, we propose constructing a single data structure for Twitter data that may incorporate several data collectors. In this instance, we build a single data schema as the master template for defining and ingesting Twitter data. As a standard, we adhere to the data definitions and naming rules outlined in the official Twitter website. A student group's collaborative effort during the CS4624 course [38] addresses part of the schema design procedure. The schema design follows documentation from official Tweet object definitions[107]. The finalized Twitter data schema is shown in Figure 4.3.

The majority of the data fields in the Twitter unified schema are string data types. Based on Twitter document definitions, we apply data types to numerical fields, including long integer and double float. Several fields, including hashtags, URLs, and media, are arrays of string or integer members. With this single Twitter data structure, we can incorporate all of our existing and any future data sources. As the raw data storage source, we apply the uniform schema and transform all Twitter data to JSON Parquet files. In the following section, we will explore how to apply this schema to Twitter data and convert it to the Parquet file format.

```
root
 |-- contributors: string
 |-- created_at: string
 |-- display_text_range: struct  (nullable)
 |    |-- start: string  (nullable)
 |    |-- end: string  (nullable)
 |-- entities: struct  (nullable)
 |    |-- hashtags: array  (nullable)
 |    |    |-- element: string (nullable)
 |    |-- media: array  (nullable)
 |    |    |-- element: string (nullable)
 |    |-- urls: array  (nullable)
 |    |    |-- element: string (nullable)
 |    |-- user_mentions: array  (nullable)
 |    |    |-- element: string (nullable)
 |-- geo: struct  (nullable)
 |    |-- country: string  (nullable)
 |    |-- latitude: double  (nullable)
 |    |-- longitude: double  (nullable)
 |    |-- type: string  (nullable)
 |-- id: string  (nullable)
 |-- in_reply_to_screen_name: string  (nullable)
 |-- in_reply_to_user_id: long  (nullable)
 |-- in_reply_to_status_id: long  (nullable)
 |-- is_quote_status: string  (nullable)
 |-- lang: string  (nullable)
 |-- lang_iso_code: string  (nullable)
 |-- metrics: struct  (nullable)
 |    |-- favorite_count: long  (nullable)
 |    |-- retweet_count: long  (nullable)
 |-- result_type: string  (nullable)
 |-- source: string  (nullable)
 |-- text: string  (nullable)
 |-- user: struct  (nullable)
 |    |-- contributors_enabled: string  (nullable)
 |    |-- created_at: string  (nullable)
 |    |-- default_profile: string  (nullable)
 |    |-- default_profile_image: boolean  (nullable)
 |    |-- description: string  (nullable)
 |    |-- follow_request_sent: string  (nullable)
 |    |-- has_extended_profile: string  (nullable)
 |    |-- id: string  (nullable)
 |    |-- is_translation_enabled: string  (nullable)
 |    |-- is_translator: string  (nullable)
 |    |-- lang: string  (nullable)
 |    |-- location: string  (nullable)
 |    |-- real_name: string  (nullable)
 |    |-- notifications: string  (nullable)
 |    |-- screen_name: string  (nullable)
 |    |-- time_zone: string  (nullable)
 |    |-- translator_type: string  (nullable)
 |    |-- url: string  (nullable)
 |    |-- utc_offset: string  (nullable)
 |    |-- verified: string  (nullable)
 |    |-- withheld_in_countries: string  (nullable)
 |    |-- metrics: struct  (nullable)
 |    |    |-- favorites_count: long  (nullable)
 |    |    |-- followers_count: long  (nullable)
 |    |    |-- friends_count: long  (nullable)
 |    |    |-- statuses_count: long  (nullable)
 |    |    |-- listed_count: long  (nullable)
 |-- entities_hashtags: array  (nullable)
 |    |-- element: string (nullable)
 |-- id_long: long  (nullable)
 |-- geo_latitude: double  (nullable)
 |-- geo_longitude: double  (nullable)
 |-- timestampUTC_int64: long  (nullable)
```

Figure 4.3: Twitter data schema. “nullable” field is allowed to be empty.

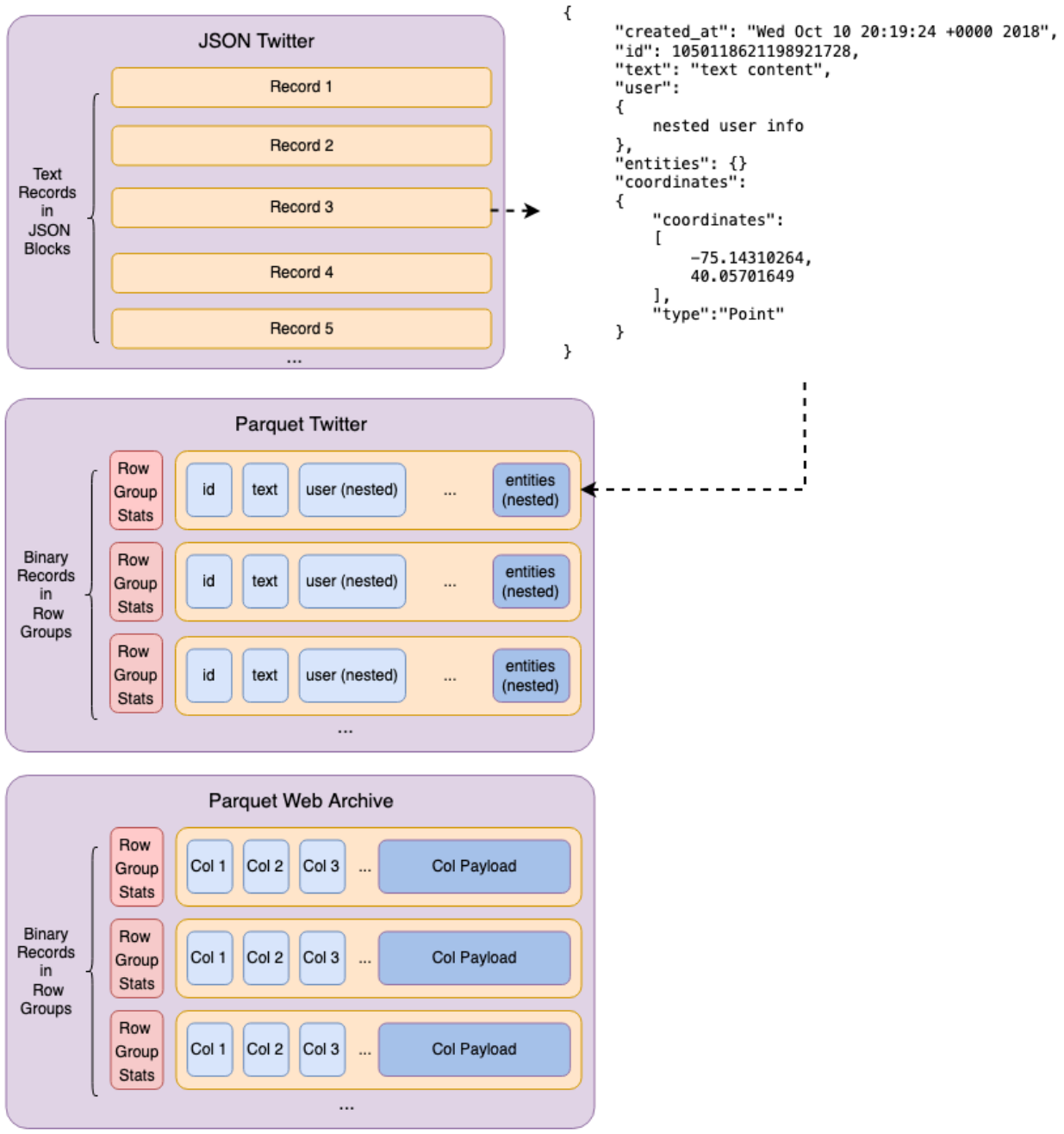


Figure 4.4: Comparing JSON Twitter, Parquet Twitter, and Parquet web archive file structures

4.4 Tweets in Parquet Format

The advantages of utilizing the Parquet data format for web archive collections were outlined in Chapters 2 and 3. We believe Twitter data collections can benefit from the advantages of the Parquet data format in similar circumstances. Twitter data inherits a well-designed data structure from our schema design, in contrast to web archive data where the primary payload field is textually structured within the file and accounts for the majority of information. Figure 4.4 shows the comparison of JSON Twitter data, Parquet Twitter data, and Parquet web archive data. Because the core tweet text content is typically much smaller than a web page payload, Twitter data fields are also more evenly spread compared to online archive data. Additionally, some Twitter data fields can be nested, whereas conventional web archive data fields are flattened. Parquet data format is created in a columnar approach. It allows nested data structures within each column by mapping and flattening nested fields into subcolumns without modifying the original structure. In this instance, we can easily convert Twitter data in JSON format to Parquet format. According to numerical fields with linear distributions, we have demonstrated in Chapter 2 that data fields in the Parquet data format that keep adequate statistical information can considerably improve the processing performance. These fields include string elements that can be mapped to dictionaries. Numerous Twitter data fields, including ID, timestamp, and location, provide appropriate statistical features. We anticipate a similar performance increase with Parquet Twitter on regular data filtration and selection workloads as with Parquet web archive data. In the performance evaluation section, we will evaluate JSON- and Parquet-formatted Twitter data on chosen Twitter analytic workloads to highlight the performance benefits of the Parquet data format.

4.5 Evaluation

4.5.1 Workloads

In this research, we will examine the advantages of Twitter data storage in Parquet format instead of with normal JSON format. To assess the performance of our infrastructure design, we employ five common real-world workloads. Spark Scala scripts are used to implement every workload. In our benchmark results, each experiment was conducted five times. Workload implementation details can be found in Appendix C. The tasks are listed below:

- Task 1: Select a subset of tweets with a list of tweet IDs.
- Task 2: Select a subset of tweets with a time range.
- Task 3: Select a subset of tweets with location bounding box.

- Task 4: Generate trending hashtags.
- Task 5: Generate sentiment analysis for each tweet.
- Task 6: Web hosting performance benchmark for individual tweet retrieval via tweet ID.

The first three tasks (1 to 3) are examples of workloads that frequently take place during data filtration and selection scenarios. Users are expected to have a list of Twitter IDs for which they wish to retrieve detailed content, given a big Twitter data repository comprising multiple collections. In this instance, for Task 1, the system retrieves the complete records of all the associated tweets from the data store using a tweet ID list file. 1 percent, 5 percent, and 10 percent of the entire collection are covered by the three ID lists that we employ.

In a variety of cases, the user would be interested in tweets within a specified time frame or geographic region. Consequently, in Tasks 2 and 3, the algorithm obtains all tweets matching the search criteria. In Task 2, we employ four distinct time intervals to assess the selectivity of the entire collection: 4, 8, and 11 percent. Similar to Task 2, three selectivity levels are used in Task 3: 1, 2, and 3 percent. In the case of Task 3, we are unable to increase the selectivity because the percentage of tweets having geo-location information is limited to 3 percent.

Tasks 4 and 5 indicate extensive analytic workloads associated with tweets' core content. Twitter hashtag analysis is widely practiced in numerous disciplines. In a number of data visualization systems, trending hashtags are generated to highlight attractive data subjects. In Task 4, we count and rank all collected hashtags to build a list of trending hashtags. Sentiment analysis, in which researchers examine the tone of tweets, is an additional popular Twitter data analysis task. This task typically entails the application of the latest artificial intelligence techniques such as deep learning. In Task 5, we classify positive and negative sentiment for each tweet in our collection by applying a sentiment analysis pipeline.

Task 6 benchmarks the web hosting performance for individual tweet retrieving and browsing. In this scenario, a user sends out a request that specifies a unique tweet ID, then the server will respond with the tweet content in JSON format. The benchmark environment shares the same setup as discussed in Section 3.4. The tweet retrieval task is performed over the raw Parquet formatted Twitter data. The Parquet formatted Twitter data is naturally ordered by tweet ID. Therefore, we expect the web hosting can take advantage of the predicate pushdown capability to boost the efficiency of web hosting task.

4.5.2 Data

Twitter data collections were obtained by the Digital Library Research Laboratory and University Libraries over a number of years, covering a broad range of topics. The data set we selected is comprised of 431 JSON files in plain text format that adhere to the data schema established in Section 4.3.2. The collection is 456.3 GB in size and comprises 784,908,073 tweets. The JSON Twitter data files are reformatted to the Parquet format. The core data structure of Parquet Twitter data conforms to the same schema as JSON data. We employ the following parameters to build the JSON and Parquet formats:

1. The number of records contained in each JSON and Parquet partition: We set a predetermined number of records per partition while generating the data with Spark. In Spark parallel processing, partitions indicate the number of concurrent processing lines. The varying number of partitions will result in varying numbers of incomplete files for each collection. For instance, if the total size of a collection is 1 GB, five partitions will result in five partial files for this collection on the HDFS storage. Typically, the number of partitions is defined by the amount of original data. Due to the changes in collection size, we utilize a set number of records to enforce the partition number, as the size variance for each tweet is tiny. In our performance evaluation, we will compare the impact of 128k, 256k, and 512k partitions on the performance of analytical workloads containing 128k, 256k, and 512k records, respectively.

2. Compression method: We generate both Snappy and standard Gzip compressed files. In the performance benchmark, we will compare the two compression algorithms.

3. Row group (block) size for Parquet data: In Chapters 2 and 3, we covered the row group setting for web archive Parquet data. In this chapter, we will investigate the effects of row group sizes on Twitter data analytics workloads. We include the 32 MB, 64 MB, and 128 MB row group sizes in our benchmark trials. Consequently, this change in group size will be reflected in Parquet data HDFS storage on the block size, which is equal to the row group size. This value cannot be modified for JSON data and defaults to the HDFS block size setting of 128 MB.

Table 4.1 shows the data size of the data types we mentioned above. Under Snappy and Gzip settings, Parquet is approximately 30 percent more space-efficient than JSON. Chapter 2 demonstrates that the storage efficiency of Parquet web archive data is approximately 5% more than that of WARC. In this instance, Twitter data obtains greater storage benefits from the Parquet data format due to its distinct data structure. In the best case scenario, we can observe that the Parquet Gzip format requires just 1/10 of the original data size when stored in JSON plain text format. This storage efficiency enhancement can drastically cut storage capacity costs.

	JSON Plain	JSON Snappy	Parquet Snappy	JSON Gzip	Parquet Gzip
Size (Gb)	456.3	100.7	67.1	58.8	42.2

Table 4.1: Data size of different data types.

4.5.3 Hardware and Software Setup

We employ the same hardware setup for all benchmark workloads as described in Section 2.5.1 of Chapter 2. Differently from Chapter 2, the CDH distribution environment is upgraded to 6.3.2 rather than 6.3.0 (Chapter 2). Nonetheless, the core software versions remains the same as CDH 6.3.0, including Hadoop, HDFS, Parquet, and Spark.

4.6 Results and Analysis

Figures 4.5 - 4.9 show the benchmark results for JSON and Parquet data under the default settings: Snappy compression, 256k records partition, and 128 block size. Parquet formatted Twitter data vastly outperforms JSON formatted Twitter data across all workloads. As shown in these figures, Parquet is approximately two times more effective at tweet ID matching in Task 1, approximately 2.5 times more effective at timestamp and geo-location filtration in Tasks 2 and 3, approximately two times more effective at trending hashtags generation in Task 4, and nearly three times more effective at sentiment analysis in Task 5.

Similar to Parquet web archive data, Parquet formatted data provides a considerable performance boost via predicate pushdown and projection pushdown on the data loading side, as demonstrated by Tasks 1, 2, and 3. At this point, it is apparent that Twitter data in Parquet format is an efficient solution for our web archive analysis infrastructure. Twitter data can be saved and handled similarly to web archive data formatted in Parquet while attaining higher performance. In the following sections, we examine the Twitter Parquet storage options in detail.

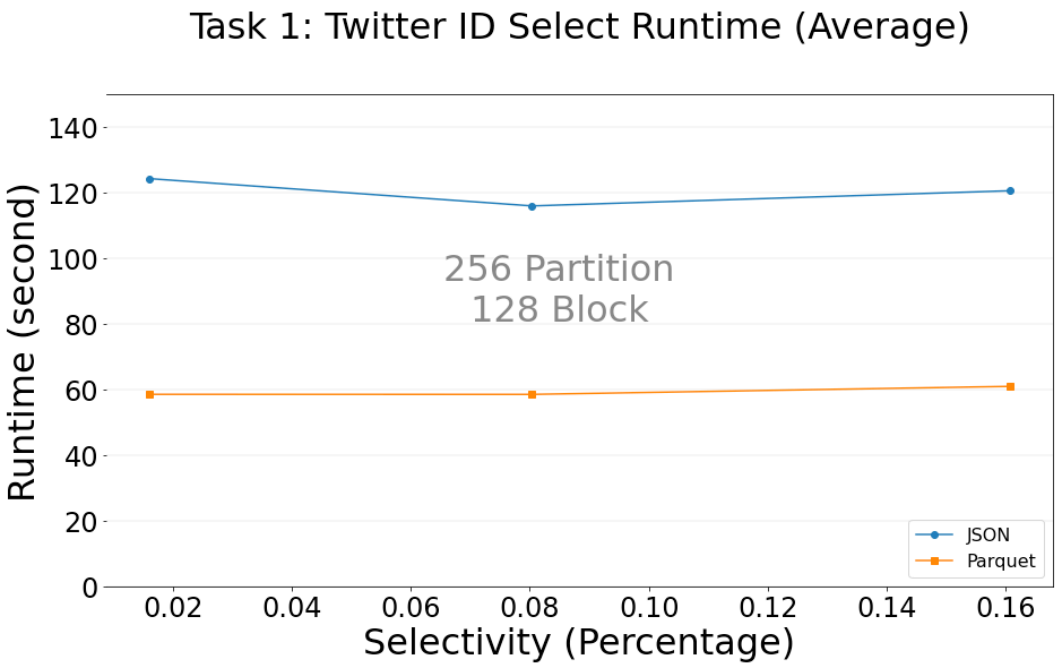


Figure 4.5: Task 1: Select a subset of tweets with a list of tweet IDs

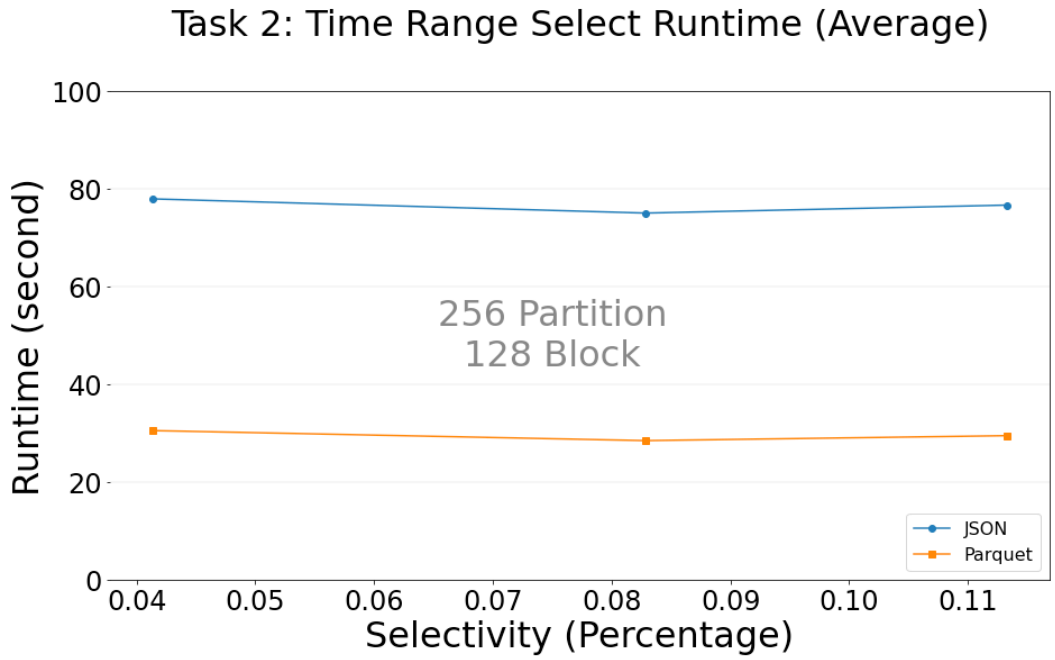


Figure 4.6: Task 2: Select a subset of tweets with a time range

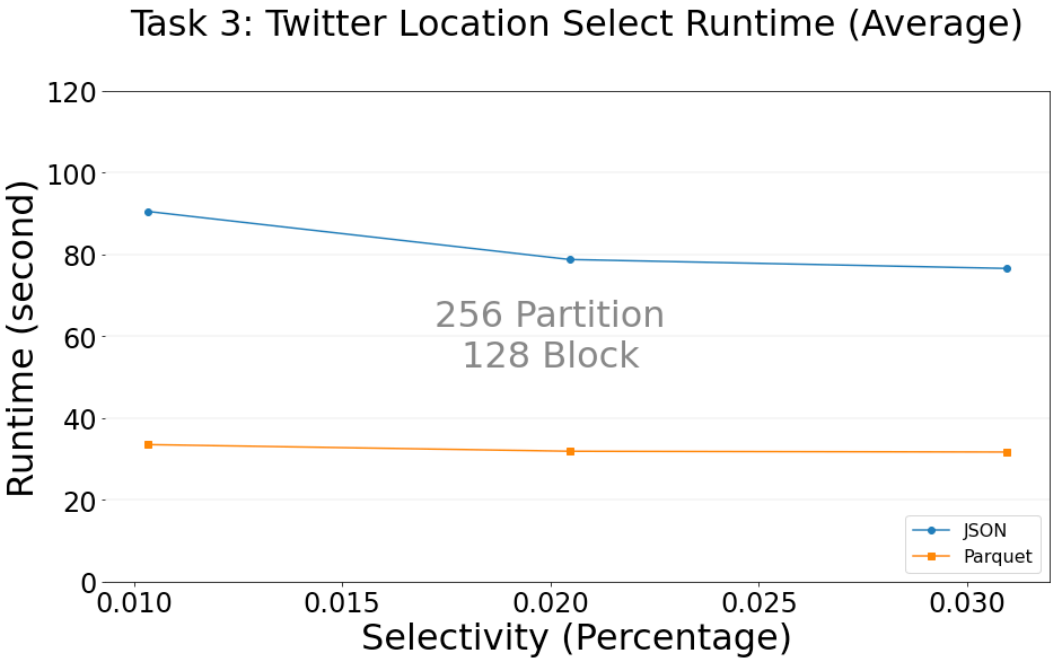


Figure 4.7: Task 3: Select a subset of tweets with location bounding box

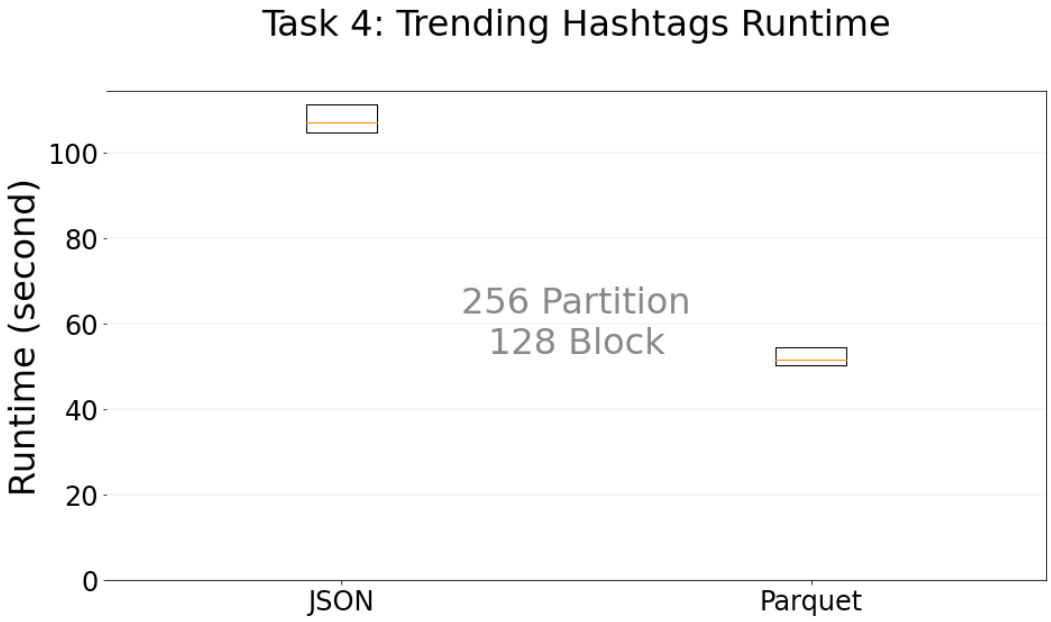


Figure 4.8: Task 4: Generate Trending Hashtags

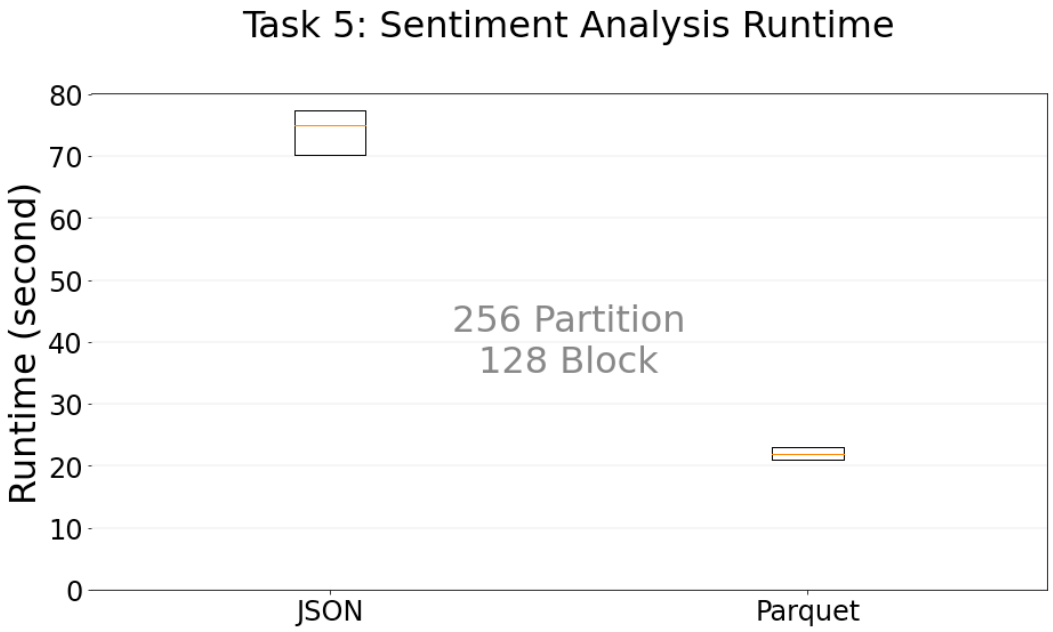


Figure 4.9: Task 5: Sentiment Analysis

4.6.1 Partitioning

Each collection is paired with a plaintext JSON file in the original file structure for our Twitter data. After Spark processing and conversion, each collection file will be partitioned to meet processing requirements. Due to the fact that each WARC file in our collection is limited to 1 GB in size, we can do the one-to-one mapping from WARC to Parquet online archive as in Chapter 2. In the case of the Twitter collection, there is no file size restriction, which is unfavorable to the parallelism of distributed computation. In order to regulate the file size while developing the new input data, we must divide up large files into smaller ones. During Spark processing, this procedure is referred to as partitioning, where the number of partitions matches the number of parallel tasks in Spark and the number of output files for each input.

Figures 4.10 and 4.11 show the performance comparison for Task 4 and Task 5 with data under different partition settings under Snappy compression. For JSON data, the default 256k setting performs best for Tasks 4 and 5. A greater partition setting for Parquet data indicates that the lower bound performance can be enhanced with a larger partition size. We believe this performance gain is the result of improved data loading and caching for larger data files. The lower bound performance relates more to the warm task start situation, in which the identical task has been executed within the system in the recent past.

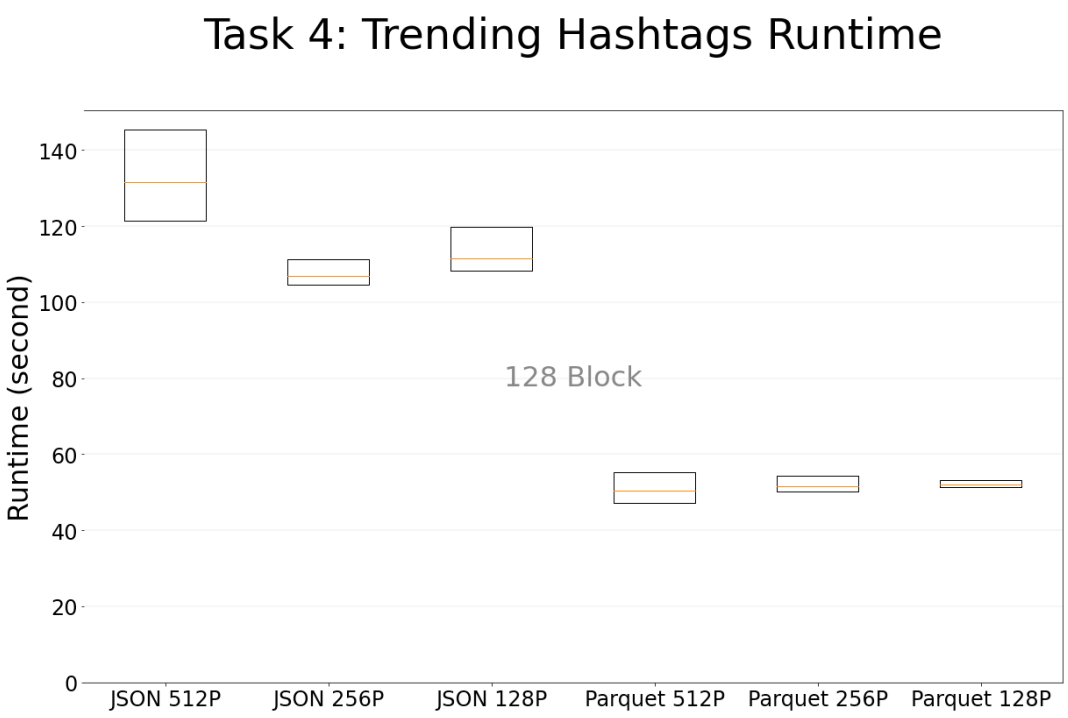


Figure 4.10: Task 4: Generate Trending Hashtags with different partition settings.

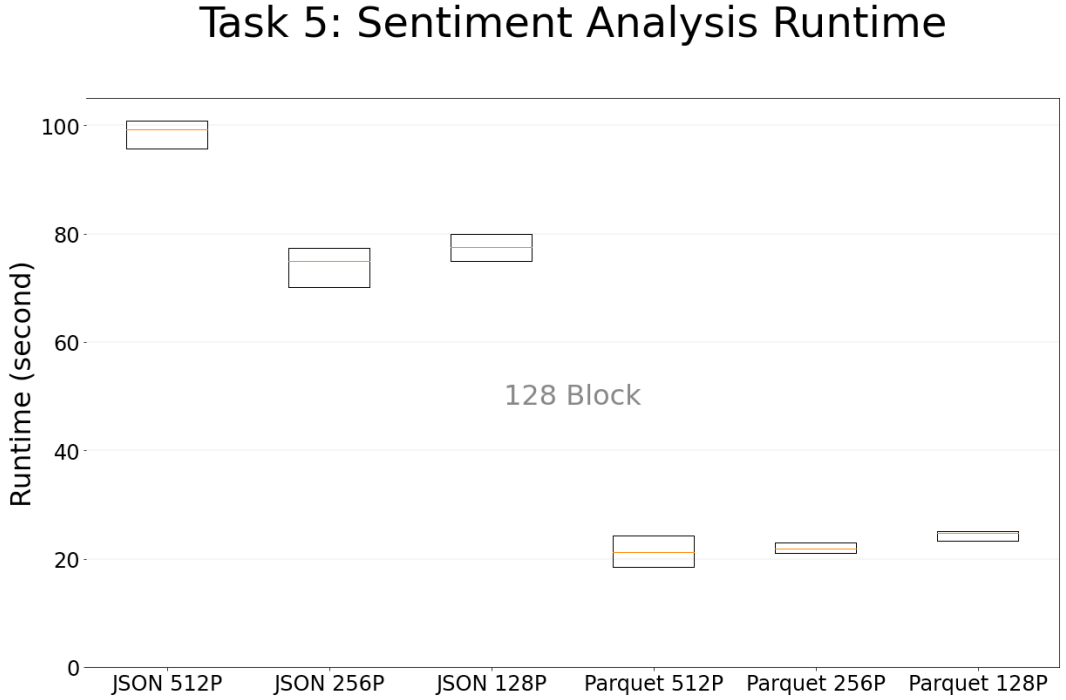


Figure 4.11: Task 5: Sentiment Analysis with different partitions.

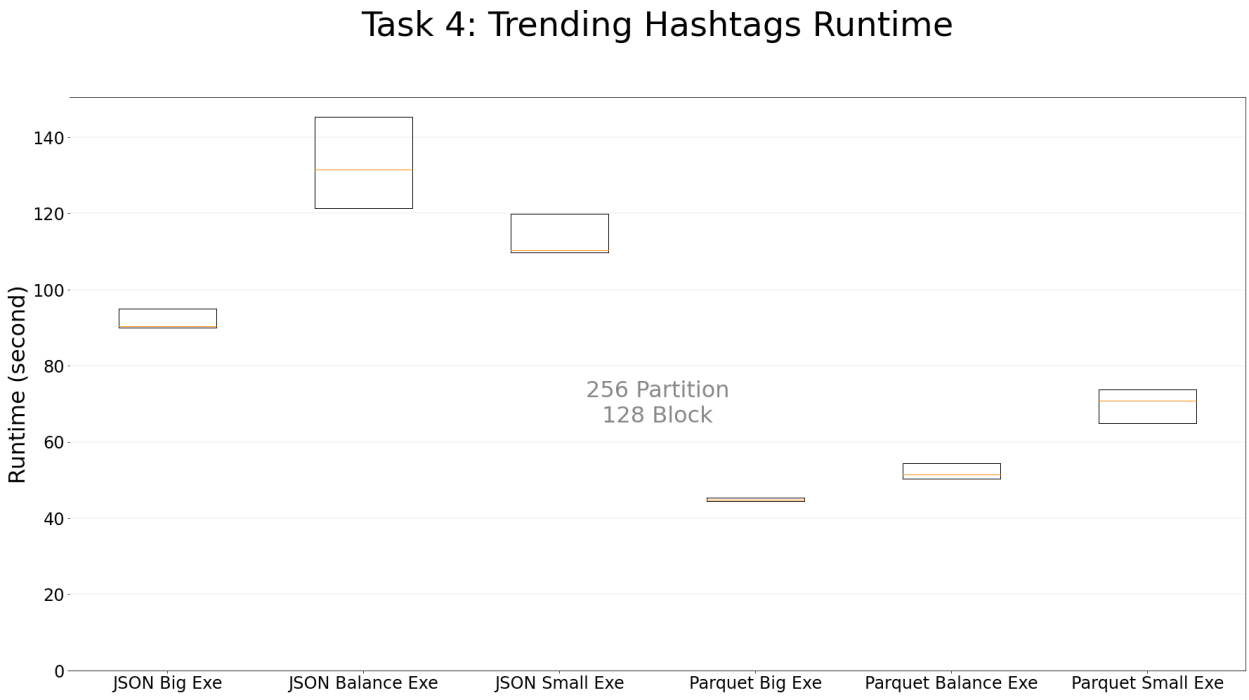


Figure 4.12: Task 4: Generate Trending Hashtags with different executor settings.

4.6.2 Computation Resource Allocation

Similar to the performance benchmark in Section 3.4.4 in Chapter 3, we explore the performance difference with the big, balanced, and small executor resource allocation strategy with Spark processing as shown in Figures 4.12 and 4.13. We note that the huge executor plan generally provides the best performance for both JSON and Parquet data, with greater data locality gained by resource allocation in this configuration.

4.6.3 Parquet Row Group Size

In Parquet data format, row group size has the ability to impact processing performance because it is directly related to the data loading structure within Parquet data, as described in Section 2.4.5. Particularly, we focus on tasks involving a large amount of data extraction for which the row group size option makes a difference. Consequently, we investigate the performance of Parquet-formatted Twitter data with various row group size choices for Tasks 1, 2, and 3. The results are shown in Figures 4.14, 4.15, and 4.16. In general, the 64 MB row group size achieves the most balanced performance across all three activities. The severe underperformance of the 128-row-group size in Tasks 1 and 2 suggests that the large row group size brings substantial data loading into the system. Due to the modest selectivity levels in Task 3, comparing with Task 1 and 2, the outcomes are not significantly different.

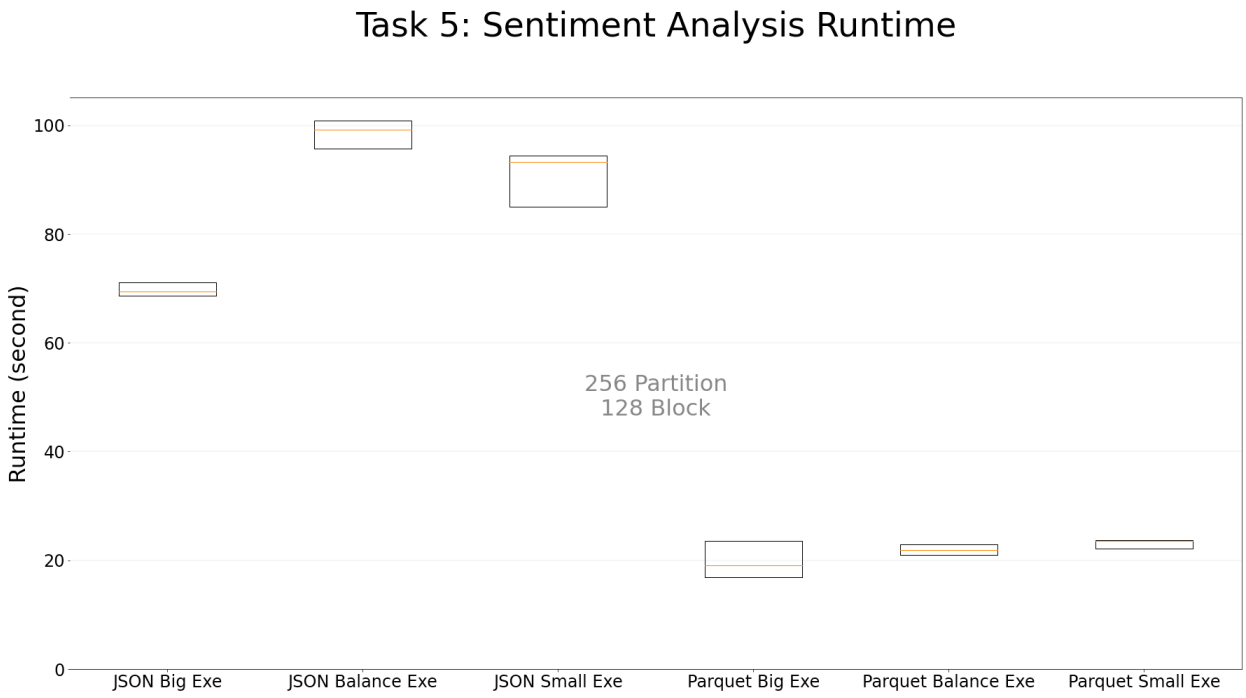


Figure 4.13: Task 5: Sentiment Analysis for each tweet with different executor strategies.

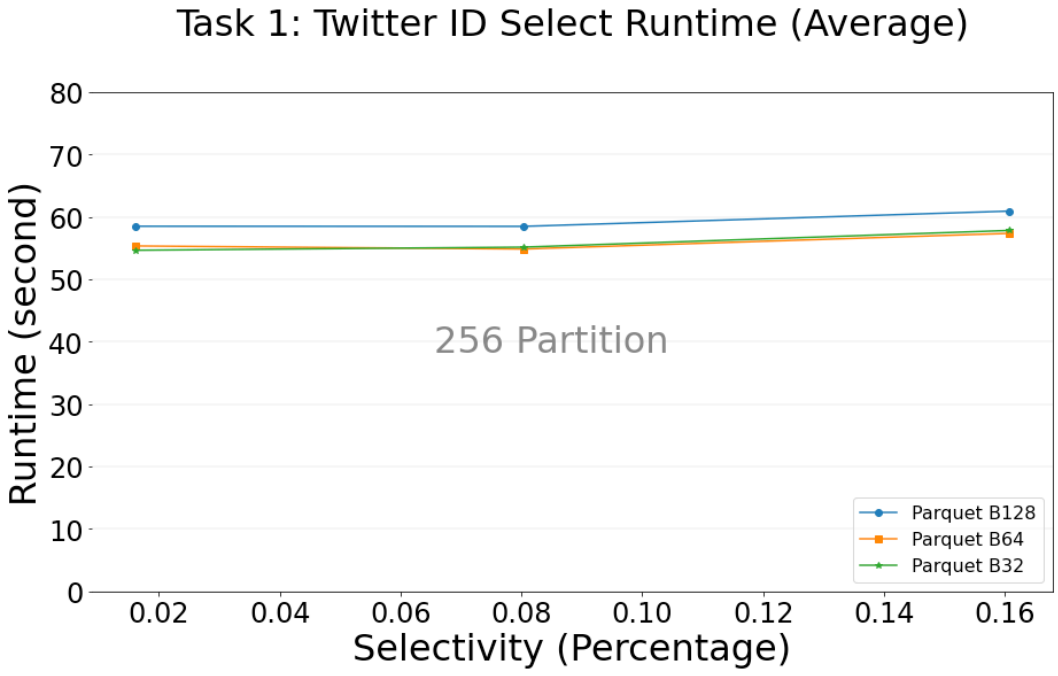


Figure 4.14: Task 1: Select a subset of tweets with a list of tweet IDs with different Parquet row group sizes

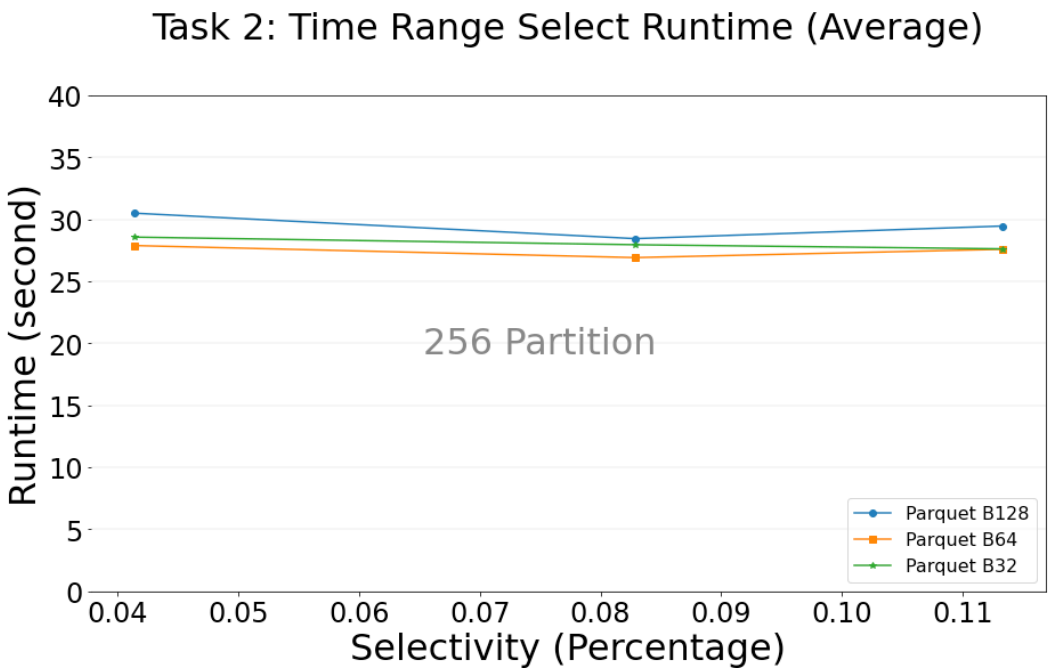


Figure 4.15: Task 2: Select a subset of tweets with a time range with different Parquet row group sizes

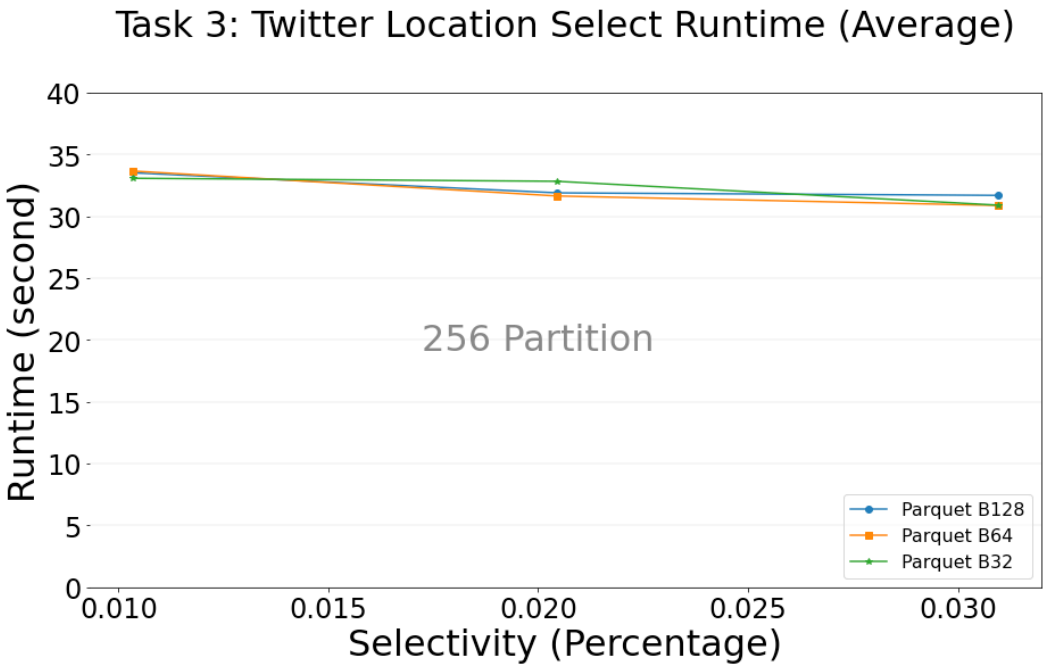


Figure 4.16: Task 3: Select a subset of tweets with location bounding box with different Parquet row group sizes

4.6.4 Compression Method

Snappy compression is the default compression algorithm for Parquet data format in Spark [13]. However, in Chapters 2 and 3, we find that Gzip is a better compression format than Snappy for Parquet web archive data. Here, we will also compare the compression algorithms Snappy and Gzip for the Twitter data case as shown in Figures 4.17 - 4.21. According to the results, Gzip compression performs better across all workloads for the Parquet data format. In the case of the JSON format, Gzip is generally comparable to Snappy, and occasionally slightly superior. Considering compression and performance efficiency, we can infer that Gzip should be the compression method of choice for Twitter data.

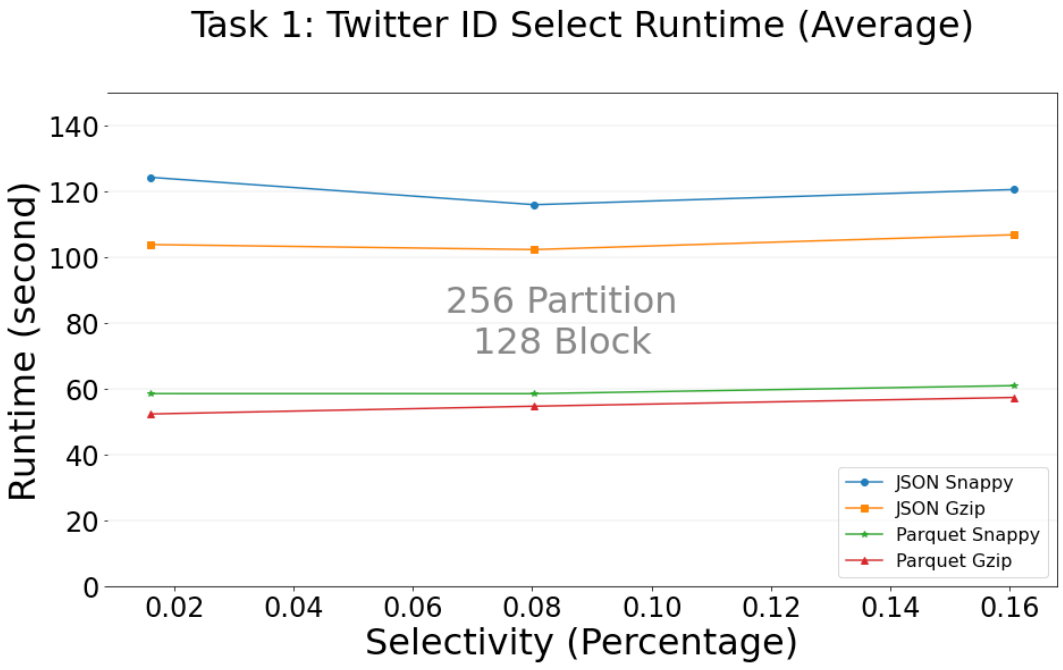


Figure 4.17: Task 1: Select a subset of tweets with a list of tweet IDs with different compression methods

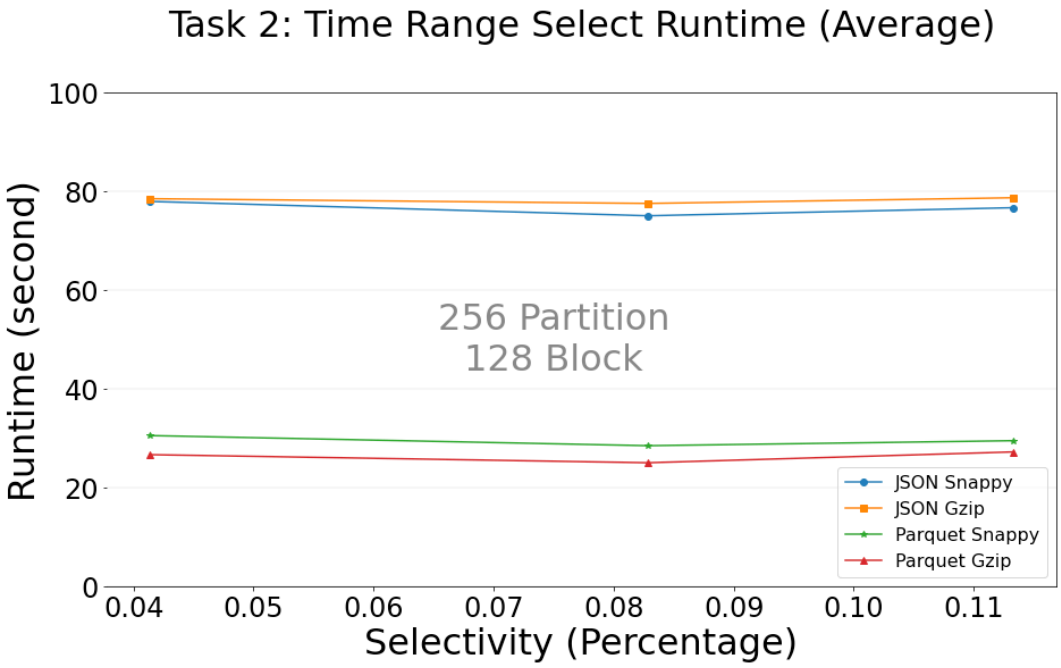


Figure 4.18: Task 2: Select a subset of tweets with a time range with different compression methods

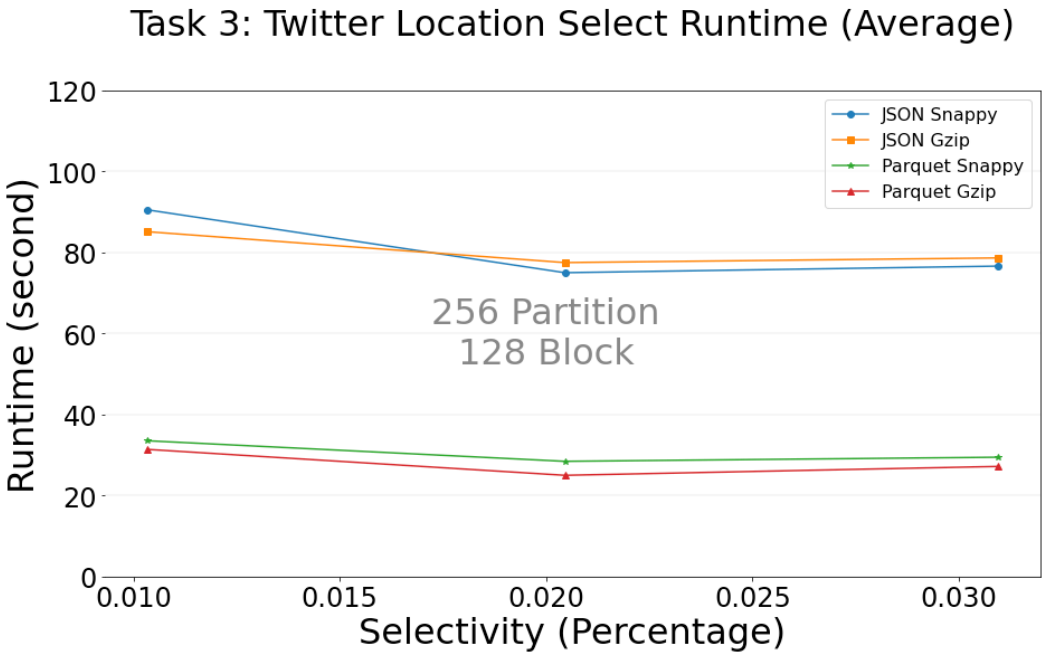


Figure 4.19: Task 3: Select a subset of tweets with location bounding box with different compression methods

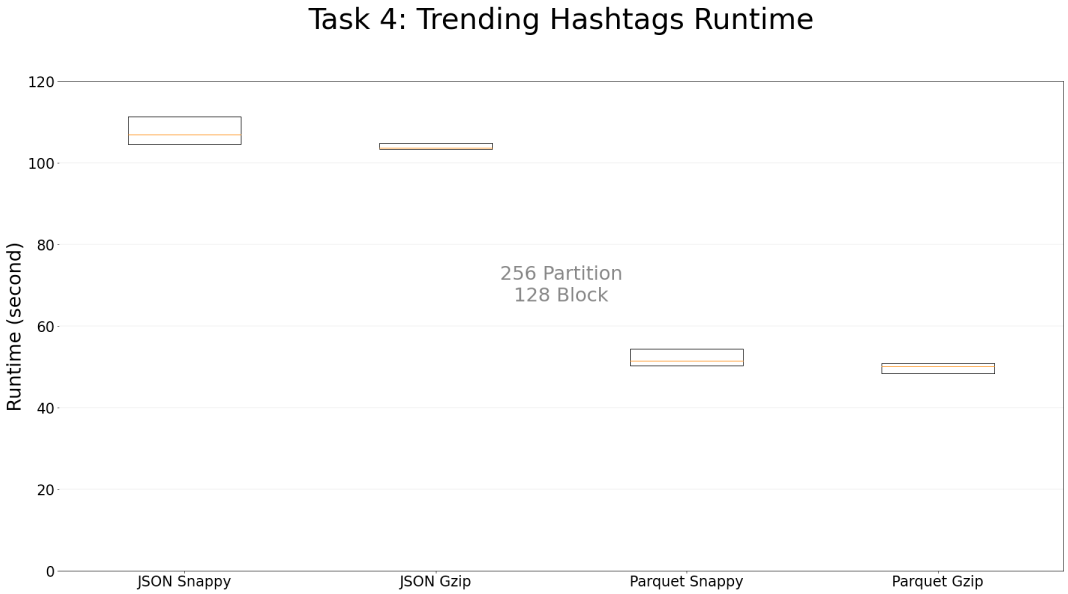


Figure 4.20: Task 4: Generate Trending Hashtags with different compression methods

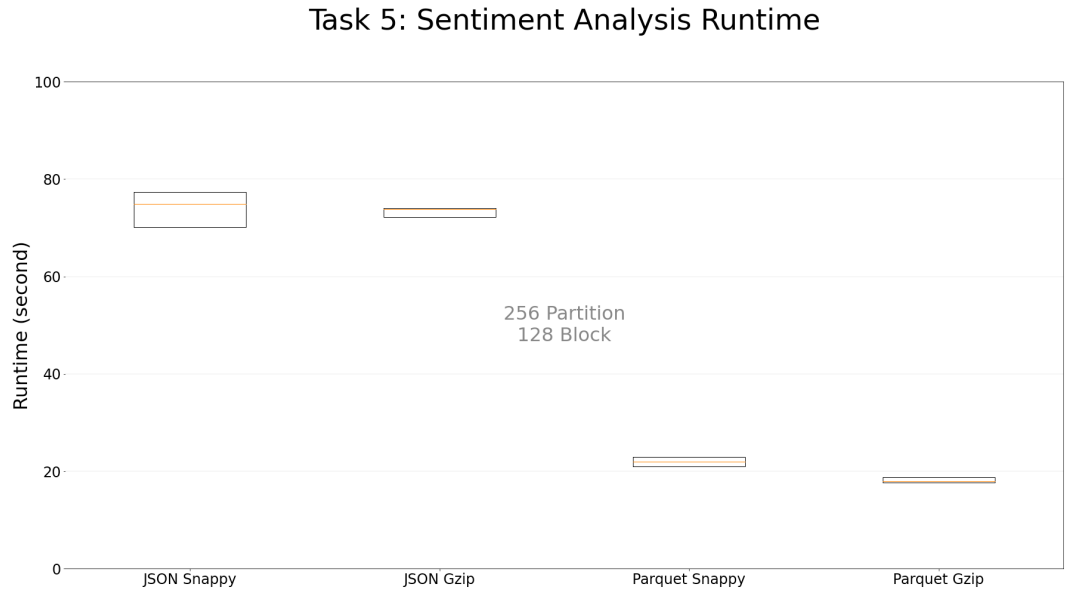


Figure 4.21: Task 5: Sentiment Analysis with different compression methods

4.6.5 Web Hosting

Figure 4.22 shows the benchmark results for Task 6. We use the huge executor resource allocation strategy as mentioned in Table 3.2 from Section 3.4.4. As we described in Section 3.4.4, we adopt FAIR as the default job scheduling method . On the client side, we set the workload with 20 concurrent clients. We also analyze the performance difference between 32, 64, and 128 MB Parquet row group sizes. According to the benchmark results, Task 6 may achieve a consistent performance of approximately around 0.5 requests per second with a response time of 40 seconds. Under three row group size choices, the 32 MB one achieves the best performance. The results indicate that the infrastructure may take use of the Parquet predicate pushdown optimization with a reduced row group size setting, resulting in more efficient data scanning and reading. Overall, we think the general performance of Task 6 illustrates that this is still a challenging task to offer the tweet hosting via raw Parquet data.

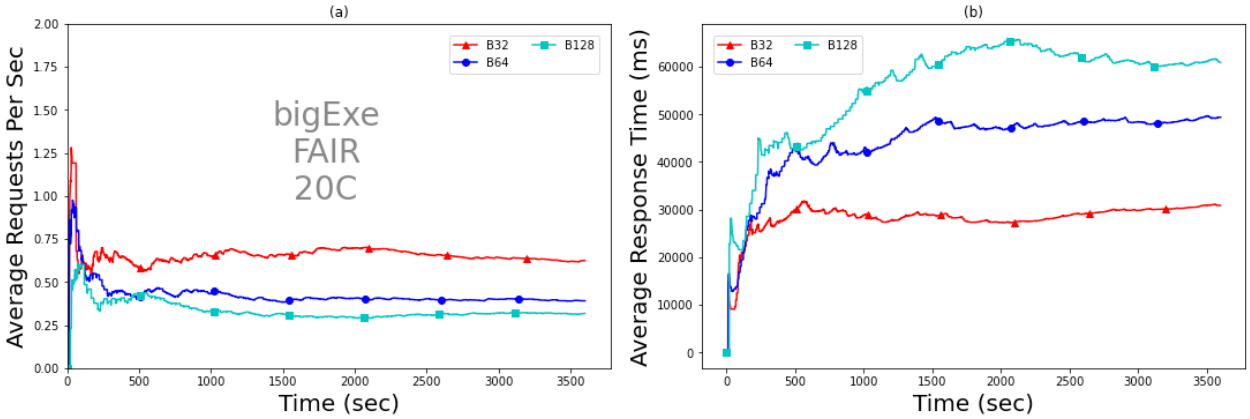


Figure 4.22: Task 6: Web hosting performance benchmark for individual tweet retrieval via tweet ID

4.7 Conclusions and Discussion

This chapter explains how Twitter data collections can be integrated into our web archive data infrastructure. We propose a single Twitter data structure that can incorporate all types of Twitter data sources and uses the common file storage language JSON for archiving. Then, we propose to replace the traditional JSON format with the Parquet data format, which we advocate for use in our proposed infrastructure for processing huge data. In order to demonstrate the performance differences between standard Twitter data storage in JSON format and the Parquet data format within our proposed infrastructure, we execute comparable analytical workload benchmarking tasks as in Chapter 2. For typical Twitter analytical workloads, our assessment demonstrates that the Parquet data format offers significantly more processing and storage efficiency than the usual JSON data format. Consequently, we

can incorporate Twitter data into our web archive big data infrastructure without requiring an additional component. With Twitter and traditional web archive data, users can utilize a consistent logic and toolset to execute analytical tasks more efficiently. We believe that utilizing both data sources may drive more innovation in research and instruction. On the side of the service provider, such an infrastructure design brings low service complexity and expense while preserving outstanding flexibility. This chapter reveals further limitations of plain text data formats, such as JSON, as an archival format for data reuse workloads. While WARC is largely utilized by the web archiving community, JSON is extensively accepted in a variety of digital applications. However, as we have shown in Chapters 2-4, it would be wise to examine the data storage bottleneck for data formats such as JSON in multiple applications.

This chapter is supplemented by Appendix C, which provides more implementation details for the accompanying experiments.

Chapter 5

Best Practices

In this study, we have undertaken a complete analysis of how to improve the accessing and processing of large web data by resolving the data storage bottleneck. A key objective of this research is to aid university libraries in facilitating the reuse of large web data for research and instruction. In addition, we believe that our techniques may aid researchers and practitioners in addressing the associated issues. This chapter seeks to serve as a guide for aspiring library practitioners, and data consumers in general.

5.1 Introduction

The Virginia Tech University Library recognizes the value of web archive data preservation as well as making the collections available on a large scale, which includes normal web archive data and social media data from Twitter. The enormous expansion of online archive size makes it challenging to generalize data collections for both preservation and accessibility. Since web archive preservation remains the primary objective of many practitioners, web archive data generally remains in a simple ISO standard format [62]. Likewise, Twitter data are typically stored in JSON format. Online archives access, on the other hand, carries the burden that such basic formats impose. This study focuses on the difficulty of web data access and provides ideas for enhancing the efficiency of data reuse inside the university library infrastructure.

The web archive community has been continuously developing new technologies to address the data access difficulties posed by existing standards. Modern infrastructures for large data, such as Hadoop and Spark, play crucial roles in online archive service systems nowadays [59, 60, 96]. Instead of dealing with typical WARC or JSON-formatted data, our research adopts a novel approach in which a modern data storage standard is utilized to improve the efficiency of data access and reuse. Under the big data infrastructure, we discover that modern data formats such as Parquet may vastly improve the accessibility of large web archive collections at a low cost and with greater flexibility, which is advantageous for practitioners who engage in more data access and reuse activities. In the rest of this chapter, we explain how web archive and Twitter data practitioners can take use of our research findings.

5.2 Reasons to Adopt

To make the most of our discoveries, it is essential to recognize their advantages and benefits. Instead of preservation, this research focuses on access situations for big web archives and Twitter data reuse. Table 5.1 gives a list of the pros and cons.

Pros	Cons
Efficient data scanning operations	Separation from general preservation standard
Efficient analytic processing operations	Data conversion cost
Efficient storage (when dropping the original data)	Storage cost without dropping the original data
More compatible tools	Expensive data manipulation

Table 5.1: Pros and cons of adopting our design

Practitioners should find the right balance between preservation and access use cases. Our design would be more useful to the user if access and reuse activities weighed more than preservation. Typically, the primary concern between these two components is the expense of storage. If both original data and reformatted data are affordable in the storage, the user can secure all the benefits. Otherwise, the choice of whether to replace the original data would be critical. The user may alternatively opt for a compromise infrastructure that utilizes Parquet-formatted data exclusively for index or derived content.

5.3 Obtaining Parquet Formatted Data

This section demonstrates how to obtain a Parquet-formatted representation of web archive and Twitter data, by converting the original data using the containerized data conversion tool and the Spark application.

5.3.1 Containerized Conversion Tool

A Virginia Tech student group has created containerized data conversion tools for WARC and Twitter data [32]. This containerized utility provides a straightforward command line interface for converting the WARC and JSON Twitter formats. In this situation, the user needs configure the container environment, Docker for instance, to support the container application. The user then can follow the user manual section to generate the desired Parquet data.

This containerized application is designed specifically for independent local systems. Be advised that the current implementation of this tool is not suited for converting big quantities of data.

5.3.2 Large Scale Conversion with Hadoop and Spark

Hadoop with Spark is the ideal choice if you need to convert a considerable amount of data and your infrastructure has access to Hadoop or Spark. This method necessitates that the user has sufficient experience operating and building Spark-related apps. Spark provides strong performance on batch processing and simple ability to produce Parquet data[1]. The user can leverage such advantages to generate Parquet data at scale.

For web archive data, the ArchiveUnleashedToolkit[2](AUT) is a popular WARC data ingestion and analytical tool based on Spark. AUT provides a simple access interface to ingest WARC data. We implemented a modified ArchiveUnleashedToolkit for reformatting WARC to Parquet and Avro as demonstrated in Chapter 2. Users can follow the packages and scripts for Parquet/Avro file writing from our example code repository on GitHub[3]. Note that our sample code uses an outdated and modified version of AUT. AUT is a rapidly expanding project; users can check the most recent version of the project and use our sample code as a guide for performing comparable conversions.

For the Twitter data, Spark supports native data ingestion from sources in the JSON data format. Users may refer to our conversion code example in Appendix C to convert the Twitter data.

5.4 Schema Design

Schema design is essential for various use cases. The methods of data conversion described in Section 5.3 utilized the schema design described in Section 3.3.4 for Parquet web archive data, and Section 4.3.2 for Parquet Twitter data. We follow the CDX-9 standard with more fields for the web archive data. For the Twitter data, the schema design follows a subset of the Twitter data API standard. The user may also refer to Common Crawl's index server schema design[4], which facilitates index searches for web archive data in the Common Crawl. The schema design can significantly influence the data scan performance as we demonstrated in Section 2.4.5. Table 5.2 shows data field examples and corresponding predicates related to data distribution.

The user may keep the predicate pushdown optimization in mind when designing a new schema.

[1] https://spark.apache.org/docs/latest/sql-data-sources-parquet.html
[2] https://archivesunleashed.org/aut/
[3] https://github.com/xw0078/WebArchiveWithParquetAvro
[4] https://commoncrawl.org/2018/03/index-to-warc-files-and-urls-in-columnar-format/

Field	Value Distribution	Predicates
Timestamp	Linear	Min/Max Range
URL	Random and Sparse	Not Available
Domain Name	Limited Name Space	Dictionary
Twitter ID	Close to Linear	Min/Max Range

Table 5.2: How data fields in Parquet take advantage of predicate pushdown through predicates over data distribution. For the URL field, the predicate is not available for performance improvement due to the data distribution property.

5.5 Derived Content

For many applications, consumers may desire additional data fields for easier access. Such fields could contain content extracted from the original data – such as text (WAT) or links (WET), labeled Tweet sentiment – from the web archive payload. The Common Crawl website[5] provides a comprehensive description of formats such as WAT and WET. Similarly, in Parquet-formatted data and derived content, we recommend creating one or more independent Parquet files that connect to the original data rather than incorporating them into the converted Parquet data. Users may gain a number of benefits from such data separation:

- Avoid costly schema evolution: A Parquet file cannot be modified to change its data content. To apply a new schema, a new Parquet file must be produced, which might be expensive for massive data conversions.
- Data field optimization: As seen in Chapter 2, users can sort and rearrange derived data for optimal performance using the Parquet predicate pushdown and project pushdown functionalities.
- Simple file-based storage: File-based storage for derived data in Parquet is compatible with Parquet raw data, preserving the simplicity of the infrastructure infrastructure.
- Alternative databases: It is possible to utilize more efficient databases instead of file based derived content, such as HBase. In this case, the user can select a different storage layer for optimal efficiency.

Common Crawl is a good example of a real-world application of this infrastructure. For many years, Common Crawl has utilized a Parquet-based index and derived content store for public access. This research and Common Crawl's share the same key idea, which enables an efficient index scan for online archive material.

[5] https://commoncrawl.org/the-data/get-started/

5.6 Analytic Tools

We recommend two interactive coding tools for working with the web data with our suggested design: Jupyter Notebook[6] and Zeppelin Notebook[7].

The popular interactive programming environment, Jupyter, offers robust support for Python-based data analytics. Users can utilize Jupyter's PySpark or Scala support to execute large-scale data analysis with Spark processing.

Zeppelin Notebook is an alternative to Jupyter Notebook, with increased native support for distributed compute tools such as Spark, Hive, and SQL. We propose Zeppelin over Jupyter if the application runs on a Hadoop infrastructure, which could minimize configuration requirements for the analytical system significantly.

5.7 Recommendations for Infrastructure Alternatives

Our approach primarily concentrates on the modern distributed computing environment. Nonetheless, it is extensible to contexts ranging from standalone systems to cloud deployments. Here, we illustrate how various infrastructures can utilize our practices. Figure 5.1 shows a overview of the alternatives and related recommendations.

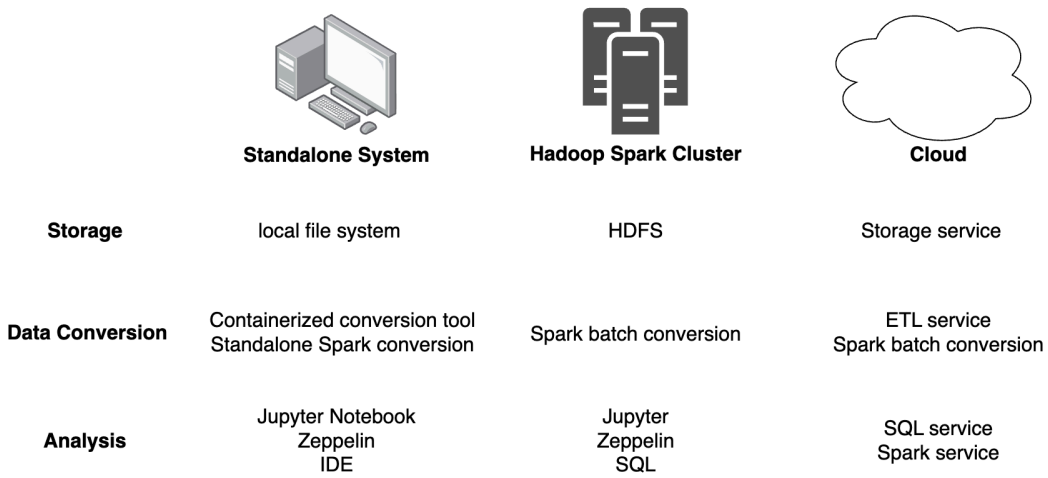


Figure 5.1: Infrastructures and practices

[6] https://jupyter.org/
[7] https://zeppelin.apache.org/

5.7.1 Standalone System

A standalone system can be a single laptop or desktop computer running a standard operating system, such as Linux, Windows, or MacOS. For data conversion, users can use the containerized conversion tool discussed in Section 5.3.1 to convert the WARC and JSON data to Parquet formatted data, for the greatest convenience. Following what is discussed in Section 5.3.2 with a standalone version of Spark[8] could also be a good alternative. For the analytical tool, we recommend using a Jupyter Notebook as in Section 5.6. Users can take advantage of the rich Python modules to work with Parquet formatted web data, with extensive flexibility.

5.7.2 Hadoop with Spark Cluster

Apache Hadoop with Spark is a widely-used, open-source, distributed computing platform, and it is also the primary platform we employ for our research. Hadoop with Spark delivers the finest large-scale accessibility to web archive and big data technologies if you have a suitable infrastructure.

Assuming all data are stored in HDFS, users can perform data conversion using the large scale conversion technique from Section 5.3.2. We recommend a Zeppelin Notebook as the interactive data analytical tool. The user may install the Zeppelin service on any node that has access to the complete cluster service, with minimal configuration effort. The Apache Hive[9] data warehouse may also be used to perform SQL-based data analysis over Parquet-formatted data, with users able to include Parquet data into Hive without modification.

If URL search is a significant aspect of a user's access activity, we suggest using Apache HBase to support URL search, as it provides efficient URL scanning performance, as demonstrated in Chapter 3. Similarly, other sparsely distributed string fields can benefit from the design of the HBase index.

Spark Optimization

Under the cluster infrastructure, Spark configuration is a significant factor that can impact the processing performance of diverse tasks. Spark configuration has two major components: resource allocation and job scheduling method. If the user's data is comparable to the Common Crawl web archive data and JSON-based Twitter data we use, then, as demonstrated in Chapters 3 and 4, we recommend adopting the big executor resource allocation plan for activities accessible to the general public. Regarding job scheduling, we recommend the FAIR scheduling method over the FIFO method. The user should also be aware of configurations

[8] https://spark.apache.org/docs/latest/spark-standalone.html
[9] https://hive.apache.org/

such as application cleaner, as described in Section 3.3.5, which may introduce fatal errors that prevent long-running applications such as web servers from functioning.

5.7.3 Cloud

Providers of cloud services such as Amazon AWS, Microsoft Azure, and Google GCP offer relevant services for Parquet-formatted data. Storage and processing components like Hadoop and Spark are also generally available in these cloud vendors; in such situations the recommendations of the previous subsection can apply.

Here, we will use AWS services as an illustration. File-based data are often saved using the Amazon S3 storage service[9]. Users can use the AWS Glue ETL service to convert data using Spark job support and the large-scale conversion technique from Section 5.3.2. Once the user obtains Parquet formatted data on S3, Amazon Athena[10] or Redshift[8] can be used to conduct SQL-based analyses on Parquet data for the analytical portion. Users may further add a Spark service to run and analyze data from S3 in a manner comparable to a local cluster environment, when conducting complicated data analysis. AWS also provides powerful NoSQL databases such as DynamoDB[7], which is similar to HBase. User may leverage relevant services to accelerate index access performance through the index database design.

Chapter 6

Contributions and Future Work

6.1 Contribution

This research introduces a big data processing infrastructure that can support huge archived web data, including web page archives and Twitter archives, to assist the data-intensive science surrounding the reuse of archival material, such as at Virginia Tech in University Libraries. We propose using the current distributed computing platform Hadoop as the foundation for the system infrastructure. We use the HDFS file system as the storage layer for our archival data storage, the Spark processing engine as the general processing layer to handle general analytical workload and information retrieval, and the HBase database system to assist with index storage, for quick search services such as URL matching. At the data level, we identify the performance bottleneck of standard online data storage formats and suggest to replace them with newer data formats such as Parquet. In addition, we conduct tests and make adjustments to the infrastructure in order to handle new data formats and optimize performance efficiency. Typical analytical and service workloads are utilized to evaluate the proposed system and storage format changes. The scalability, adaptability, and accessibility of our suggested strategy for repurposing vast quantities of archived online data are demonstrated. The specific contributions are detailed below:

1. On the level of data storage, we uncovered a serious processing efficiency limitation for huge web archive collections. We suggest utilizing modern distributed computation, alongside modern file-based formats, as the infrastructure to support online archive preservation, and to accelerate the data-to-insight cycle on data reuse. Under the Hadoop and Spark framework, we demonstrate that current data formats such as Parquet may greatly enhance the processing performance of large web archive analytical processing when compared to traditional data storage standards. This work is published in JCDL 2020 [112].

2. We designed and implemented a web archive data browsing service infrastructure. The service infrastructure is an expansion of the proposed infrastructure for large web archive data processing. We demonstrate that the infrastructure can deliver a consistent and stable browsing service over the raw file-based data storage formatted in Parquet. In contrast to complex web hosting services, our infrastructure design just stores index information in HBase and maintains a low-level of data replication. In

addition, the service infrastructure follows the same processing flow as offline analytical workloads, resulting in an incredibly simple service structure. We plan to publish this work soon in such publication venues as the Joint Conference on Digital Libraries (JCDL) or the Journal on Computing and Cultural Heritage (JOCCH).

3. We added Twitter data as a new source of archival data to our web archive system. We integrate the Twitter data storage into a standard JSON-formatted file-based structure. We recognize the limitation of existing JSON storage standards and convert Twitter data to the Parquet data format, comparable to how web archive data is stored. Our benchmark results suggest that Parquet Twitter data can significantly outperform JSON Twitter data for analytical workloads. We plan to publish this work in such publication venues as the International Conference on Theory and Practice of Digital Libraries (TPDL) or the Joint Conference on Digital Libraries (JCDL).

4. We summarized a best practice chapter to serve as a guide for aspiring library practitioners, and data consumers in general. We plan to publish this work at Web Archiving and Digital Libraries (WADL) workshops.

5. Our findings have also been applied to other research projects including vibration data [110] and ultrasonic data [64]. In spite of the fact that the primary focus of these projects is on data analysis, we are able to harness the efficiency of Parquet data by converting raw data to the Parquet data format, as we do in this research. In addition to web archive data, the vision for digital preservation at Virginia Tech's library includes data sources such as vibration and ultrasonic. In the future, we want to include such data into our system and run systematic experiments to improve processing efficiency.

6.2 Publications

A number of papers and posters have been published or accepted during my Ph.D. program, as listed below:

- Xinyue Wang and Zhiwu Xie. 2020. The Case For Alternative Web Archival Formats To Expedite The Data-To-Insight Cycle. In Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020 (JCDL '20). Association for Computing Machinery, New York, NY, USA, 177–186. https://doi.org/10.1145/3383583.3398542[112]

- Jin, Y., Wang, X., Fox, E.A., Xie, Z., Neogi, A., Mishra, R.S. and Wang, T. (2022), Numerically Trained Ultrasound AI for Monitoring Tool Degradation. Adv. Intell. Syst., 4: 2100215. https://doi.org/10.1002/aisy.202100215[64]

- X. Wang and Z. Xie, "Web Archive Analysis Using Hive and SparkSQL," 2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL), 2019, pp. 424-425, doi: 10.1109/JCDL.2019.00101[111]
- Xinyue Wang and Zhiwu Xie. 2018. Towards A Self-Learning Library For Vibration Data. In Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries (JCDL '18). Association for Computing Machinery, New York, NY, USA, 391–392. https://doi.org/10.1145/3197026.3203870[110]

6.3 Future Work

In this research, we have conducted a detailed analysis on improving large web data processing through solving the data storage bottleneck, which involves many technical details and findings. The primary goal of the research aims to help university libraries to facilitate large web data for research and teaching reuse cases. We notice that many university libraries are moving the computational infrastructure to a cloud environment. In future work, we plan to continue to investigate the web archive data access issues under the cloud environment. We believe our research can bring more value to the community, with cost efficient solutions involving cloud services.

Bibliography

[1] 540co. 2022. 540co/yourtwapperkeeper: Yourtwapperkeeper - archive your social media. Retrieved Sep. 24, 2022 from https://github.com/540co/yourTwapperKeeper

[2] Ioannis Alagiannis, Renata Borovica-Gajic, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2015. NoDB: Efficient Query Execution on Raw Data Files. *Commun. ACM* 58, 12 (Nov. 2015), 112–121. https://doi.org/10.1145/2830508

[3] Sawood Alam, Mat Kelly, and Michael L. Nelson. 2016. InterPlanetary Wayback: The Permanent Web Archive. In *Proceedings of the 16th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL '16)*. ACM, Newark, New Jersey, USA, 273–274. https://doi.org/10.1145/2910896.2925467

[4] Sawood Alam, Michael L. Nelson, Herbert Van de Sompel, Lyudmila L. Balakireva, Harihar Shankar, and David S. H. Rosenthal. 2016. Web Archive Profiling through CDX Summarization. *International Journal on Digital Libraries* 17, 3 (Sept. 2016), 223–238. https://doi.org/10.1007/s00799-016-0184-4

[5] Sawood Alam, Michele Weigle, Michael Nelson, Fernando Melo, Daniel Bicho, and Daniel Gomes. 2019. MementoMap Framework for Flexible and Adaptive Web Archive Profiling. In *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, 172–181. https://doi.org/10.1109/JCDL.2019.00033

[6] Fernando Amat, Burkhard Höckendorf, Yinan Wan, William C. Lemon, Katie McDole, and Philipp J. Keller. 2015. Efficient Processing and Analysis of Large-Scale Light-Sheet Microscopy Data. *Nature Protocols* 10, 11 (Nov. 2015), 1679–1696. https://doi.org/10.1038/nprot.2015.111

[7] Amazon. 2022. Amazon DynamoDB Documentation. Retrieved Sep. 24, 2022 from https://docs.aws.amazon.com/dynamodb/index.html

[8] Amazon. 2022. Amazon Redshift Documentation. Retrieved Sep. 24, 2022 from https://docs.aws.amazon.com/redshift/index.html

[9] Amazon. 2022. Amazon Simple Storage Service Documentation. Retrieved Sep. 24, 2022 from https://docs.aws.amazon.com/s3/index.html

[10] Amazon. 2022. What is Amazon Athena? Retrieved Sep. 24, 2022 from https://docs.aws.amazon.com/pdfs/athena/latest/ug/athena-ug.pdf#what-is

[11] Apache. 2020. Apache/Parquet-Format. The Apache Software Foundation. Retrieved Jan. 20, 2020 from https://github.com/apache/parquet-format

[12] Apache. 2020. *Avro*. Retrieved Jan. 20, 2020 from https://github.com/apache/avro

[13] Apache. 2022. Parquet files. Retrieved Sep. 24, 2022 from https://spark.apache.org/docs/2.4.4/sql-data-sources-parquet.html

[14] Internet Archive. 2020. Internet Archive. Retrieved Jan. 20, 2020 from https://archive.org/

[15] Internet Archive. 2020. Wayback Machine. Retrieved Jan. 20, 2020 from https://archive.org/web/

[16] Internet Archive. 2021. Internet Archive: About IA. Retrieved Jan. 20, 2020 from https://archive.org/about/

[17] Michael Armbrust, Ali Ghodsi, Matei Zaharia, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, and Michael J. Franklin. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*. ACM Press, Melbourne, Victoria, Australia, 1383–1394. https://doi.org/10.1145/2723372.2742797

[18] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR* abs/1407.3561 (2014). arXiv:1407.3561 http://arxiv.org/abs/1407.3561

[19] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. 2005. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. RFC Editor. Retrieved Jul. 10, 2021 from http://www.rfc-editor.org/rfc/rfc3986.txt

[20] Aditya Bhardwaj, Vanraj, Ankit Kumar, Yogendra Narayan, and Pawan Kumar. 2015. Big data emerging technologies: A CaseStudy with analyzing Twitter data using Apache Hive. In *2015 2nd International Conference on Recent Advances in Engineering & Computational Sciences (RAECS)*. 1–6. https://doi.org/10.1109/RAECS.2015.7453400

[21] Kenneth Bialousz, Kevin Kokal, Kwamina Orleans-Pobee, and Christopher Wakeley. 2014. *Computational Linguistic Analysis of Earthquake Collections*. Technical Report. Virginia Tech. CS4984 course project, http://hdl.handle.net/10919/51132.

[22] Matthew Bock. 2017. A Framework for Hadoop Based Digital Libraries of Tweets. Virginia Tech, Department of Computer Science, Master's Thesis, http://hdl.handle.net/10919/78351.

[23] Dhruba Borthakur. 2007. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation. Retrieved Dec 6. 2022 from http://web.mit.edu/mriap/hadoop/hadoop-0.13.1/docs/hdfs_design.pdf

[24] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[25] Niels Brügger and Ian Milligan (Eds.). 2019. *The SAGE Handbook of Web History* (first ed.). SAGE Publications Ltd.

[26] Yusheng Cao, Reza Mazloom, and Makanjuola Ogunleye. 2020. *CS5604 (Information Retrieval) Fall 2020 Front-end (FE) Team Project*. Technical Report. Virginia Tech. CS5604 course project, http://hdl.handle.net/10919/101526.

[27] Josiah L. Carlson. 2013. *Redis in Action*. Manning Publications Co., USA.

[28] CCSDS. 2012. *Space Data and Information Transfer Systems — Open Archival Information System (OAIS) — Reference Model*. Standard ISO 14721:2012. International Organization for Standardization.

[29] International Internet Preservation Consortium. 2020. The CDX File Format. Retrieved Jan. 20, 2020 from https://iipc.github.io/warc-specifications/specifications/cdx-format/cdx-2006/

[30] International Internet Preservation Consortium. 2020. OpenWayback. Retrieved Jan. 20, 2020 from https://github.com/iipc/openwayback

[31] International Internet Preservation Consortium. 2020. OpenWayback CDXJ File Format. Retrieved Jan. 20, 2020 from https://iipc.github.io/warc-specifications/specifications/cdx-format/openwayback-cdxj/

[32] Matthew Coscia and Andrew Weber. 2022. *Parquet Containers*. Technical Report. Virginia Tech. CS4624 course project, http://hdl.handle.net/10919/109995.

[33] Miguel Costa, Daniel Gomes, and Mário J. Silva. 2017. The Evolution of Web Archiving. *International Journal on Digital Libraries* 18, 3 (Sept. 2017), 191–205. https://doi.org/10.1007/s00799-016-0171-9

[34] Common Crawl. 2020. Common Crawl. Retrieved Jan. 20, 2020 from https://commoncrawl.org/

[35] Tung Dao, Christopher Wakeley, and Liu Weigang. 2017. *Collection Management Webpages - Fall 2016 CS5604*. Technical Report. Virginia Tech. CS5604 course project, http://hdl.handle.net/10919/76675.

[36] H. Van de Sompel, M. Nelson, and R. Sanderson. 2013. *HTTP Framework for Time-Based Access to Resource States – Memento*. RFC 7089. Internet Engineering Task Force. https://doi.org/10.17487/RFC7089

[37] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[38] Pranav Dhakal, Yash Bhargava, Anna Herms, Kenneth Powell, and Daniel Burdisso. 2021. *Library Tweets Conversion*. Technical Report. Virginia Tech. CS4624 course project, http://hdl.handle.net/10919/107086.

[39] Viet Doan, Matt Crawford, Aki Nicholakos, Robert Rizzo, and Jackson Salopek. 2019. *Tourism Destination Websites*. Technical Report. Virginia Tech. CS4624 course project, http://hdl.handle.net/10919/92622.

[40] Christos Douligeris and Aikaterini Mitrokotsa. 2004. DDoS attacks and defense mechanisms: classification and state-of-the-art. *Computer Networks* 44, 5 (2004), 643–666. https://doi.org/10.1016/j.comnet.2003.10.003

[41] Mohamed M. G. Farag, Sunshin Lee, and Edward A. Fox. 2018. Focused crawler for events. *International Journal on Digital Libraries* 19, 1 (March 2018), 3–19. https://doi.org/10.1007/s00799-016-0207-1

[42] Md. Nowraj Farhan, Md. Ahsan Habib, and Md. Arshad Ali. 2018. A study and Performance Comparison of MapReduce and Apache Spark on Twitter Data on Hadoop Cluster. *International Journal of Information Technology and Computer Science* (2018).

[43] Zeon Trevor Fernando, Ivana Marenzi, and Wolfgang Nejdl. 2018. ArchiveWeb: Collaboratively Extending and Exploring Web Archive Collections—How Would You like to Work with Your Collections? *International Journal on Digital Libraries* 19, 1 (March 2018), 39–55. https://doi.org/10.1007/s00799-016-0206-2

[44] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. 1999. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Internet Engineering Task Force. https://doi.org/10.17487/RFC2616

[45] Fiverr. 2022. 24 best wayback machine services to buy online | Fiverr. Retrieved Sep. 24, 2022 from https://www.fiverr.com/gigs/wayback-machine

[46] Edward A. Fox, Marcos André Gonçalves, and Rao Shen. 2012. Introduction. In *Theoretical Foundations for Digital Libraries: The 5S (Societies, Scenarios, Spaces, Structures, Streams) Approach*, Edward A. Fox, Marcos André Gonçalves, and Rao Shen (Eds.). Springer International Publishing, Cham, 1–42. https://doi.org/10.1007/978-3-031-02279-1_1

[47] Ned Freed and Nathaniel S. Borenstein. 1996. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046. Internet Engineering Task Force. https://doi.org/10.17487/RFC2046

[48] Jeremy Freeman, Nikita Vladimirov, Takashi Kawashima, Yu Mu, Nicholas J. Sofroniew, Davis V. Bennett, Joshua Rosen, Chao-Tsung Yang, Loren L. Looger, and Misha B. Ahrens. 2014. Mapping Brain Activity at Scale with Cluster Computing. *Nature Methods* 11, 9 (Sept. 2014), 941–950. https://doi.org/10.1038/nmeth.3041

[49] Gevent. 2021. Gevent. Retrieved Sep. 24, 2022 from https://github.com/gevent/gevent

[50] Vinay Goel. 2011. Web Archive Metadata File Specification. Retrieved Jan. 20, 2020 from https://webarchive.jira.com/wiki/spaces/Iresearch/pages/13467719/Web+Archive+Metadata+File+Specification

[51] gojomo. 2011. Class SURT. Retrieved Jul. 10, 2021 from http://crawler.archive.org/apidocs/org/archive/util/SURT.html

[52] Daniel Gomes, João Miranda, and Miguel Costa. 2011. A Survey on Web Archiving Initiatives. In *Research and Advanced Technology for Digital Libraries*, Stefan Gradmann, Francesca Borri, Carlo Meghini, and Heiko Schuldt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 408–420.

[53] Shawn Graham, Ian Milligan, and Scott Weingart. 2015. *Exploring Big Historical Data: The Historian's Macroscope* (reprint ed.). Imperial College Press, London.

[54] P. Greenfield, M. Droettboom, and E. Bray. 2015. ASDF: A New Data Format for Astronomy. *Astronomy and Computing* 12 (Sept. 2015), 240–251. https://doi.org/10.1016/j.ascom.2015.06.004

[55] Miguel Grinberg. 2018. *Flask web development: Developing web applications with Python*. O'Reilly Media, Inc.

[56] Object Management Group. 2012. *Common Object Request Broker Architecture*. Standard 3.3. Retrieved Dec. 10, 2022 from https://www.omg.org/spec/CORBA/3.3/

[57] HappyBase. 2021. HappyBase. Retrieved Sep. 24, 2022 from https://github.com/python-happybase/happybase

[58] Tony Hey, Stewart Tansley, and Kristin Tolle (Eds.). 2009. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington. http://research.microsoft.com/en-us/collaboration/fourthparadigm/

[59] Helge Holzmann, Vinay Goel, and Avishek Anand. 2016. ArchiveSpark: Efficient Web Archive Access, Extraction and Derivation. In *2016 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. ACM, New York, NY, USA, 83–92. https://doi.org/10.1145/2910896.2910902

[60] Helge Holzmann, Nick Ruest, Jefferson Bailey, Alex Dempsey, Samantha Fritz, Peggy Lee, and Ian Milligan. 2022. ABCDEF - The 6 key features behind scalable, multi-tenant web archive processing with ARCH: Archive, Big Data, Concurrent, Distributed, Efficient, Flexible. In *2022 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. 1–11.

[61] Digital Methods Initiative. 2022. Digital Methods Initiative - Twitter capture and Analysis Toolset. Retrieved Sep. 24, 2022 from https://github.com/digitalmethodsinitiative/dmi-tcat

[62] ISO. 2009. *Information and Documentation - WARC File Format*. Standard ISO 28500:2009. International Organization for Standardization.

[63] Ian Jacobs and Norman Walsh. 2004. Architecture of the World Wide Web, Volume One. W3C Recommendation 15 December 2004. *World Wide Web Consortium* (2004). http://www.w3.org/TR/webarch/

[64] Yuqi Jin, Xinyue Wang, Edward A. Fox, Zhiwu Xie, Arup Neogi, Rajiv S. Mishra, and Tianhao Wang. 2022. Numerically Trained Ultrasound AI for Monitoring Tool Degradation. *Advanced Intelligent Systems* 4, 5 (2022), 2100215. https://doi.org/10.1002/aisy.202100215

[65] Rashid Kamal, Munam Ali Shah, Asad Hanif, and J Ahmad. 2017. Real-time opinion mining of Twitter data using spring XD and Hadoop. In *2017 23rd International Conference on Automation and Computing (ICAC)*. 1–4. https://doi.org/10.23919/IConAC.2017.8082091

[66] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1119–1130. https://doi.org/10.14778/2732977.2732986

[67] Andrea Kavanaugh, Edward Fox, Steven Sheetz, Seungwon Yang, Lin Li, Travis Whalen, Donald Shoemaker, Apostel Nastev, and Lexing Xie. 2011. Social Media for Cities, Counties and Communities. *Final Grant Report to VT CCSR* (March 2011).

[68] Andrea L. Kavanaugh, Steven D. Sheetz, Rodrigo Sandoval-Almazan, John C. Tedesco, and Edward A. Fox. 2016. Media use during conflicts: Information seeking and political efficacy during the 2012 Mexican elections. *Government Information Quarterly* 33, 3 (2016), 595–602. https://doi.org/10.1016/j.giq.2016.01.004 Open and Smart Governments: Strategies, Tools, and Experiences.

[69] Martin Klein, Herbert Van de Sompel, Robert Sanderson, Harihar Shankar, Lyudmila Balakireva, Ke Zhou, and Richard Tobin. 2014. Scholarly Context Not Found: One in Five Articles Suffers from Reference Rot. *PLOS ONE* 9, 12 (Dec. 2014), 115253. https://doi.org/10.1371/journal.pone.0115253

[70] Michael Kofler. 2001. *What Is MySQL?* Apress, 3–19. https://doi.org/10.1007/978-1-4302-0853-2_1

[71] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proc. VLDB Endow.* 10, 10 (June 2017), 1118–1129. https://doi.org/10.14778/3115404.3115416

[72] Jimmy Lin, Milad Gholami, and Jinfeng Rao. 2014. Infrastructure for Supporting Exploration and Discovery in Web Archives. In *Proceedings of the 23rd International Conference on World Wide Web*. ACM, New York, NY, USA, 851–856. https://doi.org/10.1145/2567948.2579045

[73] Jimmy Lin, Ian Milligan, Jeremy Wiebe, and Alice Zhou. 2017. Warcbase: Scalable Analytics Infrastructure for Exploring Web Archives. *J. Comput. Cult. Herit.* 10, 4 (July 2017), 22:1–22:30. https://doi.org/10.1145/3097570

[74] Justin Littman, Dan Kerchner, Laura Wrubel, Aditya Dharne, Rajat Vij, Dan Chudnov, Rithvikmundra, Victor, Yonah Bromberg Gaber, Soomin Park, Nick Bearman, Dolsy Smith, Somanath304, Reed Underwood, Nicholas Dias, and Christie Peterson. 2020. *gwu-libraries/sfm-ui: Version 2.3.0*. Zenodo. https://doi.org/10.5281/ZENODO.597278

[75] Jialin Liu, Evan Racah, Quincey Koziol, Shane Canon, Alex Gittens, Lisa Gerhardt, Surendra Byna, Mike F Ringenburg, and Mr Prabhat. 2016. H5Spark: Bridging the I/O Gap between Spark and Scientific Data Formats on HPC Systems. In *CUG2016 Proceedings*. Cray User Group, London, England, UK.

[76] Locust. 2022. Locust.io. Retrieved Sep. 24, 2022 from https://locust.io/

[77] Madhavi Mallapragada. 2019. Cultural Historiography of the 'Homepage'. In *The SAGE Handbook of Web History*. SAGE Publications Ltd, 387–399. https://doi.org/10.4135/9781526470546

[78] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, André Schumacher, Anthony D. Joseph, and David A. Patterson. 2013. *Adam: Genomics Formats and Processing Patterns for Cloud Scale Computing*. University of California, Berkeley Technical Report UCB/EECS-2013. UCB/EECS-2013-207, EECS Department, University of California, Berkeley.

[79] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[80] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2015. MLlib: Machine Learning in Apache Spark. (May 2015). arXiv:1505.06807

[81] Stephen Merity. 2014. Navigating the WARC File Format. Retrieved Dec. 10, 2022 from https://commoncrawl.org/2014/04/navigating-the-warc-file-format/

[82] Sun Microsystems. 1988. *RPC: Remote Procedure Call Protocol specification*. RFC 1050. Internet Engineering Task Force. https://doi.org/10.17487/RFC1050

[83] Ian Milligan. 2019. *History in the Age of Abundance?: How the Web Is Transforming Historical Research*. McGill-Queen's University Press, Montreal.

[84] Jessica D. Mink. 2015. Astronomical Data Formats: What We Have and How We Got Here. *Astronomy and Computing* 12 (Sept. 2015), 128–132. https://doi.org/10.1016/j.ascom.2015.07.001

[85] Sebastian Nagel. 2018. Index to WARC Files and URLs in Columnar Format. Retrieved Jan. 20, 2020 from https://commoncrawl.org/2018/03/index-to-warc-files-and-urls-in-columnar-format/

[86] Nikolaos Nodarakis, Spyros Sioutas, Athanasios K. Tsakalidis, and Giannis Tzimas. 2016. Using Hadoop for Large Scale Analysis on Twitter: A Technical Report. *CoRR* abs/1602.01248 (2016). arXiv:1602.01248 http://arxiv.org/abs/1602.01248

[87] Frank Austin Nothaft, Matt Massie, Timothy Danford, Zhao Zhang, Uri Laserson, Carl Yeksigian, Jey Kottalam, Arun Ahuja, Jeff Hammerbacher, Michael Linderman, Michael J. Franklin, Anthony D. Joseph, and David A. Patterson. 2015. Rethinking Data-Intensive Science Using Scalable Analytics Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, Melbourne, Victoria, Australia, 631–646. https://doi.org/10.1145/2723372.2742787

[88] Pagefreezer. 2022. Monitoring and archiving solutions for online data. Retrieved Sep. 24, 2022 from https://www.pagefreezer.com/

[89] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *Proc. VLDB Endow.* 11, 11 (July 2018), 1576–1589. https://doi.org/10.14778/3236187.3236207

[90] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 35th SIGMOD International Conference on Management of Data - SIGMOD '09*. ACM Press, Providence, Rhode Island, USA, 165. https://doi.org/10.1145/1559845.1559865

[91] Julien Peloton, Christian Arnault, and Stéphane Plaszczynski. 2018. FITS Data Source for Apache Spark. *Computing and Software for Big Science* 2, 1 (Oct. 2018), 7. https://doi.org/10.1007/s41781-018-0014-z

[92] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 263–273.

[93] P. Resnick. 2001. *Internet Message Format*. RFC 2822. Internet Engineering Task Force. https://doi.org/10.17487/RFC2822

[94] Anisha P. Rodrigues and Niranjan N. Chiplunkar. 2018. Real-time Twitter data analysis using Hadoop ecosystem. *Cogent Engineering* 5, 1 (2018), 1534519. https://doi.org/10.1080/23311916.2018.1534519 arXiv:https://doi.org/10.1080/23311916.2018.1534519

[95] Nick Ruest, Jimmy Lin, Ian Milligan, and Samantha Fritz. 2020. The Archives Unleashed Project: Technology, Process, and Community to Improve Scholarly Access to Web Archives. *arXiv:2001.05399 [cs]* (Jan. 2020). arXiv:2001.05399 [cs]

[96] Nick Ruest, Jimmy Lin, Ian Milligan, and Samantha Fritz. 2020. The Archives Unleashed Project: Technology, Process, and Community to Improve Scholarly Access to Web Archives. In *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020* (Virtual Event, China) *(JCDL '20)*. Association for Computing Machinery, New York, NY, USA, 157–166. https://doi.org/10.1145/3383583.3398513

[97] Hany M. SalahEldeen and Michael L. Nelson. 2012. Losing My Revolution: How Many Resources Shared on Social Media Have Been Lost?. In *Theory and Practice of Digital Libraries (Lecture Notes in Computer Science)*, Panayiotis Zaphiris, George Buchanan, Edie Rasmussen, and Fernando Loizides (Eds.). Springer, Berlin, Heidelberg, 125–137. https://doi.org/10.1007/978-3-642-33290-6_14

[98] sam and raj. 2014. Thanks! we were writing up a response at the same time: The Wayback Machine Data...: Hacker news. Retrieved Sep 15, 2022 from https://news.ycombinator.com/item?id=7723726

[99] Steven D. Sheetz, Andrea L. Kavanaugh, Edward A. Fox, Riham Hassan, Seungwon Yang, Mohamed Magdy, and Donald J. Shoemaker. 2019. Information Uses and Gratifications Related to Crisis: Student Perceptions since the Egyptian Uprising. In *Proceedings of the 16th International Conference on Information Systems for Crisis Response and Management, València, Spain, May 19-22, 2019*, Zeno Franco, José J. González, and José H. Canós (Eds.). ISCRAM Association.

[100] Danya Shere, Ahmad Ayub, Rebecca Mueller, Lexi Fabian, and Akshat Shah. 2020. *US State Tourism Websites*. Technical Report. Virginia Tech. CS4624 course project, http://hdl.handle.net/10919/98257.

[101] Reza Shiftehfar. 2018. Uber's Big Data Platform: 100+ Petabytes with Minute Latency. Retrieved Jan. 20, 2020 from https://eng.uber.com/uber-big-data-platform/

[102] Statista. 2021. Twitter: number of users worldwide 2020. Retrieved Dec 6. 2021 from https://www.statista.com/statistics/303681/twitter-users-worldwide/

[103] Statista. 2021. User-generated internet content per minute 2021. Retrieved Dec 6. 2021 from https://www.statista.com/statistics/195140/new-user-generated-content-uploaded-by-users-per-minute/

[104] Stillo. 2022. Stillo. Retrieved Sep. 24, 2022 from https://www.stillio.com/wayback-machine-alternative

[105] Archives Unleashed Team. 2020. Archivesunleashed/Aut. Archives Unleashed. Retrieved Jan. 20, 2020 from https://github.com/archivesunleashed/aut

[106] B. Thomas, T. Jenness, F. Economou, P. Greenfield, P. Hirst, D. S. Berry, E. Bray, N. Gray, D. Muna, J. Turner, M. de Val-Borro, J. Santander-Vela, D. Shupe, J. Good, G. B. Berriman, S. Kitaeff, J. Fay, O. Laurino, A. Alexov, W. Landry, J. Masters, A. Brazier, R. Schaaf, K. Edwards, R. O. Redman, T. R. Marsh, O. Streicher, P. Norris, S. Pascual, M. Davie, M. Droettboom, T. Robitaille, R. Campana, A. Hagen, P. Hartogh, D. Klaes, M. W. Craig, and D. Homeier. 2015. Learning from FITS: Limitations in Use in Modern Astronomical Research. *Astronomy and Computing* 12 (2015), 133–145. https://doi.org/10.1016/j.ascom.2015.01.009

[107] Twitter. 2022. Data dictionary: Standard v1.1. Retrieved Dec. 18, 2022 from https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/tweet

[108] Herbert Van de Sompel, Michael L. Nelson, Robert Sanderson, Lyudmila Balakireva, Scott Ainsworth, and Harihar Shankar. 2009. Memento: Time Travel for the Web. (2009). arXiv:0911.1112

[109] Abhinav Verelly, Gruhn David, Ashutosh Bhattarai, and Shane Grishaw. 2021. *US State Tourism*. Technical Report. Virginia Tech. CS4624 course project, http://hdl.handle.net/10919/103269.

[110] Xinyue Wang and Zhiwu Xie. 2018. Towards A Self-Learning Library For Vibration Data. In *Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries* (Fort Worth, Texas, USA) *(JCDL '18)*. Association for Computing Machinery, New York, NY, USA, 391–392. https://doi.org/10.1145/3197026.3203870

[111] Xinyue Wang and Zhiwu Xie. 2019. Web Archive Analysis Using Hive and SparkSQL. In *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. 424–425. https://doi.org/10.1109/JCDL.2019.00101

[112] Xinyue Wang and Zhiwu Xie. 2020. The Case For Alternative Web Archival Formats To Expedite The Data-To-Insight Cycle. In *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020* (Virtual Event, China) *(JCDL '20)*. Association for Computing Machinery, New York, NY, USA, 177–186. https://doi.org/10.1145/3383583.3398542

[113] Matthew S. Weber. 2018. Methods and Approaches to Using Web Archives in Computational Communication Research. *Communication Methods and Measures* 12, 2-3 (April 2018), 200–215. https://doi.org/10.1080/19312458.2018.1447657

[114] Webrecorder. 2020. Pywb. Webrecorder. Retrieved Jan. 20, 2020 from https://github.com/webrecorder/pywb

[115] Zhiwu Xie and Edward A. Fox. 2017. Advancing Library Cyberinfrastructure for Big Data Sharing and Reuse. *Information Services & Use* 37, 3 (Jan. 2017), 319–323. https://doi.org/10.3233/ISU-170853

[116] Zhang Xuan, Huang Wei, Wang Ji, and Geng Tianyu. 2014. *Unsupervised Event Extraction from News and Twitter*. Technical Report. Virginia Tech. CS5604 course project, http://hdl.handle.net/10919/47954.

[117] Tian Yang, Kenjiro Taura, and Liu Chao. 2017. SDAC: Porting Scientific Data to Spark RDDs. In *Network and Parallel Computing (Lecture Notes in Computer Science)*, Xuanhua Shi, Hong An, Chao Wang, Mahmut Kandemir, and Hai Jin (Eds.). Springer International Publishing, 127–130.

[118] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10.

[119] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (July 2015), 1920–1948. https://doi.org/10.1109/TKDE.2015.2427795

Appendices

Appendix A

Supplement to Chapter 2

This appendix provides implementation details related to Chapter 2.

A.1 Example Code Snippets Used in the Experiments

A.1.1 Data Schema

We use the following schema to define the Parquet/Avro data structure for web archive data.

```
/*********************************************
  Parquet/Avro File Generation and Configurations
*********************************************/

// Generate Parquet file from WARC with customized AUT
val AUT_conversion_schema = StructType( // master schema for AU convertion
      List(
        StructField(name = "key", dataType = StringType, nullable = false),
        StructField(name = "surtUrl", dataType = StringType, nullable = false),
        StructField(name = "timestamp", dataType = StringType, nullable = false),
        StructField(name = "originalUrl", dataType = StringType, nullable =
        ↪  false),
        StructField(name = "mime", dataType = StringType, nullable = false),
        StructField(name = "status", dataType = StringType, nullable = false),
        StructField(name = "digest", dataType = StringType, nullable = false),
        StructField(name = "redirectUrl", dataType = StringType, nullable =
        ↪  false),
        StructField(name = "meta", dataType = StringType, nullable = false),
        StructField(name = "contentLength", dataType = LongType, nullable =
        ↪  false),
        StructField(name = "offset", dataType = LongType, nullable = false),
        StructField(name = "filename", dataType = StringType, nullable = false),
```

```
        StructField(name = "allheader", dataType = StringType, nullable = false),
        StructField(name = "payload", dataType = StringType, nullable = false)
    )
)
```

A.1.2 Data Conversion Implementation

The conversion from WARC formatted data to Parquet/Avro formatted data is implemented as:

```
/*********************************************
  ArchiveSpark (AS, WARC-CDX experiments) Version: 2.7.6
  ArchiveUnleashedToolkit (AUT, WARC experiments) Version: 0.17
*********************************************/

// parquet writing conf
sqlContext.setConf("parquet.dictionary.page.size", "5242880")
sqlContext.setConf("parquet.block.size", "33554432")
sqlContext.setConf("parquet.filter.statistics.enabled", "true")
sqlContext.setConf("parquet.filter.dictionary.enabled", "true")
sqlContext.setConf("spark.sql.parquet.outputTimestampType", "TIMESTAMP_MILLIS")
 ↪  // ensures correct timestamp format with predicate pushdown
sqlContext.setConf("spark.sql.parquet.enableVectorizedReader", "false")

// use AUT to convert WARC to Parquet/Avro
val warc_rdd = { RecordLoader.loadArchives(warcPath, sc)
             .map(r => Row(
             (r.getSurt+r.getOffset.toString+outFileName)
             ,r.getSurt
             ,r.getCrawlDate
             ,r.getUrl
             ,r.getMimeType
             ,r.getHttpStatus
             ,NulltoString(r.getDigest)
             ,r.getRedirect
             ,r.getMeta
             ,r.getLength
             ,r.getOffset
             ,outFileName
             ,NulltoString(r.getHeaderFields),NulltoString(r.getContentString)))
             }
val warc_df = sqlContext.createDataFrame(all_rdd,AUT_conversion_schema)
warc_df{
    .coalesce(1)
```

```
    .write
    .format("parquet")
    .option("compression","gzip")
    .mode("overwrite")
    .save(outPath)
}
// note that AUT "r.getCrawlDate" function does not extract exact unix timestamp from the WARC,
// for our experiments, we decide to extract the accurate timestamp from the WARC header as an addtion step so that we can get the timestamp type column.
// (this step can be optimized through modifying AUT to reduce multiple conversions)
val fixTimestamp = (s:String) => {
    if (s != null){
        //print(s.getClass.toString)
        val Pattern = "(2018-05-.*)".r
        Pattern.findFirstIn(s).getOrElse("null").replace("T"," ").replace("Z","")
    }
    else{
        "null"
    }
}
val fixTimestampUDF = udf(fixTimestamp)
val parquetDf = spark.read.format("parquet").schema(AU_conversion_schema).load("source path")
val fixedDf = parquetDf{
    .withColumn("timestamp",fixTimestampUDF(col("allheader")))
    .withColumn("timestamp",unix_timestamp($"timestamp", "yyyy-MM-dd HH:mm:ss"))
    .cast(TimestampType)
    .as("timestamp"))
    }
fixedDf{
    .coalesce(1)
    .write
    .format("parquet")
    .option("compression","gzip")
    .mode("overwrite")
    .save("destination path")
    }

// Avro Conversion instead of Parquet
data.coalesce(1).write.format("avro")
    .option("compression","gzip").mode("overwrite").save("destination_path")
```

A.2 Benchmark Implementation

The benchmark workloads for our experiments are implemented as:

```
/*********************************************
  Workload Part
  Data loading
*********************************************/
// Parquet / Avro
val df = spark.read.format("parquet").load("data_dir/*.parquet")
val df = spark.read.format("avro").load("data_dir/*.avro")
// AS
val records = ArchiveSpark.load(WarcCdxHdfsSpec(cdxPath,warcPath))
// AUT
val records = RecordLoader.loadArchives(warcPath, sc)

/*********************************************
  Data Count
*********************************************/
// Parquet,Avro
val count = df.count
// AS / AUT
val count = records.count

/*********************************************
  Timestamp Filtering
*********************************************/
// Parquet,Avro
val filtered = df.filter(col("timestamp") >
↪  unix_timestamp(lit(timeFrom)).cast("timestamp") && col("timestamp") <
↪  unix_timestamp(lit(timeTo)).cast("timestamp")) // INT64 timestamp with
↪  predicate pushdown
val filtered = df.withColumn("timestampStr",unix_timestamp($"timestampStr",
↪  "yyyy-MM-dd HH:mm:ss").cast("timestamp")).filter(col("timestampStr") >
↪  unix_timestamp(lit(timeFrom)).cast("timestamp") && col("timestampStr") <
↪  unix_timestamp(lit(timeTo)).cast("timestamp")) // string timestamp without
↪  predicate pushdown
// AS
val filtered = records.filter(r => r.time.isAfter(timeFrom) &&
↪  r.time.isBefore(timeTo)).enrich(StringContent)
// AUT: AUT only provides for year,month,day filtering
val filtered = RecordLoader.loadArchives(warcPath,
↪  sc).keepDate(List(time),ExtractDate.DateComponent.YYYYMMDD)

```

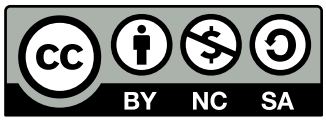
```
/*********************************************
  URL Filtering
*********************************************/
// Parquet,Avro
val filtered = df.filter($"originalUrl".isin(URLs:_*))
// Parquet (D-PP)
val filtered =
    df.filter($"domain".isin(domains:_*)).filter($"originalUrl".isin(URLs:_*))
// AS
val filtered = records.filter(r =>
    URLs.contains(r.originalUrl)).enrich(StringContent)
// AU
val filtered = RecordLoader.loadArchives(warcPath,
    sc).keepUrlPatterns(URLs.toSet)

/*********************************************
  Topic Modeling
  Reference Link:
  https://databricks-prod-cloudfront.cloud.databricks.com/
    public/4027ec902e239c93eaaa8714f173bcfc/
    3741049972324885/3783546674231782/
    4413065072037724/latest.html
*********************************************/
// extract puretext
import org.jsoup.Jsoup
val html_extractor = (strpayload:String) => {
    Jsoup.parse(strpayload).text().replaceAll("[\\r\\n]+", " ")
}
val text_extractorUDF = udf(html_extractor)
val puretext =
    df.withColumn("puretext",text_extractorUDF($"payload")).select("puretext") //
    Parquet,Avro
val puretext = records.enrich(HtmlText).map(r =>
    r.valueOrElse(HtmlText,"")).toDF("puretext") // AS
val puretext = records.map(r =>
    (RemoveHTML(r.getContentString))).toDF("puretext") // AU

// Apply LDA pipeline to extracted puretext
val indexedDf = puretext.withColumn("id",monotonicallyIncreasingId)
val tokenizer = new RegexTokenizer()
    .setPattern("[\\W_]+")
    .setMinTokenLength(4)
    .setInputCol("puretext")
    .setOutputCol("tokens")
val tokenized_df = tokenizer.transform(indexedDf)
```

```
val stopwords = sc.textFile("/path/to/stopwords").collect()
val remover = new StopWordsRemover()
    .setStopWords(stopwords)
    .setInputCol("tokens")
    .setOutputCol("filtered")
val filtered_df = remover.transform(tokenized_df)
val vectorizer = new CountVectorizer()
    .setInputCol("filtered")
    .setOutputCol("features")
    .setVocabSize(10000).setMinDF(5)
    .fit(filtered_df)
val countVectors = vectorizer.transform(filtered_df).select("id", "features")
val lda_countVector = countVectors.map { case Row(id: Long, countVector:
↪  org.apache.spark.ml.linalg.Vector) => (id, Vectors.fromML(countVector)) }.rdd
val numTopics = 20
val lda = new LDA().setOptimizer(new
↪  OnlineLDAOptimizer().setMiniBatchFraction(0.8)).setK(numTopics).setMaxIterations(3)
    .setDocConcentration(-1).setTopicConcentration(-1)
lda.run(lda_countVector)
```

Appendix B

Supplement to Chapter 3

This appendix provides implementation details related to Chapter 3.



B.1 Example Code Snippets Used in the Experiments

B.1.1 Flask Server Implementation

The Flask based demo web server is implemented as:

```
# defaults
import os
os.environ['SPARK_HOME'] =
↪  "/opt/cloudera/parcels/CDH-6.3.2-1.cdh6.3.2.p0.1605554/lib/spark"
from gevent import monkey
monkey.patch_all()
import findspark
findspark.init()
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.sql.functions import to_date
from pyspark import StorageLevel
from pyspark.conf import SparkConf
import index_search_methods
import importlib
import threading
importlib.reload(index_search_methods)
from werkzeug.routing import BaseConverter
import happybase
from surt import surt
```

```
21 import re
22 from gevent.pywsgi import WSGIServer
23 import gc
24 import random
25
26
27 def hbase_search(target_url):
28     connection = happybase.Connection(host='node11.dld.lan'
29                                       ,port=9090
30                                       ,compat='0.92'
31                                       ,transport='framed'
32                                       ,protocol='compact' )
33     connection.open()
34     table = connection.table('parquet_index')
35     surtUrl = surt(target_url)
36     result_itr = table.scan(row_prefix = (surtUrl+",").encode("UTF-8"))
37     result = []
38     for key, data in result_itr:
39         row = {}
40         row["timestamp"] = datetime \\
41             .utcfromtimestamp(int(data[b'parquet_gzip_surtIndex_one:crawltime']
               ↪  \\
42             .decode("UTF-8"))) \\
43             .strftime('%Y-%m-%d %H:%M:%S')
44         row["index"] = data[b'parquet_gzip_surtIndex_one:id'].decode("UTF-8")
45         row["filename"] =
           ↪  data[b'parquet_gzip_surtIndex_one:filename'].decode("UTF-8")
46         row['link'] = row["filename"]+"/"+re.sub(r'\
           ↪  |:',"-",row["timestamp"])+"/"+row["index"]
47         print(row['link'])
48         result.append(row)
49
50     return result
51
52
53
54
55 def scanEmptyCheck(itr):
56     for key, data in itr:
57         if key:
58             return 1
59         return 0
60
61 def parquet_payload_retrieve(filename,index):
62     print("File: ",filename)
```

```
63      print("ID: ",index)
64
65      df_load = \\
66          spark.read.parquet(
67          "/path/to/collection/"+filename)
68      target = df_load.filter(col("id")==int(index))
69      payload = target.select("payload")
70      result = payload.first()[0]
71      #spark.catalog.clearCache()
72      return result
73
74  # library.py
75  from flask import Flask, render_template, request, redirect, url_for, session
76  from datetime import datetime
77
78
79  app = Flask("Web Archive")
80
81  conf = pyspark.SparkConf().setAll([
82      ('spark.driver.memory','20g')
83      , ("spark.master", "yarn")
84      , ("spark.sql.parquet.columnarReaderBatchSize","1000")
85      , ('spark.sql.parquet.enableVectorizedReader','true')
86      , ("spark.scheduler.pool", "wa_pool")
87      , ("spark.scheduler.allocation.file", "/path/to/spark_fair_pool.xml")
88      , ("spark.cleaner.periodicGC.interval", "30min")
89      , ("spark.shuffle.service.enabled","true")
90      , ('spark.locality.wait','3s')
91      , ("spark.cleaner.referenceTracking.blocking", "true")
92      , ("spark.scheduler.mode", "FIFO")
93      , ('spark.executor.cores','5')
94      , ('spark.executor.memory','7G')
95      , ("spark.dynamicAllocation.enabled", "true")
96      , ("spark.speculation","false")
97    ])
98
99  spark =
    ↪  SparkSession.builder.appName("web-archive-parquet-server").config(conf=conf).getOrCreate()
    ↪  # single context for all jobs
100
101
102 hostname = "http://node11.dld.lan:9991/"
103
104 class RegexConverter(BaseConverter):
105     def __init__(self, map, *args):
```

```
106         self.map = map
107         self.regex = args[0]
108 app.url_map.converters['regex'] = RegexConverter
109 
110 #endpoint for test
111 @app.route('/test/<regex(".*"):pars>', methods=['GET'])
112 def test(pars):
113     print("Input pars: ",pars)
114     return render_template('searchHB.html',searched = 0)
115 
116 
117 @app.route('/search',methods=['GET', 'POST'])
118 def search():
119     if request.method == "POST":
120         target_url = request.form['url_search']
121         query = target_url
122         return redirect(url_for(".search",searched = 1,query=query))
123 
124     if 'searched' in request.args:
125         query = request.args["query"]
126         print("Input Query: ",query)
127         result = hbase_search(query)
128         if len(result) == 0:
129             no_result = 1
130         else:
131             no_result = 0
132         return render_template('searchHB.html',searched = 1,no_result=no_result,
            ↪  result=result,target_url=query,hostname=hostname)
133     else:
134         return render_template('searchHB.html',searched = 0)
135 
136 @app.route('/payload/<filename>/<timestamp>/<offset>',methods=['GET'])
137 def single_page_retrieve(filename, timestamp,offset):
138     return parquet_payload_retrieve(filename,offset)
139 
140 if __name__ == '__main__':
141     from werkzeug.serving import run_simple
142     http_server = WSGIServer(('node11.dld.lan', 9991), app)
143     http_server.serve_forever()
```

B.1.2 Locust Implementation

The client-side testing with Locust is implemented as:

```
import gevent
from gevent import monkey
monkey.patch_all(select=False)
import time
import random
from locust import HttpUser, TaskSet, task, constant, between
from locust import LoadTestShape
from locust import events
from locust.runners import STATE_STOPPING, STATE_STOPPED, STATE_CLEANUP,
    WorkerRunner
from csv import writer
import time
import os


out_filename =
    "50C_20DM_3LW_5EC_7EM_FIFO_0INS_1hBT_1CB_30mCI_0SPC_60EIT_JAVAGC_1VR1000BS.csv"
output_file = "custom_results/"+out_filename

if os.path.exists(output_file):
    os.remove(output_file)
    print(output_file+" removed")
else:
    print(output_file+" created")


def append_list_as_row(file_name, list_of_elem):
    # Open file in append mode
    with open(file_name, 'a+', newline='') as write_obj:
        # Create a writer object from csv module
        csv_writer = writer(write_obj)
        # Add contents of list as last row in the csv file
        csv_writer.writerow(list_of_elem)

def checker(environment):
    while not environment.runner.state in [STATE_STOPPING, STATE_STOPPED,
        STATE_CLEANUP]:
        time.sleep(1)
        if environment.runner.user_count != 0:
            content = [
                time.time()
                ,environment.runner.stats.total.avg_response_time
                ,environment.runner.stats.total.total_rps
                ,environment.runner.stats.total.current_rps
                ,environment.runner.stats.total
```

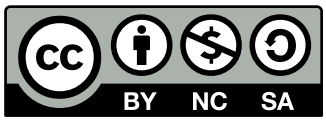
```
                    .get_current_response_time_percentile(0.95)
                ,environment.runner.user_count]
            append_list_as_row(output_file,content)

@events.init.add_listener
def on_locust_init(environment, **_kwargs):
    # only run this on master & standalone
    if not isinstance(environment.runner, WorkerRunner):
        gevent.spawn(checker, environment)

class UserTasks(TaskSet):

    def url_loader(self):
        with open("test_seeds/payload_10000_test.txt") as f:
            content = f.readlines()
        content = [x.rstrip() for x in content]
        return content

    def on_start(self):
        self.payload_retrieve10000 = self.url_loader()

    @task
    def random_index_search(self):
        self.client.get(random.choice(self.payload_retrieve10000))

class WebsiteUser(HttpUser):
    wait_time = constant(1)
    tasks = [UserTasks]


usersNum = int(out_filename.split("C_")[0])

class StagesShape(LoadTestShape):
    """
    A simply load test shape class that has different user and spawn_rate at
    different stages.
    Keyword arguments:
        stages -- A list of dicts, each representing a stage with the following
↪   keys:
            duration -- When this many seconds pass the test is advanced to the
↪   next stage
            users -- Total user count
            spawn_rate -- Number of users to start/stop per second
            stop -- A boolean that can stop that test at a specific stage
        stop_at_end -- Can be set to stop once all stages have run.
```

```
86      """
87
88      # duration should be continous
89
90      stages = [
91          {"duration": 3600, "users": usersNum, "spawn_rate": 10,"stop":False}
92      ]
93
94
95      def tick(self):
96          run_time = self.get_run_time()
97          for stage in self.stages:
98              if run_time < stage["duration"]+1:
99                  tick_data = (stage["users"], stage["spawn_rate"])
100                 return tick_data
101         return None
```

Appendix C

Supplement to Chapter 4

This appendix provides implementation details related to Chapter 4.



C.1 Example Code Snippets Used in the Experiments

C.1.1 Trending Hashtag

The trending hashtag generation task is implemented as:

```
import scala.math.pow
import java.lang.System.nanoTime
import java.time.LocalDateTime
import spark.implicits._
import org.apache.spark.sql.functions.countDistinct

val t0: Long = nanoTime
val in_file_name = "*.parquet"
val in_file_path = "/path/"
val df_parquet =  spark.read.format("parquet").load(in_file_path+in_file_name)
val filtered = {
    df_parquet
    .select("entities_hashtags")
    .select(explode_outer($"entities_hashtags").alias("uHt"))
    .groupBy("uHt").count().sort(col("count").desc)
}
val res = filtered.count()
```

C.1.2 Sentiment Analysis

The sentiment analysis task is implemented as:

```scala
import spark.implicits._
import com.johnsnowlabs.nlp.base.DocumentAssembler
import com.johnsnowlabs.nlp.annotator.UniversalSentenceEncoder
import com.johnsnowlabs.nlp.annotators.classifier.dl.SentimentDLModel
import org.apache.spark.ml.Pipeline
import scala.math.pow
import java.lang.System.nanoTime
import java.time.LocalDateTime
import org.apache.spark.sql.functions.countDistinct


val documentAssembler = new DocumentAssembler()
  .setInputCol("text")
  .setOutputCol("document")

val useEmbeddings =
↪  UniversalSentenceEncoder.load("/models/tfhub_use_en_2.4.0_2.4_1587136330099/")
    .setInputCols("document")
    .setOutputCol("sentence_embeddings")

val sentiment =
↪  SentimentDLModel.load("/models/sentimentdl_use_twitter_en_2.7.1_2.4_1610983524713")
  .setInputCols("sentence_embeddings")
  .setThreshold(0.7F)
  .setOutputCol("sentiment")

val pipeline = new Pipeline().setStages(Array(
  documentAssembler,
  useEmbeddings,
  sentiment
))

val in_file_name = "*.parquet"
val in_file_path = "/path/"
val df_parquet =  spark.read.format("parquet").load(in_file_path+in_file_name)
val df_filtered = df_parquet.select("text")
val pipelineModel = pipeline.fit(df_filtered)
val df_result = pipelineModel.transform(df_filtered)
val res = df_result.select("text", "sentiment.result").count()
```

C.1.3 Tweet ID Matching

The tweet ID matching task is implemented as:

```
import scala.math.pow
import java.lang.System.nanoTime
import java.time.LocalDateTime

val query_file = "1percent_id.parquet"
val query_df =  spark.read.format("parquet").load("/path/"+query_file)

val in_file_name = "*.parquet"
val in_file_path = "/path/"
val df_parquet =  spark.read.format("parquet").load(in_file_path+in_file_name)
val df_parquet_filtered = df_parquet.join(query_df, df_parquet("id_long") ===
↪  query_df("id_long"), "inner")
val res = df_parquet_filtered.count()
```

C.1.4 Timestamp Filtering

The timestamp filtering task is implemented as:

```
import scala.math.pow
import java.lang.System.nanoTime
import java.time.LocalDateTime

val timerange = Map("selectivity" -> "4percent", "timeFrom" -> "2017-10-28
↪  19:16:00", "timeTo" -> "2018-03-28 19:16:00")

val timeFrom = timerange("timeFrom")
val timeTo = timerange("timeTo")
val selectivity = timerange("selectivity")

val in_file_name = "*.parquet"
val in_file_path = "/path/"
val df_parquet =  spark.read.format("parquet").load(in_file_path+in_file_name)
val df_parquet_filtered = {
    df_parquet
    .filter(col("timestampUTC_int64").cast("timestamp") >
    ↪  unix_timestamp(lit(timeFrom)).cast("timestamp") &&
    ↪  col("timestampUTC_int64").cast("timestamp") <
    ↪  unix_timestamp(lit(timeTo)).cast("timestamp"))
}
val res = df_parquet_filtered.count()
```

C.1.5 Geo-location Filtering

The geo-location filtering task is implemented as:

```
val boxMap = Map("selectivity" -> "1percent", "size" -> 30D)

val vt_lat = 37.229098350784156
val vt_lon = -80.42346296500547
val right_lon = vt_lon + boxMap("size").asInstanceOf[Double]
val right_lat = vt_lat + boxMap("size").asInstanceOf[Double]
val left_lon = vt_lon - boxMap("size").asInstanceOf[Double]
val left_lat = vt_lat - boxMap("size").asInstanceOf[Double]

val in_file_name = "*.parquet"
val in_file_path = "/path/"
val df_parquet =  spark.read.format("parquet").load(in_file_path+in_file_name)
val df_parquet_filtered = {
    df_parquet
    .filter(col("geo.longitude") > left_lon && col("geo.longitude") < right_lon
    ↪  && col("geo.latitude") > left_lat && col("geo.latitude") < right_lat )
}
val res = df_parquet_filtered.count()
```