Final Project Report

ArchiveSpark

Instructor: Dr. Edward A. Fox
CS 5974 - Independent Study

Andrej Galad
agalad @ vt.edu
Virginia Polytechnic Institute and State University

Dec. 8, 2016

Table of Contents

List of Figures

List of Tables

1. Abstract

Without any doubt Web archives represent an invaluable source of information for researchers of various disciplines. However, in order to efficiently leverage them as a scholarly resource, the researchers typically have to first be able to process large datasets, filter down based on their needs, and quickly derive smaller datasets relevant for a particular study. As such, an archive-specific tool is needed that would be able to satisfy the above-mentioned requirements as well as other, non-functional criteria such as ease of use, extensibility, and reusability.

To address this problem researchers from Internet Archive [2], a San Francisco-based nonprofit memory institution, in collaboration with L3S Research Center [3], introduced *ArchiveSpark* [1] - a framework for efficient, distributed Web archive processing that builds a research corpus by working on existing and standardized data formats commonly held by Web archiving institutions. Unlike the existing state-of-the-art solutions, ArchiveSpark is capable of leveraging pre-generated archive metadata indexes, which results in a significant speed-up in terms of data processing and corpus extraction. Additionally, its inherent extensibility allows the user to write custom processing code leveraging 3rd party APIs and libraries to address domain-specific needs.

This project expands the work at the Internet Archive of researcher Vinay Goel and of Jefferson Bailey (co-PI on two NSF-funded collaborative projects with Virginia Tech: IDEAL, GETAR) on the ArchiveSpark project - a framework for efficient Web archive access, extraction, and derivation.

The main goal of the project is to quantitatively and qualitatively evaluate ArchiveSpark against mainstream Web archive processing solutions and extend it as necessary with regard to the processing of the IDEAL project's school shooting collection. This also relates to an IMLS funded project.

This report describes the efforts and contributions made as part of this project. The primary focus of these efforts is comprehensive evaluation of ArchiveSpark against existing archive-processing solutions (pure Apache Spark with preinstalled Warcbase tools and HBase) in a variety of environments and setups in order to comparatively analyze performance improvements that ArchiveSpark brings to the table as well as understand the shortcomings and tradeoffs its usage means under varying scenarios.

The main aim of this project is to achieve three major goals. First is to get familiar with both ArchiveSpark as well as all the tools and dependencies ArchiveSpark depends upon such as Apache Spark, Warcbase, Jupyter, and Internet Archive's CDX generator. This precursor step is necessary in order to be able to replicate results obtained from the ArchiveSpark benchmark subproject on a locally deployed single-instance Cloudera stack (CDH 5.8.0). Replication involves both benchmark runs and data preparation: metadata extraction, bulk ingestion for HBase. The next goal is to execute ArchiveSpark benchmarks at scale - using either the university-hosted DLRL cluster (provided necessary infrastructure is deployed in a timely manner) or an external public cloud provider such as Amazon. Benchmark implementation should represent realistic use

cases and should also take into consideration efforts necessary for the preparation (CDX extraction and ingestion). Finally, part of the benchmarking efforts will inevitably lead to potential extension of the base ArchiveSpark framework. These extensions will be most likely in the form of new ArchiveSpark enrichments - processing and filtering functions to augment base archive RDDs.

To accomplish these goals among other things also requires close collaboration with the Internet Archive, a nonprofit memory institution operating from San Francisco, California, as well L3S Research Center (Helge Holzmann, the original author of ArchiveSpark). All the efforts of this project will be put up for review as the idea is to augment functionality/increase credibility of the original tool. As such, the end goal of the project is a successful pull request to the upstream project.

2. ArchiveSpark

ArchiveSpark [1] is a framework enabling efficient data access, extraction, and derivation on Web archive data. The primary benefit of ArchiveSpark resides in its performance and extensibility; with a relatively simple API it allows for extensible, expressive queries over the original dataset while maintaining acceptable speed of computing even for vast archives.

In order to assure fast processing ArchiveSpark makes use of two techniques. The first one involves the use of pre-generated CDX metadata indexes to selectively access resources from a Web archive. This approach is optimized for efficiency as it allows for fast filtering and extraction of a defined subset of records while avoiding performing a full scan on all records in (W)ARC files (like some of the other techniques do). CDX metadata index allows to perform efficient queries on selected common fields (much smaller dataset) and then augment the final dataset with only relevant fields from associated W(ARC)s. Since corpora used in scientific fields often comprise of data derived from a small subset of the entire Web archive, ArchiveSpark is well suited for these types of use cases.

The second technique ArchiveSpark relies on has to do with incremental filtering of the corpus. As such, a specific workflow is recommended for maximum efficiency. As mentioned above, at first, the filtering involves only metadata contained in the CDX indexes. Once this approach has been exhausted the second stage kicks in, leveraging file pointers contained in the CDX records (essentially offsets to the original archive files). ArchiveSpark selectively accesses the filtered records from the underlying (W)ARC les. Researchers/users selectively augment the record's metadata with headers and content from the (W)ARC records in order to apply further filtering/processing. This is where the extensibility of ArchiveSpark comes in place. The framework defines a concept of enrichments - essentially customized Spark mapping functions specifically tailored to derive new information (record fields). Framework-defined enrichments already address the majority of common scenarios such as extraction of named entities, HTML content, or hyperlink data. However, the API is flexible enough to allow the users to define their own enrichments, executing custom code or external tools. Then, based on the derived

information, further filters and enrichments may be applied iteratively. Finally, ArchiveSpark allows the user to save the resulting corpus in a custom JSON format, tailored to support data lineage.

Ultimately, ArchiveSpark is simply an API/specialized library for Web archive extraction and derivation and as such it is based on Apache Spark [4] and greatly inspired by its API. Much like Spark the framework leverages parallelizable data structures - RDDs - and hence it is fully compatible with any transformation methods provided by “vanilla” Spark. Additionally, its implementation is in the same language - Scala.

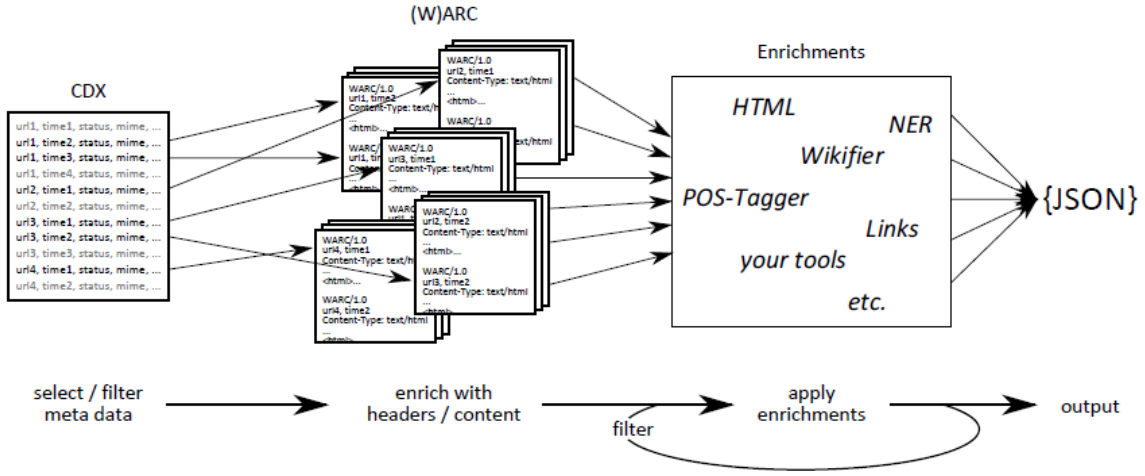


Fig 1. ArchiveSpark’s filtering -> enrichment -> export workflow [1]

3. Standard Formats

In order to better understand the notion of Web archiving the reader of this report should have a basic understanding of the most typical file formats used in the majority of modern Web archives. Among these, the most important ones are WARC/ARC files and CDX files (indexes).

3.1. WARC/ARC

WARC/ARC files represent the most fundamental file format in the world of Web archiving. The name is an abbreviation of Web Archive (Web ARChive) and it is registered/standardized as ISO 28500 [5]. WARC is essentially a format for storing archived Web resources (see Figure 2). Every record in a WARC file represents a capture of a single Web resource at a particular instant of time. As such, a WARC record is comprised of a header section that includes the URL of the resource, the timestamp of capture together with other metadata, as well as a payload section containing the actual body returned by the Web server (this can be anything ranging from plain text, JSON, XML, HTML, binary data (video, picture, music), CSS, JavaScript, ...). In case of HTTP responses, the payload also contains corresponding HTTP headers (origin, status code, ...).

Before WARC was introduced as a main format for storing Web archives, archived records used to be stored using its precursor - ARC. ARC format essentially is comprised of 2 sections - version block and URL record. A version block contains the most essential information about an actual archive file - its name and version. What follows is the URL record fields of the Web archive corresponding to one object within the ARC file. Among these are URL, IP address, archive date, content type, result code, checksum, location, offset, filename, and length [6].

```
WARC/1.0
WARC-Type: response
WARC-Record-ID: <urn:uuid:9e6f625c-74f6-4f9b-bec0-cbebc1102f4f>
WARC-Date: 2015-06-13T18:07:56Z
WARC-Target-URI: http://netbootcamp.s3.amazonaws.com/wp-content/uploads/netbootcamp-logo-internet-training.png
WARC-IP-Address: 54.231.65.41
Content-Type: application/http;msgtype=response
WARC-Payload-Digest: sha1:IS672GHFV5GLGOWBVVEN6BLF3DTLS7NB
Content-Length: 18196
WARC-Block-Digest: sha1:LI2ZUQH5PH2UDHGOODXHSGBEHC3UXDG4

HTTP/1.1 200 OK
x-amz-id-2: Bw5mQr6nHrJs+CtzJ0QImShahazsS7hqEgNswjI1jOsZ/+tByVayRWVUjL4ggMht
x-amz-request-id: D774CC57D37483F3
Date: Sat, 13 Jun 2015 18:08:35 GMT
Last-Modified: Wed, 24 Dec 2014 14:38:19 GMT
ETag: "1563196cca285cd1aa801a7a1ee91519"
Accept-Ranges: bytes
Content-Type: image/png
Content-Length: 17850
Server: AmazonS3
```

Fig 2. Sample WARC file [7]

3.2. CDX

Another very important Web archiving format, which albeit not being standardized is pretty much widely considered to be an indexing standard, is CDX. CDX is essentially an index file (reduced WARC file) containing a number of metadata fields for a particular Web capture including pointers to the (W)ARC file describing where a particular capture is stored (essentially an offset to a record in the WARC file). CDX's header line specifies the metadata fields contained in the index file (see Table 1 for a comprehensive list). Typical CDX files feature either 9 or 11 space-separated fields (field types). These are utilized by the Wayback Machine [2] to serve records to users browsing the archive. Since the Wayback Machine software is currently the access method of choice for the majority of Web archives, corresponding CDX files tend to be automatically generated by and readily available for these. For the workflows that operate off of custom WARC collections (scrapped outside of the Wayback Machine) several tools can be leveraged to automatically extract corresponding CDX files - Python-based library CDX writer [8] or Java-based jar Archive Metadata Extractor [9].

CDX Letters (Headers)	Meaning
A	canonized URL
B	news group
C	rulespace category
D	compressed dat file offset
F	canonized frame
G	multi-column language description
H	canonized host
I	canonized image
J	canonized jump point
K	FBIS
L	canonized link
M	meta tags (AIF)
N	massaged URL
P	canonized path
Q	language string
R	canonized redirect
U	uniqueness
V	compressed arc file offset *
X	canonized URL in the other href tags
Y	canonized URL in other src tags
Z	canonized URL found in script
a	original URL
b	date
c	old style checksum
d	uncompressed dat file offset
e	IP
f	frame
g	filename
h	original host
i	image
j	original jump point
k	new style checksum
l	link
m	mime type of original document
n	arc document length
o	port
p	original path
r	redirect
s	response code
t	title
v	uncompressed arc file offset
x	URL in the other href tags
y	URL in other src tags
z	URL found in script
#	comment

Table 1. CDX and DAT legend (highlighted are default fields) [10]

4. Environment

One of the prevalent features of ArchiveSpark is its flexible cross-platform application. In its nature the framework doesn't limit its user to any particular environment and only requires installed Apache Spark - version 1.6.1 and higher. Being essentially a plain SBT [11] Spark-importable library specifically tailored for Web archive file processing gives it enough flexibility to be applied in a variety of different scenarios and environments - ranging from standalone solitary Apache Spark instance to large-scale YARN/Mesos-orchestrated [12] clusters with dozens of worker

instances running Cloudera [13] or Hortonworks [14] stacks. Additionally, for even greater ease-of-use as well as environments where ArchiveSpark dependencies cannot be installed (complexity, security concerns, ...) the authors of the framework also prepackaged an ArchiveSpark installation as a Docker image [15].

! Current version of ArchiveSpark Docker image features the latest version of ArchiveSpark - 2.1.0 - that works with Spark 2.0.2 (Scala 2.11 -> Java 8) !

5. Usage

Although ArchiveSpark can be used several different ways, there are generally 2 approaches to use it directly (without referencing it in a separate project). Out of both of these the recommended approach is to use ArchiveSpark interactively from within the environment of Jupyter Notebook [16] - interactive Web application used to create and share live code documents. To install Jupyter on a local Spark/Hadoop cluster one can install this tool using the Python pip package manager:

```
# after Python 2.7 is installed
pip install --upgrade pip
pip install jupyter
```

Or, one can install the full Anaconda toolkit [17], which happens to contain the tool explicitly. For Cloudera-based clustered environments, especially those running on a public cloud, this tutorial explains how Cloudera Manager parcels can be leveraged to bootstrap Anaconda without much configuration overhead:

https://blogs.msdn.microsoft.com/pliu/2016/06/19/run-jupyter-notebook-on-cloudera/

After Jupyter is up and running, what remains is to download, configure, and deploy the ArchiveSpark kernel. This Jupyter kernel supplied by authors of ArchiveSpark is essentially a set of Python startup + configuration scripts allowing a simpler framework setup. At the time of this report the kernel is only compatible with the latest Spark 2.x. This version of Spark however contains non-backwards compatible changes with previous versions (1.x.x) as it is written in Scala 2.11, which in turn requires Java 8. As such, if the environment doesn't allow for cluster upgrade it is much easier to build ArchiveSpark and its dependencies with the older version of Scala and replace jar files in the kernel lib directory with newly compiled ones.

To build ArchiveSpark locally first clone the repository from GitHub upstream - https://github.com/helgeho/ArchiveSpark and run:

Please note that Scala and sbt must be installed

1. sbt assemblyPackageDependency
2. sbt assembly

Now, once both Jupyter and Spark 2/1.x.x are available, the following steps must be executed to properly set up the ArchiveSpark kernel.

1. Create kernel directory if it does not exist yet: *mkdir -p ~/.ipython/kernels*
2. Unpack the ArchiveSpark/Toree to the kernel dir: *tar -zxf archivespark-kernel.tar.gz -C ~/.ipython/kernels*
3. Edit the kernel configuration file *~/.ipython/kernels/archivespark/kernel.json* in order to customize it according to the environment
    - replace USERNAME on line 5 after "argv": [with your local username (i.e., *cloudera*)
    - set SPARK_HOME to point to the path of current Spark installation (i.e., */home/cloudera/spark*)
    - change HADOOP_CONF_DIR and the specified SPARK_OPTS if necessary (remove HADOOP_CONF_DIR if using ArchiveSpark with standalone Spark distribution, change *master* from *yarn-client* to *local*)
4. Optionally, replace ArchiveSpark JAR files located under *~/.ipython/kernels/archivespark/lib* with locally built project JAR files (under *ArchiveSpark/target/scala-x.x.x*)
5. Restart Jupyter (i.e., jupyter-notebook, jupyter notebook, pyspark (for parcels installation)

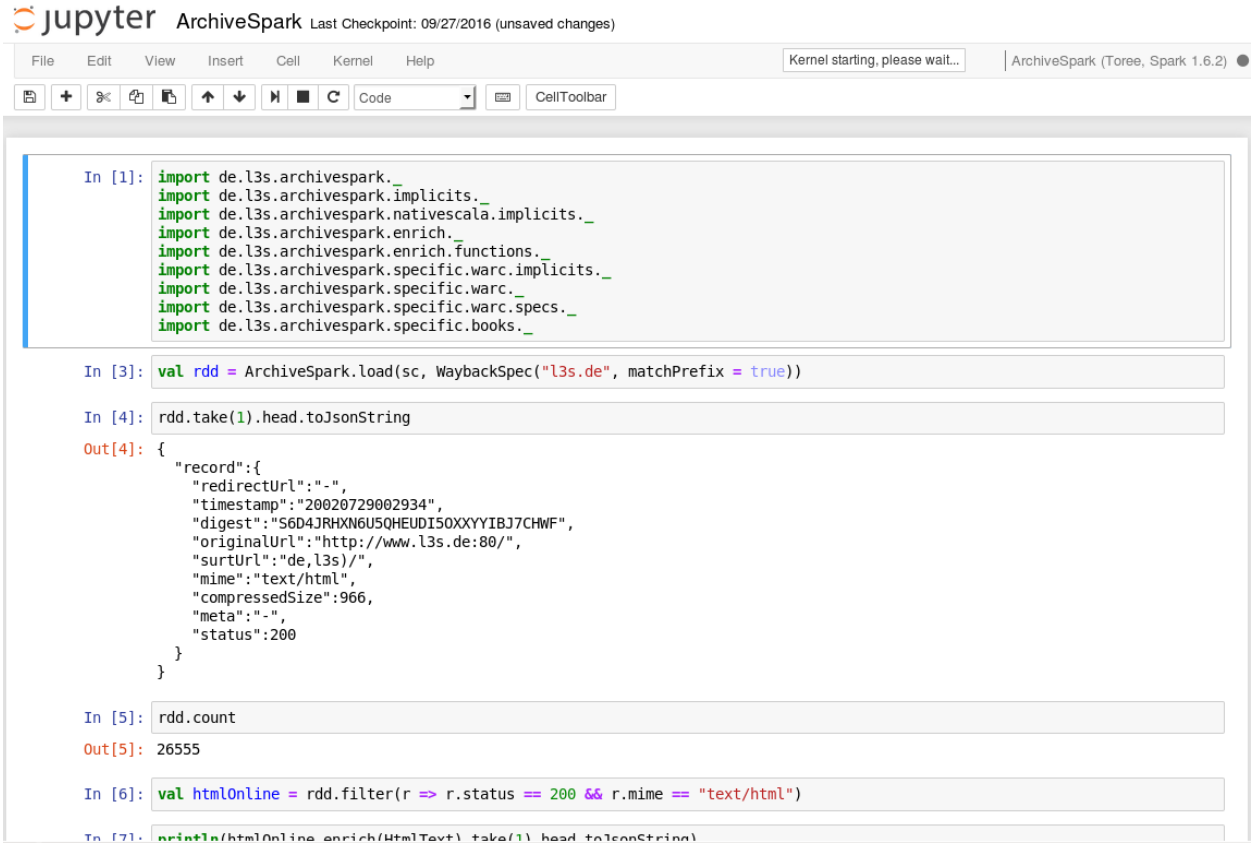


Fig 3. Sample Jupyter Notebook running ArchiveSpark

Alternatively, ArchiveSpark can obviously be used outside of the Jupyter sandbox - by simply leveraging the default Spark console. In order to start Spark shell with ArchiveSpark JARS on the classpath several parameters must be specified - notably jars (ArchiveSpark JARS (full path)), and master type (default is local). A sample command can be found below:

```
spark-shell --executor-memory 2g --master local  --conf spark.driver.maxResultSize=0 --conf spark.serializer=org.apache.spark.serializer.KryoSerializer  --conf spark.rdd.compress=true --jars /home/cloudera/ArchiveSpark/target/scala-2.10/archivespark-assembly-2.0.1.jar,/home/cloudera/ArchiveSpark/target/scala-2.10/archivespark-assembly-2.0.1-deps.jar
```

6. Evaluation

In order to evaluate performance impact resulting from leveraging CDX index files a set of benchmarks was created. These benchmarks evaluate efficiency and performance of dataset processing using 3 different approaches - ArchiveSpark and 2 baseline solutions: scan-based approach using pure Spark and Warcbase-based [18] approach using HBase - in 4 different types of scenarios.

Actual benchmarks have 2 phases - preprocessing and execution. As far as preprocessing is concerned both ArchiveSpark and HBase require certain preconditions being met. For ArchiveSpark this corresponds to CDX index files extraction - a one time step that allows to create lookup indexes necessary for initial filtering. HBase on the other hand requires initial data load - WARC ingestion. For both ArchiveSpark and pure Spark approach, WARC files get stored in the Hadoop HDFS.

Overall, the 3 evaluated systems are:

1. ArchiveSpark
2. Spark: Using Warcbase's Spark library
3. HBase: Using Warcbase's HBase library

In order to build and run the ArchiveSpark benchmark subproject certain conditions must be met. For starters, the benchmark heavily relies on the presence of the Warcbase library for both the pure Spark and HBase approaches. Unfortunately, this library doesn't provide official releases available via package manager - SBT, Gradle, Maven - which is why it needs to be built separately (please see Warcbase section 8.1.2. for details) and then added on the classpath. Built JAR files (both warcbase-core and warcbase-hbase) must be copied to the root of ArchiveSpark benchmark project and placed under the *lib* directory (might be created if doesn't exist). Afterwards, the building requires the exact same steps as the parent project - ArchiveSpark:

```
# Please note that Scala and sbt must be installed

3. sbt assemblyPackageDependency
4. sbt assembly
```

! Please note that at the time of this report the only functional version of ArchiveSpark benchmark is the one residing in the VTUL fork - https://github.com/VTUL/ArchiveSpark. Please, follow the instructions in the README.md !

6.1. Setup

As already mentioned above, in order for the benchmarks to be successful certain preconditions must be met. As such, several WARC-based tools have been leveraged to properly prepare the environment for actual runs.

6.1.1. CDX-Writer

CDX-Writer is an open-source Python script/repository used to create CDX index files out of WARC files. Written in Python 2.7 it allows to efficiently extract and output WARC files metadata and optionally store them in separate files. The usage is fairly straightforward:

```
cdx_writer.py [options] {filename}.warc.gz > {filename}.cdx
```

Unfortunately, CDX-Writer doesn't work "out-of-the-box", which is why manual setup is required. As such, in order to run the CDX-Writer a user must first install all the necessary dependencies via Python pip:

```
pip install git+git://github.com/rajbot/surt#egg=surt
pip install tldextract
pip install chardet
```

After that, warc-tools must be downloaded and extracted in the directory with the main CDX-Writer script - cdx-writer.py.

```
wget https://bitbucket.org/rajbot/warc-tools/get/e8266e15f7b6.zip
unzip e8266e15f7b6.zip
```

! Careful: When extracting CDX metadata using CDX-Writer make sure you run the script from the same directory where the files are located - this way the CDX file will contain proper value in the filename column (just the filename) allowing you to upload WARC files and CDX files together to HDFS and allowing ArchiveSpark to find matching records for CDX records !

6.1.2. Warcbase

Warcbase is essentially an open-source platform for managing Web archives built on top of Hadoop and HBase. The platform provides a flexible data model for storing and managing raw content as well as metadata and extracted knowledge. Its tight integration with Hadoop provides powerful tools for analytics and data processing via Spark.

Other than archive analysis Warcbase also provides several tools for data manipulation. Overall, Warcbase is essentially a Maven project featuring 2 modules - warcbase-core and warcbase-hbase. Warcbase-core contains base code for WARC file extraction and analysis together with domain-specific entities. The Warcbase-hbase submodule features HBase-specific code for data ingestion and maintenance. Unlike the majority of widely used open-source tools, Warcbase doesn't contain any pre-built releases but instead allows the user to custom-build the solution for a particular platform.

In the context of ArchiveSpark benchmarks Warcbase gets used in order to insert a WARC collection in HBase. In order to build and use Warcbase the user must first have Maven [19] installed. After this precondition has been met Warcbase build can be achieved with the following command run in the root directory:

```
# build + tests
mvn clean package
# or just build
mvn clean package -DskipTests
```

One of the inherent parts of the Warcbase build process is helper script generation. Warcbase leverages several Maven plugins - one of them being the *appassembler* [20] plugin. This nifty tool allows the build to generate both shell/bash and command prompt utility/wrapper scripts in order to facilitate logic execution. After a successful build these scripts reside under *warcbase/warcbase-hbase/appassembler/bin* directory. One execution script that is of particular interest in ArchiveSpark benchmarks is called *IngestFiles*. Its invocation forces Warcbase to read and parse existing WARC/ARC files in the specified directory, generate a new HBase table, and store the archive payload in an HBase-specific way.

To execute IngestFiles user first needs to point Warcbase to an existing HBase configuration via CLASSPATH_PREFIX environment variable. Additionally, he needs to supply appropriate parameters such as WARC files location (directory), HBase table name as well as the type of action (create or append). Below is a sample ingestion command.

```
# -dir (WARC directory), -name (HBase table name), -create (action)
export CLASSPATH_PREFIX="/etc/hbase/conf/"
target/appassembler/bin/IngestFiles -dir ~/desktop/WARC-directory/ -name example -create -gz
```

Warcbase exploits certain properties of HBase to enable fast access to corresponding Web archives. Different captures of a crawled Web resource are stored as timestamped versions of the same record in HBase. URLs are stored in an inverted, sort-friendly format (*www.archive.org/collection/123* becomes *org.archive.www/collection/123*) and are used as row keys for fast lookups with the MIME type serving as a column qualifier (column name). The main reason for this preprocessing has to do with the way HBase stores its records. Row keys for HBase tables are lexicographically sorted, supported by the B-tree index for fast lookup [21]. As such, similar WARC domains tend to be stored close to each other, enabling very efficient domain queries (common URL prefix). Overall, the above-mentioned design decisions allow for an efficient selection and filtering process based on these three attributes.

! Unfortunately, at the time of this report, native (upstream) Warcbase ingestion tools exit with an exception. The error is caused by non-backwards-compatible changes between different versions of hbase-client library. There is a pending pull request against the upstream to address the problem - Issue #255 - Fixed NoSuchFieldFoundException in IngestFiles. If you require IngestFiles capabilities please use VTUL forked Warcbase where this error has already been fixed - https://github.com/vtul/warcbase !

6.1.3. Cloudera on AWS

Amazon AWS [22] is an increasingly more popular solution for developers and researchers alike. Its affordability and flexibility enables the users to quickly provision necessary resources in order to run large scale experiments without the hassle of setting up necessary infrastructure. Cloudera, recognizing the benefits of a public cloud, provides its own custom solution to quickly provision and manage the Cloudera cluster on AWS. Its tool, Cloudera Director, enables to effortlessly and securely bootstrap an arbitrarily large Cloudera cluster (1 manager node, n worker nodes).

In order to deploy Cloudera Director on AWS, Amazon Virtual Private Cloud (Amazon VPC) must be created, a standalone EC2 instance provisioned, and a secure connection created. Cloudera Director supports several versions of operating systems - RHEL 6.5, 6.7, 7.1, 7.2 and Ubuntu 14.04 - as well as several versions of JDK - Oracle JDK 7 and 8 [23]. Additionally, the Director also supports EC2 spot instances for maximized cost-efficiency.

After the EC2 environment has been set up (instance provisioned) several steps need to be undertaken in order to get Cloudera Director up and running. After successfully SSH-ing to the running instance, the user must first install Java. Afterwards, Cloudera Director can be added to package manager (*http://archive.cloudera.com/director/redhat/7/x86_64/director/cloudera-director.repo* for RHELs) and both director client and server can be installed. The next step is to start Cloudera director service by running *sudo service cloudera-director-server start* (regardless of target platform). The last important step has to do with security. Cloudera provides its own software-based mechanism to manage access, which is why it is recommended to disable AWS-provided security policies such as instance firewall.

As mentioned above, AWS provides several access control mechanisms to manage instance security out-of-the-box. Out of these the 2 prevalent ones are instance firewall (disabled in the previous step) and EC2 security groups. AWS security groups restrict specific IP addresses to access specific EC2 resources (ports). However, in the case of the Cloudera cluster it is recommended that the users leverage different mechanism to access their cluster instances. As such, the users are encouraged to restrict all external access to the resources and only connect to their cluster via SOCKS proxy [24]. The main reason for this workflow is the fact that behind the scenes Cloudera Director interacts with AWS APIs using user-supplied AWS credentials - access key ID and secret key. These credentials are used to automatically (programmatically) provision necessary resources - EC2, RDS, CloudWatch, … As such, a potential attacker (upon successful infiltration attempt) could easily access existing AWS resources and freely create new ones.

In order to connect to an internal Cloudera VPC network two steps need to be completed. First, a secure SOCKS channel must be established over the SSH connection. The command is:

```
ssh -i "<aws-key-file>.pem" -CND 8157 ec2-user@<ec2-instance-running-director-server>
```

Afterwards, in order to access Cloudera Director UI an instance of a Web browser must be launched containing a proxy server as a parameter. For Google Chrome the following command does the trick:

```
# make sure the port number is the same as above
"C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" ^
--user-data-dir="%USERPROFILE%\chrome-with-proxy" ^
--proxy-server="socks5://localhost:8157"
```

Finally, typing *localhost:7189* opens up an instance of running Cloudera Director.

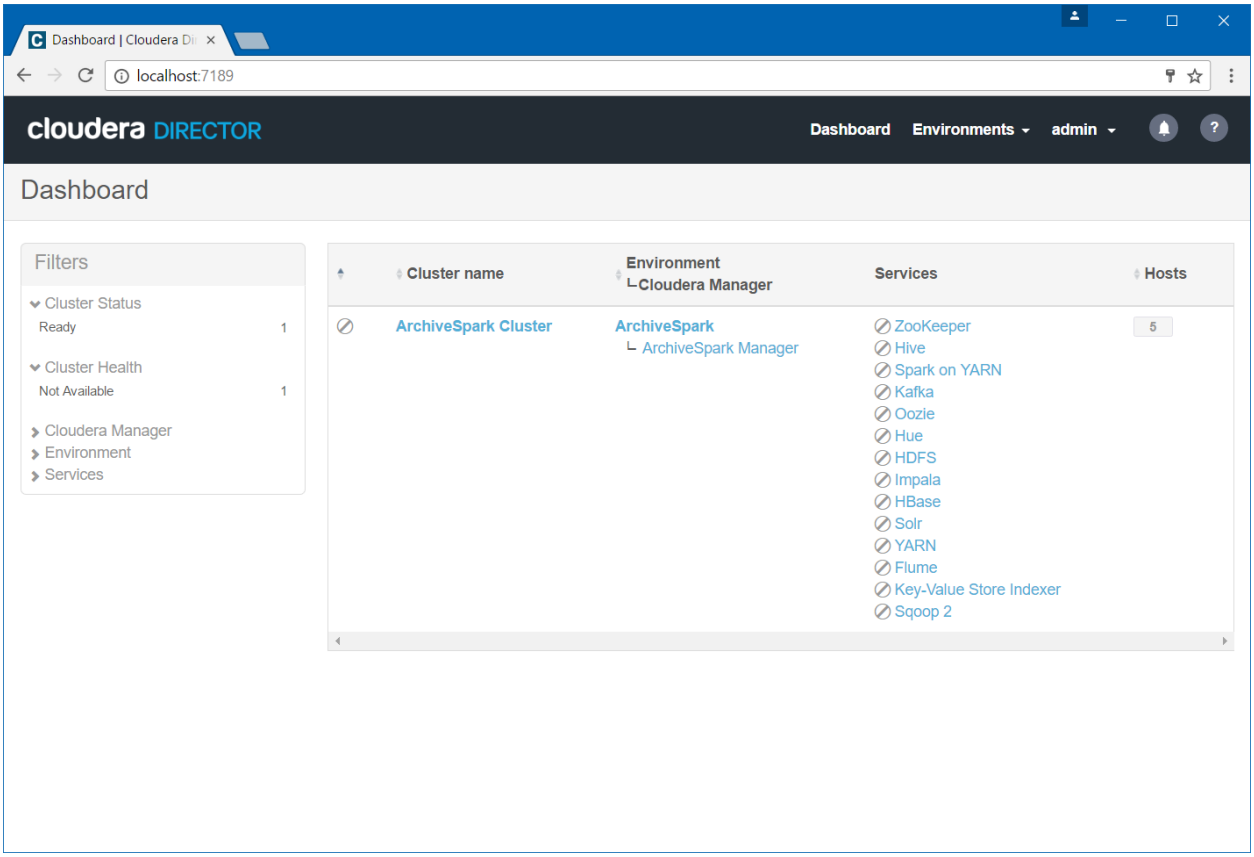


Fig 4. Cloudera Director on AWS

Ultimately, Cloudera Director allows bootstrapping a full Cloudera cluster [25]. An user can select the *Add Environment* action and simply follow the wizard to first deploy an instance of Cloudera Manager (service orchestration) followed by actual worker instances. The wizard is flexible enough to allow users to choose instance details (AWS AMI, instance size), to upload remote access credentials (AWS access key and username), as well as to specify the Cloudera distribution (using pre-compiled Cloudera parcels). The bootstrapping process essentially takes care of everything - from provisioning AWS resources (via provided access credentials (access key ID and secret key) to the actual installation of Cloudera stack (Zookeeper, Hive, Spark, Solr, …). After the wizard has completed (takes several minutes) Cloudera Manager UI should be accessible at *<private-ip>:7180*.

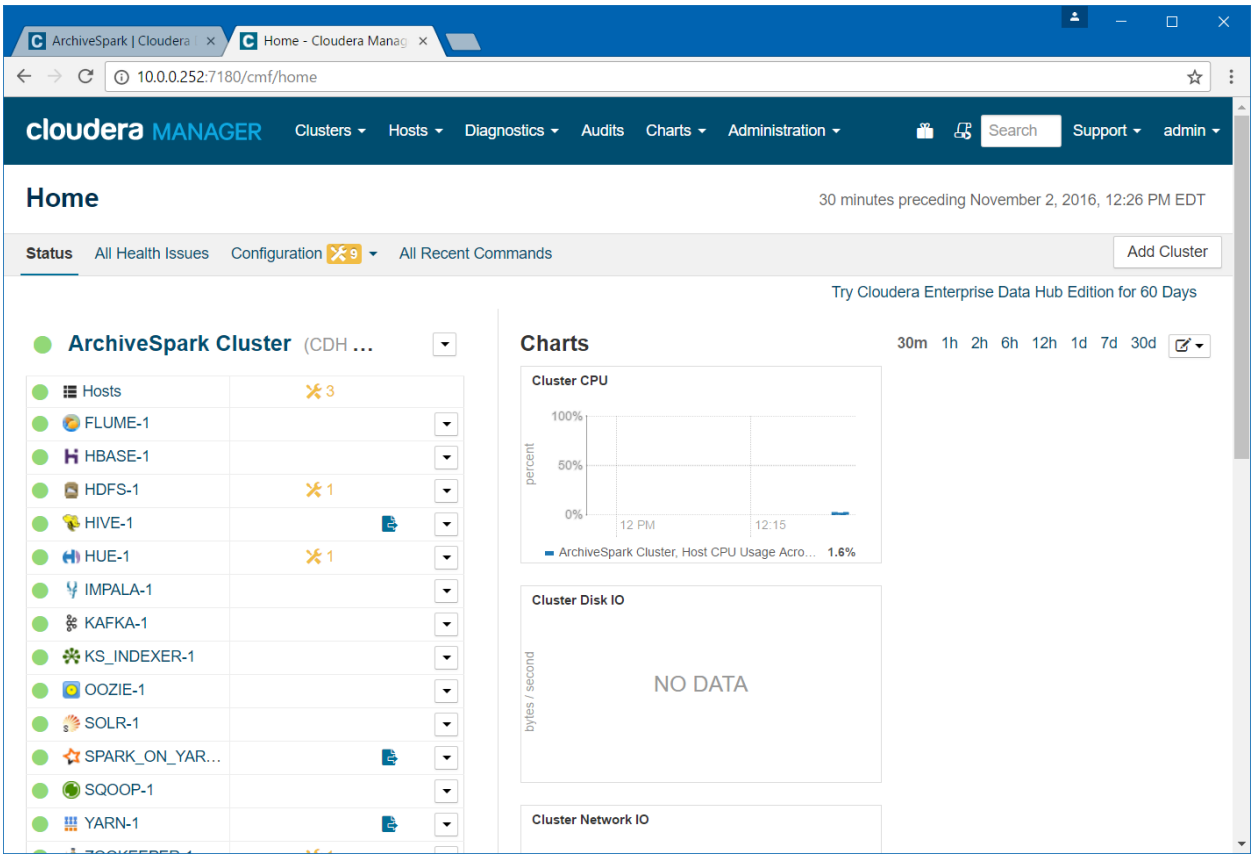


Fig 5. Cloudera Manager on AWS

! When running the benchmark in the distributed environment such as AWS or DLRL cluster using YARN as a resource manager you need to make sure that the HBASE_CONF_DIR environmental variable is properly set on all the cluster nodes. This variable should contain the path to the HBase configuration directory - typically /etc/hbase. If you forget to set this variable everywhere the benchmark will get stuck in the HBase stage !

6.2. Benchmarking

In order to provide comprehensive evaluation of multiple scenarios ArchiveSpark needs to be deployed and tested in different types of environments under different conditions. The goal is to understand how ArchiveSpark fares against the state-of-the-art for varying sizes of datasets and different types of extraction operations. As such, we need to analyze a relationship (ratio) between data preprocessing and actual execution for increasing sizes of datasets as well understand how this technique compares to the other two (pure Spark, HBase).

Experiment environment-wise 2 types were initially intended to be involved. For local testing and framework augmentation I used a local VM running Cloudera Quickstart CDH 5.8.2 (update 2). This VM runs 64-bit Centos 6.7 and operates with one virtual CPU and 4896 MB of RAM. As such,

it is an ideal candidate for both development as well as small scale execution. However, in order to provide more representative results (representative of the scenarios for which ArchiveSpark was initially intended) the decision has been made to leverage the Virginia Tech DLRL Hadoop cluster [26]. Throughout the course of this semester however this decision has been adjusted as the cluster currently only supports the ArchiveSpark-incompatible version of Cloudera (CDH 5.6.0). As such, all the benchmarking efforts were migrated to a hosted public cloud service provided by Amazon - AWS (courtesy of Dr. Zhiwu Xie).

All the different runs and benchmark setups were ultimately executed using the same set of AWS-provisioned resources - 5-node Cloudera CDH 5.8.2 cluster consisting of m4.xlarge AWS EC2 instances. The instance specification described in the Table 2 corresponds to the most recent information provided by Amazon [27]:

vCPUs	4x (2.3 GHz Intel Xeon® E5-2686 v4 (Broadwell))
Memory	16 GiB
Storage	30 GB EBS-optimized
Network Bandwidth	750 Mbps

Table 2. M4.xlarge EC2 AWS instance type

6.2.1 Small Scale

6.2.1.1. Overview

Both the small scale and the medium scale benchmark runs involve Web archive dataset filtering subsequently followed by a relevant records extraction. As far as the small scale benchmarking is concerned, we decided to focus on 4 different types of scenarios that were initially introduced in the ArchiveSpark paper's evaluation:

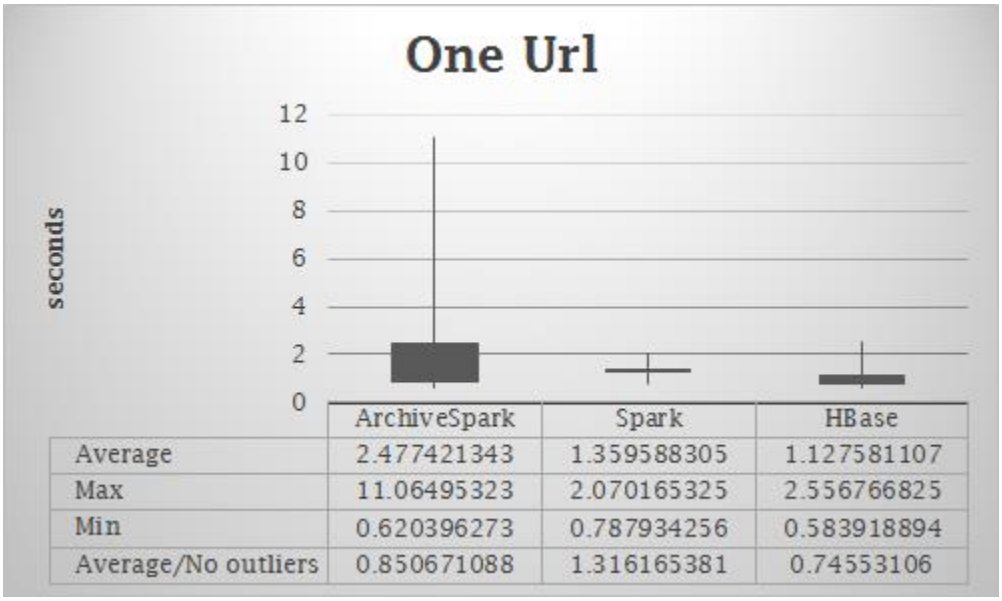
1. Filtering of the dataset for a particular URL (one URL benchmark)
2. Filtering of the dataset for a particular domain (one domain benchmark)
3. Filtering of the dataset for a range of records (one month benchmark)
4. Filtering of the dataset for a particular active domain - status code = 200 (one active domain benchmark)

These initial benchmarks essentially cover all the 3 different scenarios of WARC records extraction supported by HBase - by URL (row key), by MIME type (column name), and by retrieval timestamp (HBase table timestamp). The dataset chosen for the processing consists of one capture of archive.it domain - what we consider a representative sample of a small Web archive collection. The overall, this sample features 261 WARC records requiring 2.49 MB of disk space.

6.2.1.2. Results

For all the individual small scale benchmark runs I collected 10 distinct results for each setup - benchmarking scenario + technology (one URL + ArchiveSpark). Afterwards, I used stock visualization technique to display time deviation as well as ranged estimate of actual (precise) value for a particular run. Processed results thus feature maximum value, minimum value, and an average value as well as average value without outliers. Min/max range then showcases measured values spread while average/average-no-outliers range contains the information about most-likely value (middle->closer-to-no-outliers).

One URL

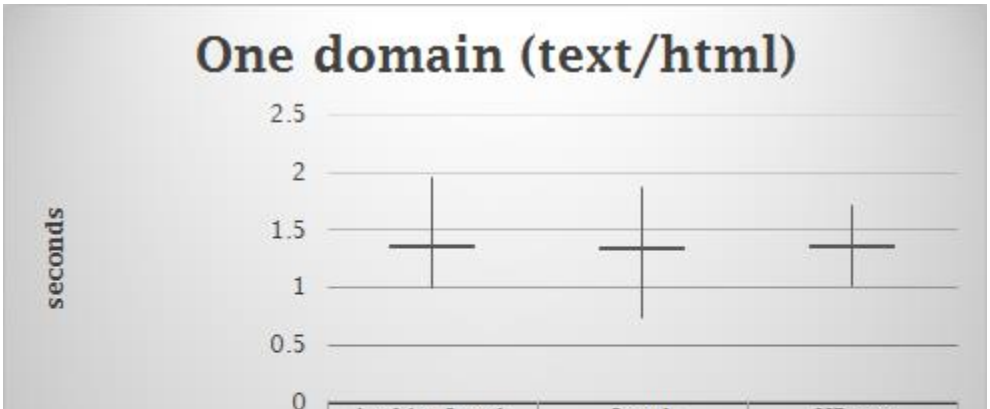


	ArchiveSpark	Spark	HBase
Average	2.477421343	1.359588305	1.127581107
Max	11.06495323	2.070165325	2.556766825
Min	0.620396273	0.787934256	0.583918894
Average/No outliers	0.850671088	1.316165381	0.74553106

Fig 6. Small Scale - stock chart for one URL benchmark

Several interesting findings about the Figure 7 include these. I found out that the vast spread of result values experienced with ArchiveSpark wasn't caused by the framework itself but rather by the architecture - most likely YARN (cluster resource manager) performing resource lookup and executor allocation/partitioning. Subsequent runs of the same benchmark setup fall closer towards the average/no-outliers result and other types of benchmarks (one domain) don't exhibit this behavior. As such, the results suggest HBase as the fastest solution (due to the B-Tree-like index on the row keys (URLs) allowing fast and efficient searches), ArchiveSpark second (CDX indices), and Spark coming out last.

One domain (text/html)

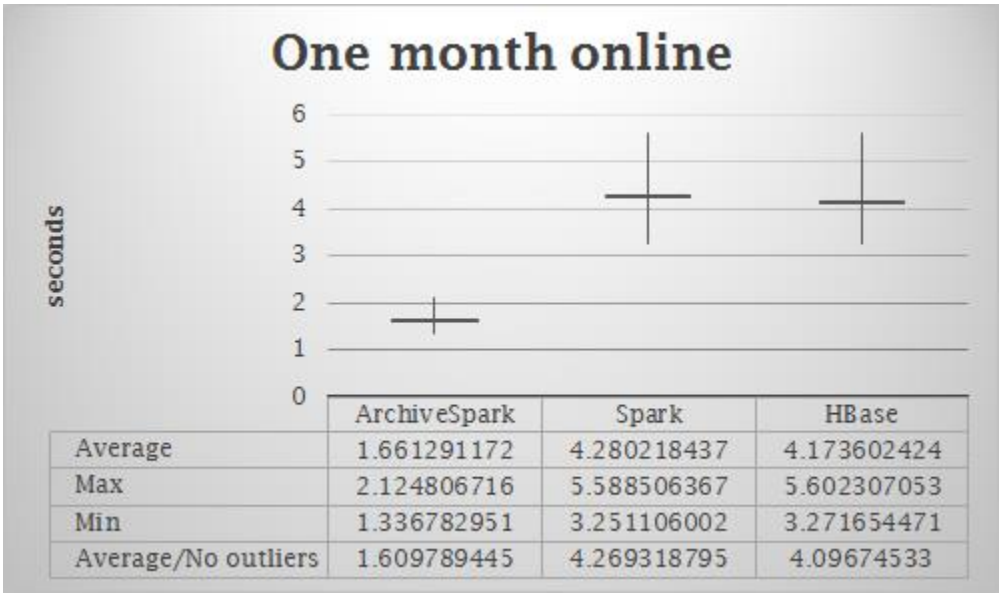


	ArchiveSpark	Spark	HBase
Average	1.362194425	1.347103491	1.371052337
Max	1.959518501	1.873476908	1.722339119
Min	1.003479605	0.745562134	1.026370613
Average/No outliers	1.362194425	1.347103491	1.371052337

Fig 7. Small scale - stock chart for one domain benchmark

Interestingly enough, the results for one domain came out very similarly (with pure Spark holding the first place for the fastest run - min). In this case HBase leverages 2 important techniques. Inverted URLs as row keys with B-Tree index allow for fast and efficient prefix lookup while MIME type; column further narrows down the search (this is most likely responsible for the smallest spread of values). Additionally, CDX indexes for ArchiveSpark didn't seem to provide much of the performance benefit for this particular scenario.

One month online

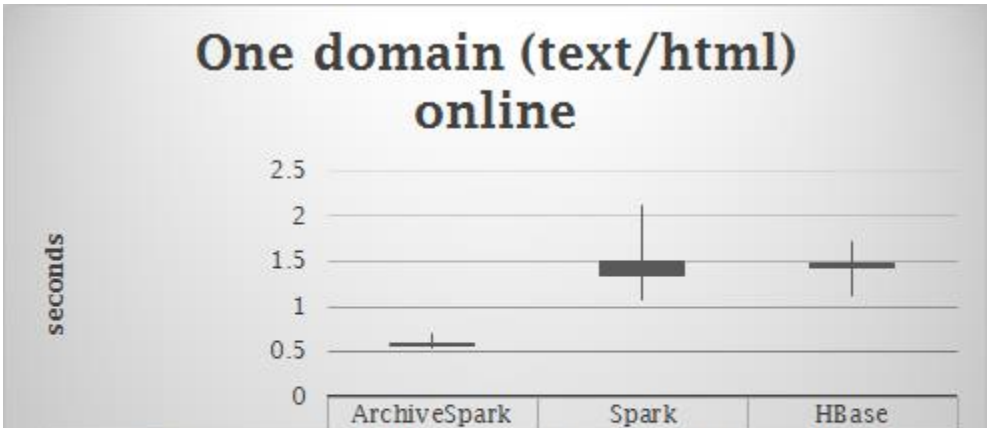


	ArchiveSpark	Spark	HBase
Average	1.661291172	4.280218437	4.173602424
Max	2.124806716	5.588506367	5.602307053
Min	1.336782951	3.251106002	3.271654471
Average/No outliers	1.609789445	4.269318795	4.09674533

Fig 8. Small scale - stock chart for one month benchmark

One month filtering is the first benchmark result where the benefits of ArchiveSpark start to appear. ArchiveSpark clearly dominated this category - both average-wise and spread-wise. Additionally, the results suggest that HBase timestamp lookup isn't as fast as row lookup (probably sequential scan). Comparable results of HBase and Spark can most likely be attributed to the fact that the HBase workflow consists of 2 parts (timestamp filtering by HBase engine and subsequent status code filtering using Spark). ArchiveSpark on the other hand maintains comparable performance to domain lookup as both timestamp and status code are featured in CDX indices.

One domain (text/html) online



	ArchiveSpark	Spark	HBase
Average	0.60295292	1.497542806	1.468800563
Max	0.712397826	2.122264735	1.710818983
Min	0.547894969	1.073288654	1.11752257
Average/No outliers	0.590792374	1.347723263	1.441909627

Fig 9. Small scale - stock chart for one active domain benchmark

One online domain benchmark is essentially one domain benchmark with one notable exception - filtering also involves a status code. As we can see, HBase performance begins to fall considering that this field is not present among HBase native filtering capabilities and thus must be extracted in the second phase - Spark - from the WARC payload. ArchiveSpark is able to retain rather optimal performance because of the presence of all required fields in the CDX metadata.

6.2.2. Medium Scale

6.2.2.1. Overview

Medium scale benchmarking definitely consists of somewhat more interesting and more realistic setups that fully evaluate the performance of the ArchiveSpark solution against the baseline. Unlike for the small scale, this time around I performed the evaluation using a slowly increasing dataset to better observe evolution in performance for all 3 solutions. In addition to the increasing dataset the benchmark runs evolved to incorporate filterings that are initially absent in the lookup data for all 3 techniques. As such, updated set of scenarios contains the following:

1. Filtering of the dataset for a particular URL (one URL benchmark)
2. Filtering of the dataset for a particular domain (one domain benchmark)
3. Filtering of the dataset for a particular active domain - status code = 200 (one active domain benchmark)
4. Filtering of the dataset for pages containing scripts (pages with scripts benchmark)

A medium scale benchmark then leverages a different dataset - WIDE collection [28]. This collection provided by the Internet Archive corresponds to data obtained from the Website Crawl executed on February 25, 2011. The dataset consists of hundreds of different domains and

altogether contains 214470 records. For easier manipulation the authors split the collection into 9 roughly equal parts (approx. 1 GB each) summing up to 9063 MB. As such, my medium scale benchmark leverages these parts, slowly increasing the dataset all the way to 9 parts and capturing results for all the distributions.

6.2.2.2. Preprocessing

Unlike in the case of small scale runs this time I also took into consideration the cost of preprocessing as it also influences the overall cost of a particular technique (even though it is only executed once).

CDX Extraction	4 minutes 41 seconds
HDFS Upload	2 minutes 46 seconds
HBase Ingestion (x9)	1 file – (1 minute 10 seconds <=> 1 minute 32 seconds) Sequential ingestion – approx. 13 minutes 54 seconds

Table 3. Medium scale preprocessing cost

To clarify, the ingestion results contain the range of durations it took to ingest 1 WARC collection part in the HBase. As such, sequentially, the overall duration sums up to approximately 13 minutes 54 seconds. Unfortunately, the Warcbase ingestion utility doesn't support parallelization of tasks out of the box but technically other approaches could be leveraged to bring the overall cost down (several processes, maybe map-reduce).

6.2.2.3. Results

Once again, just like in the case of small scale benchmarking I collected several results for each setup. This time however my visualization technique slightly changed. For starters in order to understand the overall evolution of the experiment I plotted result averages for all 3 technologies next to each other while augmenting the number of collection parts from 1 all the way to 9 - essentially a grouped bar chart. Afterwards, because of Spark largely skewing y values I decided to make another set of chart for every scenario simply comparing ArchiveSpark and HBase (line charts). Finally, after a full collection had been preprocessed I performed one last in order to capture the spread of values for all the data (stock charts).

One URL

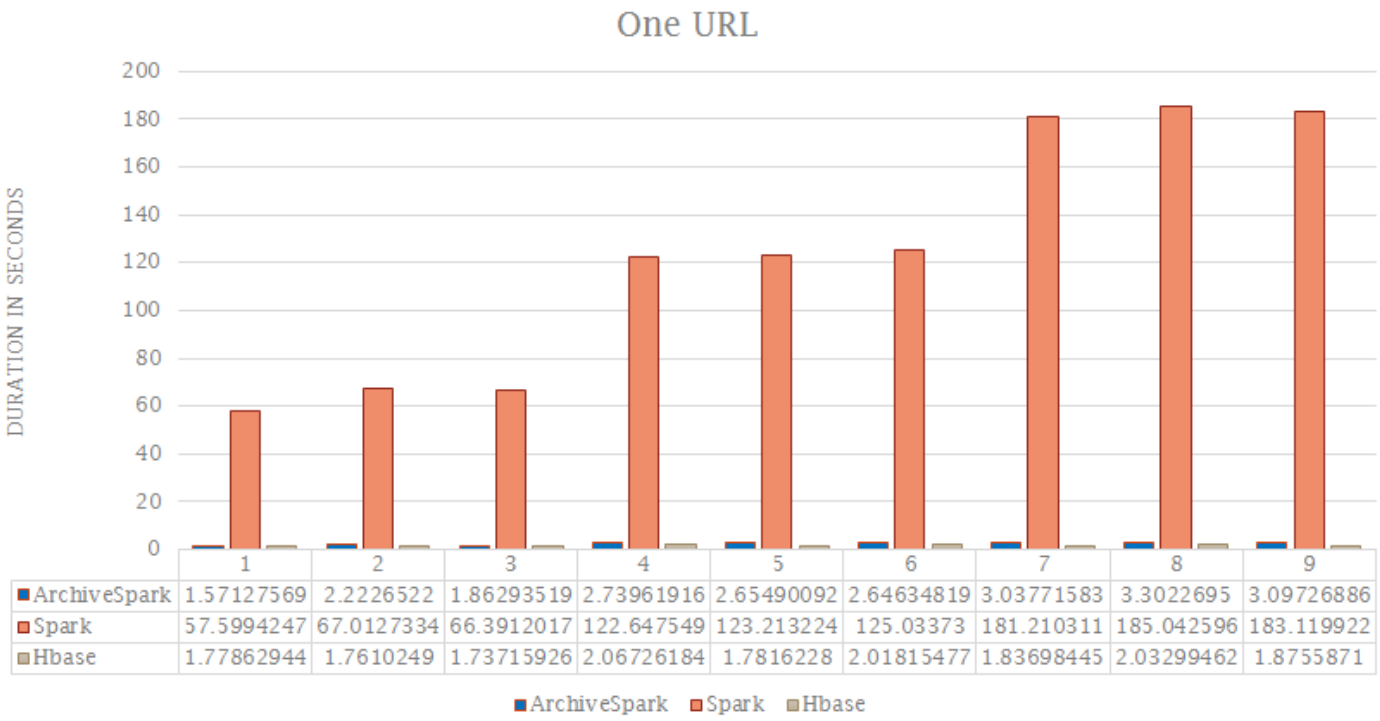


Fig 10. Medium scale - bar chart for one URL benchmark

The evolution for one URL suggests huge overhead of pure Spark processing which only keeps increasing as the collection gets bigger. What is interesting is Spark's fairly comparable performance for 3 subsequent runs (1GB, 2GB, and 3GB) followed by deterioration in performance by the factor of 2 for the next group. My hypotheses in this scenario is that the Cloudera setup I had been using could successfully leverage only 3 processing nodes/3 workers/3 executors at the time, which would explain this particular evolution.

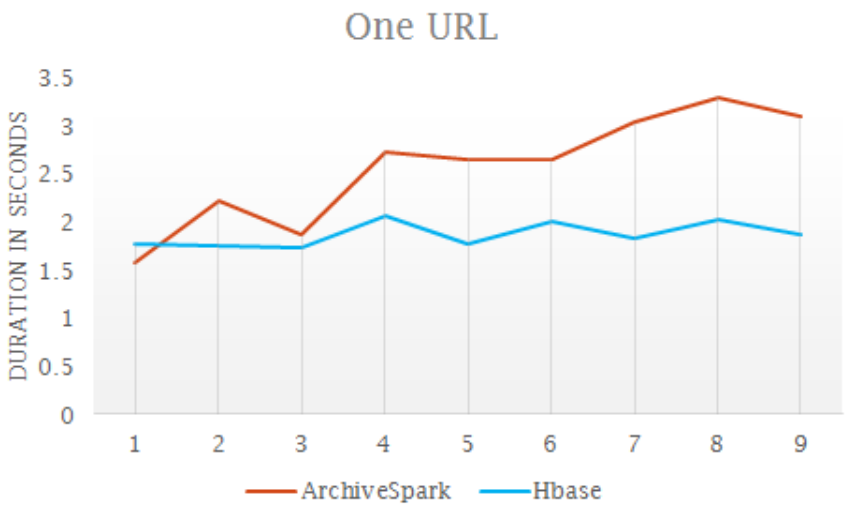


Fig 11. Medium scale - line chart for one URL benchmark

The results suggest a slowly increasing duration of the benchmark for ArchiveSpark with fairly stable/consistent performance from HBase. My explanation of this behavior has to do with the way HBase stores the data - using B-Tree prefix indices on row keys (sorted) and sharding for even faster lookup. As such, ideally, with direct lookup the engine only needs to find an appropriate data shard (O(# of shards)) after which O(logn) search can be leveraged. As such, the performance is influenced only minimally. ArchiveSpark on the other hand, albeit leveraging CDX metadata, still has to perform sequential lookup on all the records (O(n)) in order to retrieve the desired value.

One domain (text/html)

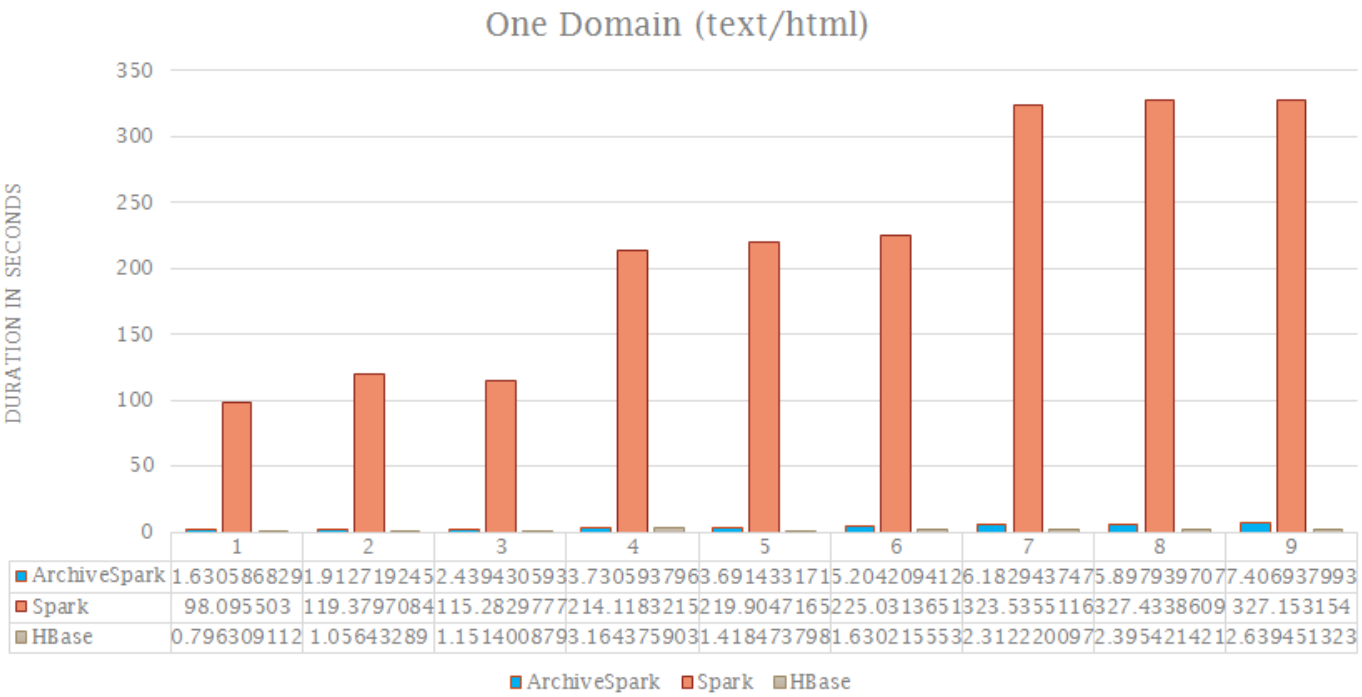


Fig 12. Medium scale - bar chart for one domain benchmark

Much like in the previous results we observe continuous decrements in performance for pure Spark and once again they follow the same factor-2 trend. However, the overall duration has also worsened when compared to one URL benchmark. Several factors could be responsible - usage of regular expressions (heavy string domain matching) or even larger final dataset (several values as opposed to one).

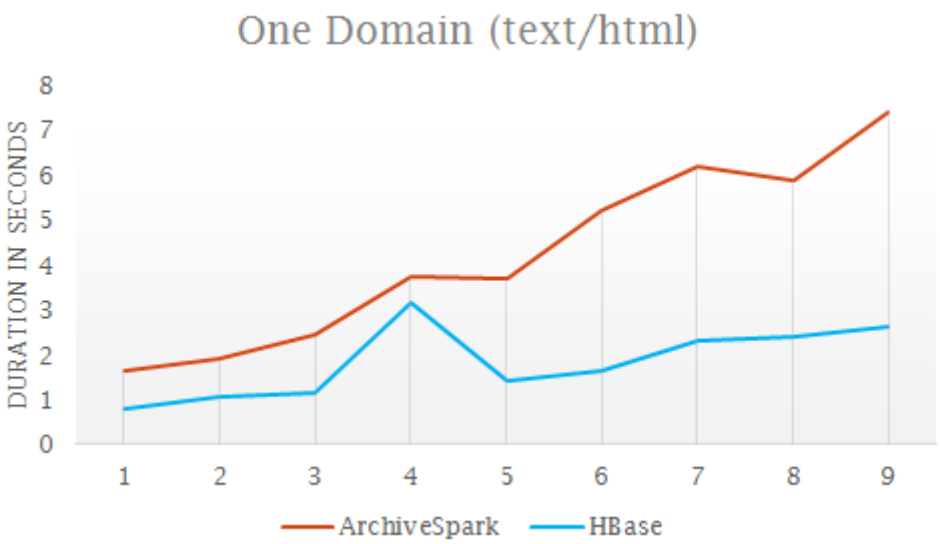


Fig 13. Medium scale - line chart for one domain benchmark

Once again HBase fares better than ArchiveSpark although this time around the difference between data points gets smaller with increasing size of the collection. As such, we can see slow linear decrease in performance when observing HBase results. This makes sense as the denser the dataset the more matching values for a filtered domain. Also, the same mechanisms as discussed above (sharding + B-Tree) seem to have a positive effect on the performance as HBase curve grows slower than the ArchiveSpark curve.

One domain (text/html) online

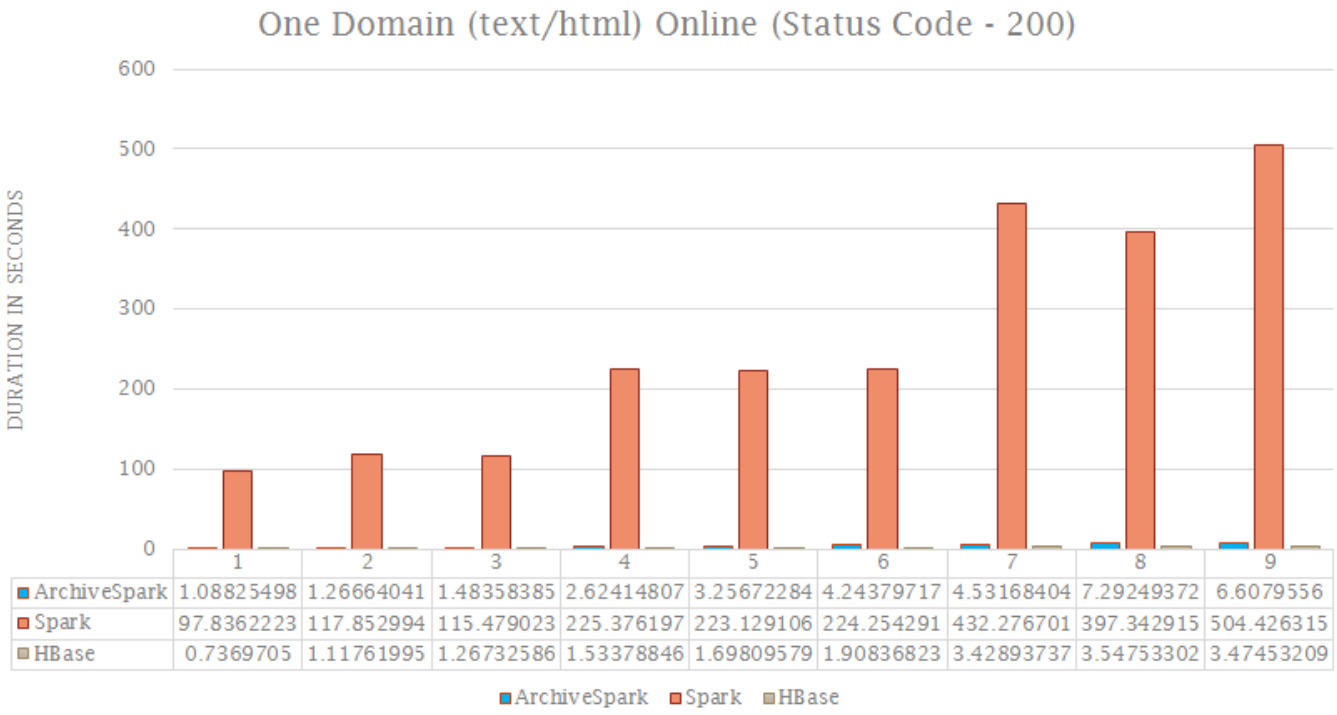


Fig 14. Medium scale - bar chart for one online domain benchmark

Once again we can observe factor-2 deterioration in Spark's performance. This time however the additional filtering for status code seems to do more harm as the durations for 7, 8, and 9 parts have gone up by 100 seconds when compared to the previous scenario.

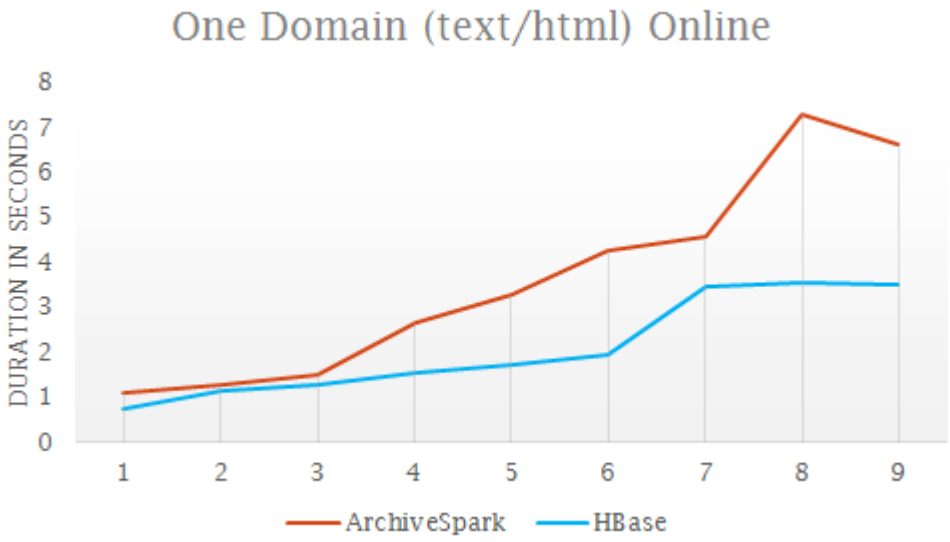
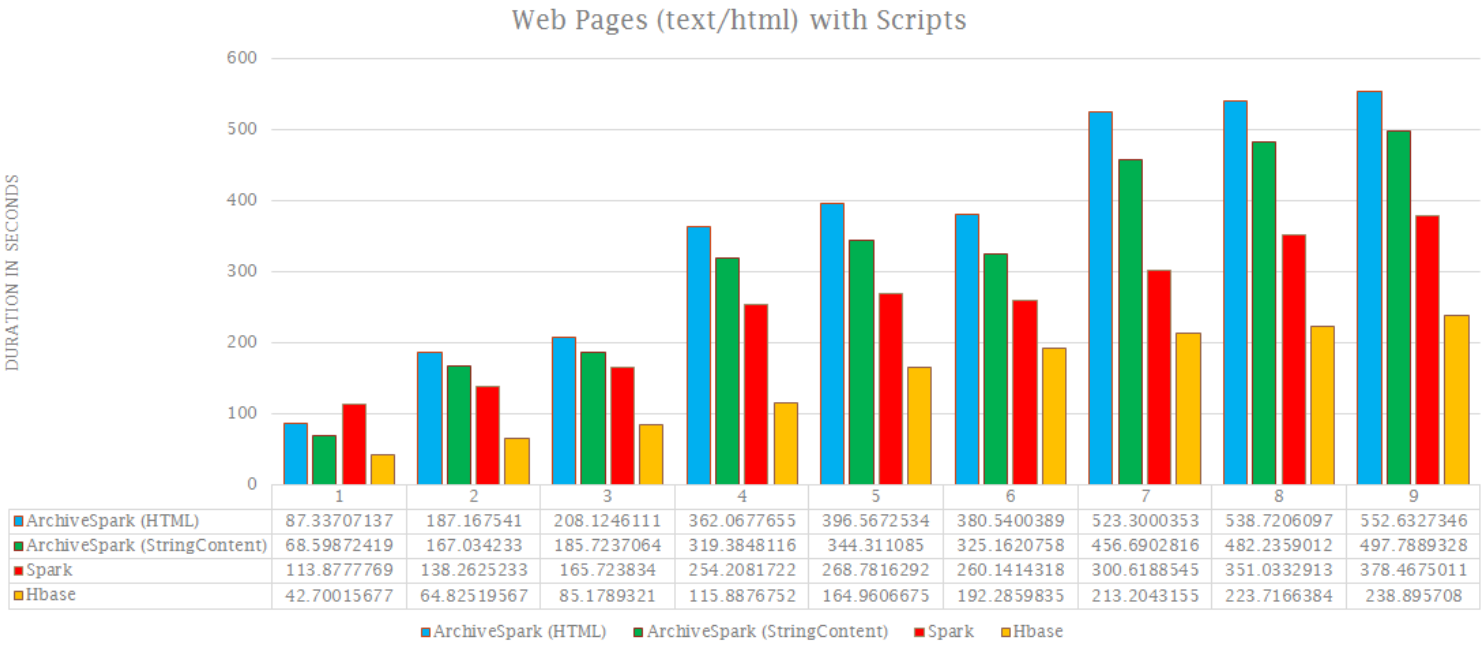


Fig 15. Medium scale - line chart for one online domain benchmark

This is the part where things get interesting. Once again we can observe slow degradation in performance for both techniques - this time however the difference is significantly smaller - especially for the first 3 parts of the entire collection where the difference is pretty much non-existent. What is interesting however is the fact that this same testing scenario showed opposite results in the case of small scale benchmarking - with ArchiveSpark clearly outperforming HBase. HBase workflow in this case is split in the domain lookup using row keys followed by status code filtering performed on the resulting dataset in Spark. The above result thus suggests that the first part of the run is fast enough to counteract poor follow-up performance involving filtering (must use full payload). As such, overall performance even exceeds that of ArchiveSpark, which contains all the necessary fields as part of the metadata.

Pages with scripts



	1	2	3	4	5	6	7	8	9
ArchiveSpark (HTML)	87.33707137	187.167541	208.1246111	362.0677655	396.5672534	380.5400389	523.3000353	538.7206097	552.6327346
ArchiveSpark (StringContent)	68.59872419	167.034233	185.7237064	319.3848116	344.311085	325.1620758	456.6902816	482.2359012	497.7889328
Spark	113.8777769	138.2625233	165.723834	254.2081722	268.7816292	260.1414318	300.6188545	351.0332913	378.4675011
Hbase	42.70015677	64.82519567	85.1789321	115.8876752	164.9606675	192.2859835	213.2043155	223.7166384	238.895708

Fig 16. Medium scale - bar chart for pages with scripts benchmark

My last benchmarking scenario only features one chart as the line chart wasn't particularly necessary. This run is specific by a property that the information on which we are filtering isn't contained in the ArchiveSpark CDX metadata index and thus the records must be augmented in order to proceed (incremental filtering technique). As such the only filtering information we can initially leverage is the MIME type (text/html). Additionally, the above results feature 2 entries for ArchiveSpark as these correspond to 2 different ways the resulting dataset can be achieved. Both of these ways involve the concept of ArchiveSpark enrichments - 1st one - HTML, 2nd one - StringContent. The more intuitive way that probably the majority of researchers would choose is HTML. This workflow enriches the records with corresponding HTML elements (in this case scripts). The enrichment internally relies on the jsoup [29] Java HTML parser to create a tree like structure from the textual payload of a particular WARC record. The second approach is slightly less intuitive - StringContent enrichment extracts WARC binary payload (array of bytes) and turns

it into a string appending it to the record (plain unformatted string). Afterwards a regular expression is used to filter out the records that don't contain scripts (this is also the approach used for pure Spark and HBase). The final results are interesting, with ArchiveSpark coming out loser for both approaches (even worse than Spark). HBase showcases the best performance which is interesting considering the majority of processing is happening in the second stage (Spark) - only filtering on MIME type. Pure Spark comes out second outperforming ArchiveSpark. My theory for this outcome has to do with the way ArchiveSpark is designed. The fundamental idea around which ArchiveSpark is based has to do with lazy execution; the only eagerly fetched resources are CDX metadata fields. As such all the enrichments get attached to the records (functional approach of composing functions), only to be executed when an eager operation gets invoked - such as regular expression matching. The Warcbase library for Spark on the other hand fetches the entire payload eagerly and then extract the necessary fields on demand. The lazy execution of enrichments thus seems to hurt the ArchiveSpark's performance in this case as the initial filtering on CDX metadata still leaves a sizeable collection. Additionally, what is interesting is the fact that the intuitive way of obtaining the dataset (using HTML) proves to be slower than the less intuitive one (StringContent). The reason most likely has to do with the jsoup algorithm, which has to first reconstruct a full HTML tree in order to perform an additional filtering. StringContent on the other hand simply treats the entire payload as a string leaving the "heavy work" to regular expression matching. This sums up to an interesting finding - the preferred/most intuitive way of doing things in ArchiveSpark might not always be the fastest. As such, knowledge of the internal implementation helps to make a better decision when extracting the corpus.

Full dataset

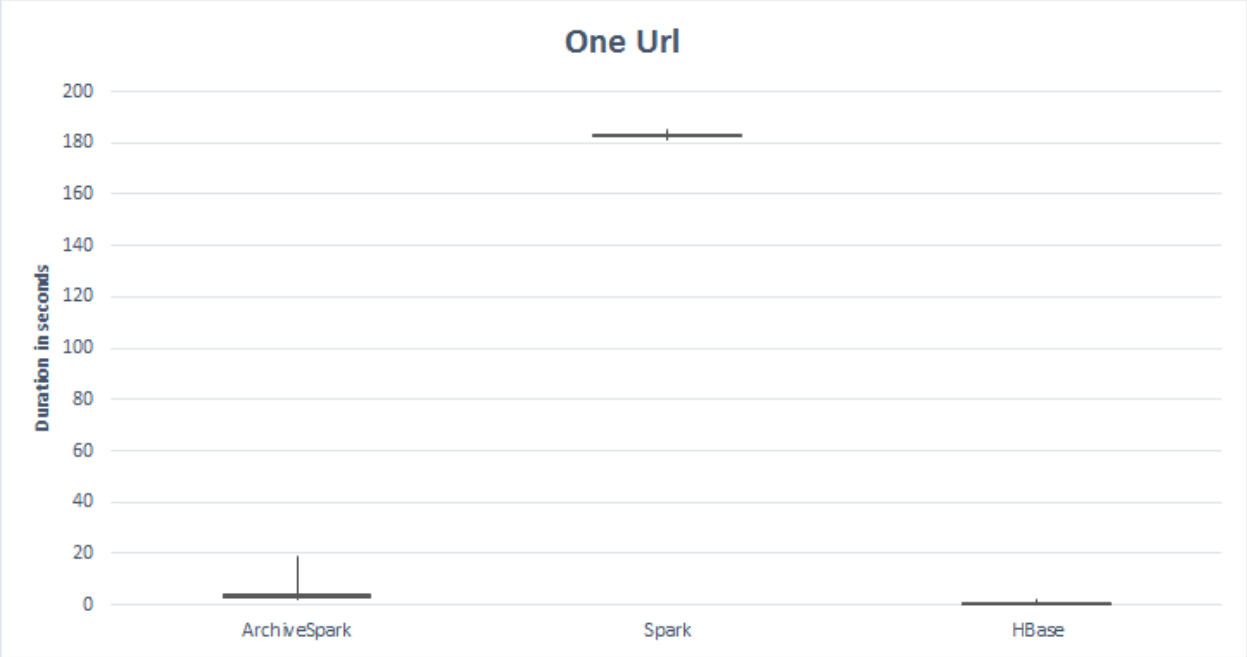


Fig 17. Full dataset - stock chart for one URL benchmark

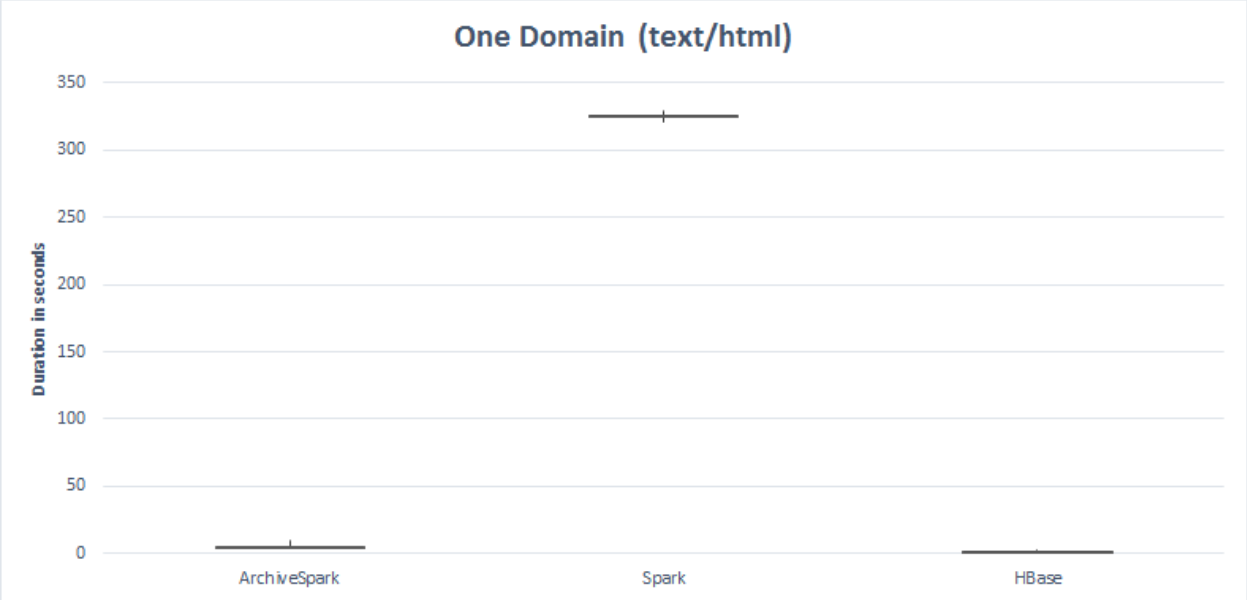


Fig 18. Full dataset - stock chart for one domain benchmark

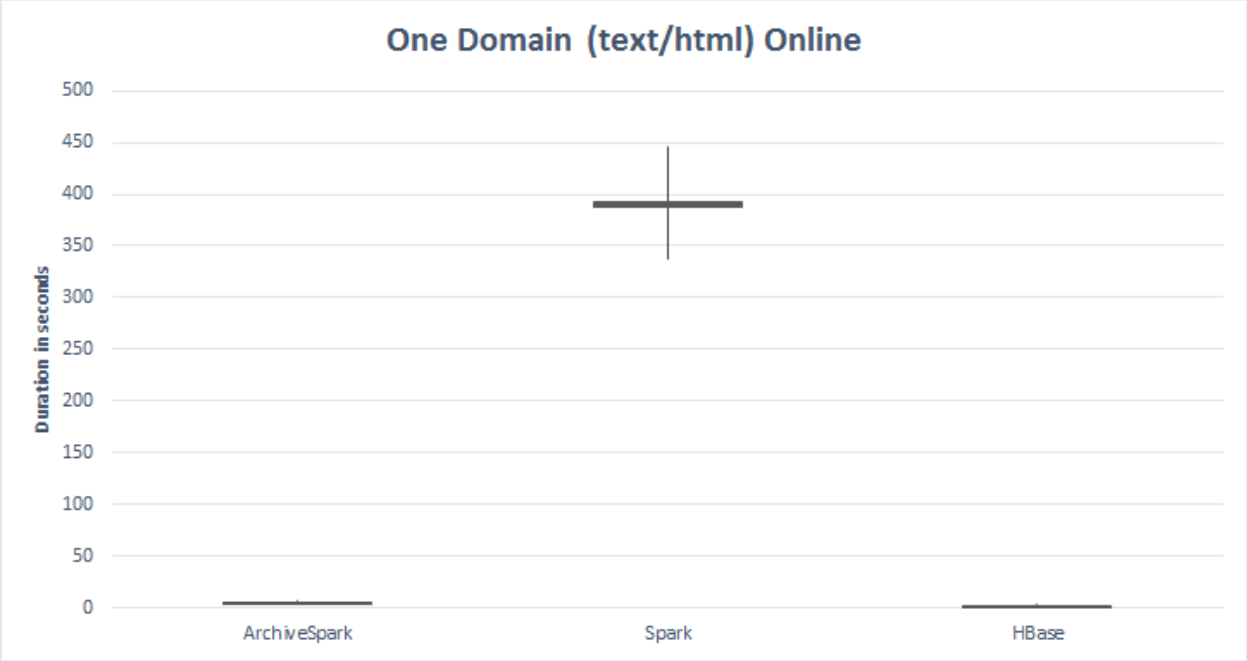


Fig 19. Full dataset - stock chart for one active domain benchmark

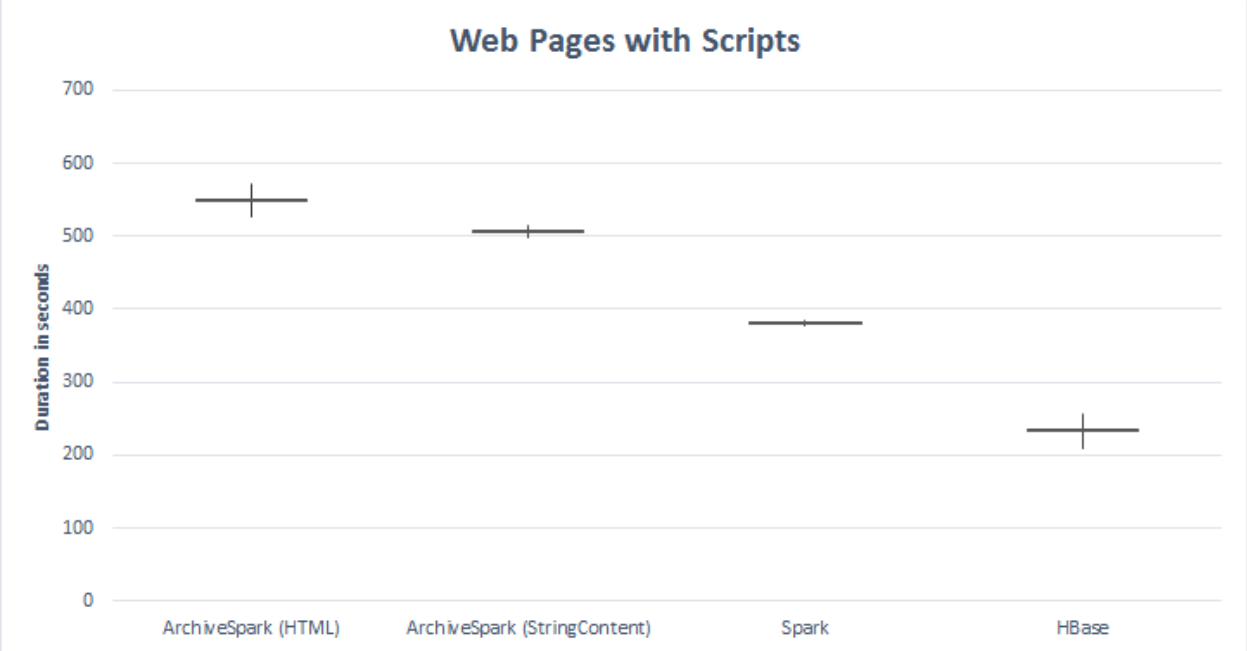


Fig 20. Full dataset - stock chart for pages with scripts benchmark

Spread charts for the full dataset don't really reveal too much interesting information. Still, several interesting findings involve the fact that Spark seems to be hurting quite a bit in terms of stability/consistency of the results when comparing one domain and one active domain benchmark - status code checking seems to be the main culprit here. Other than that it is fairly interesting that pure Spark seems to be the most stable in the last scenario even though it is not the fastest. Conversion of a binary array for HBase appears to be the most unstable - even when

compared to the ArchiveSpark (StringContent) scenario. Finally, without much surprise HTML enrichment's performance is much less predictable than StringContent's.

7. Enrichments (ArchiveSpark in Practice)

The final section of my report is intended for ArchiveSpark users. At this stage of its development ArchiveSpark still somewhat lacks in the department of documentation, which is why I believe this particular section should be very helpful for the new users interested in the framework. In addition to this section I also prepared a small demo including the majority of features mentioned below together with other content. This demo is provided as Jupyter Notebook containing code statements with explanations together with corresponding outputs, which makes it easy for everyone to replicate in their custom environment. Link to the demo as well as final presentation slides is accessible to all Virginia Tech students at this URL (you might need to switch your account if vt.edu is not your default Google one):

http://tinyurl.com/zejgc9f

Enrichments were introduced by ArchiveSpark as part of its incremental filtering workflow. They are by definition lazily executed and can also be composed in order to produce composite results (similarly to the composition of functions or UNIX pipelining). ArchiveSpark defines some of these as part of the framework but it also allows the user to quickly implement custom ones (can use custom 3rd party libraries).

7.1. Featured

This is the list of enrichments shipping with ArchiveSpark 2.1.0:

WarcPayload

- Extracts full content of WARC record replacing the original CDX record
- !! Only works with HDFS-based, local filesystem-based WARC records !!
- *Result*: new JSON record with the following fields:
    - httpStatusLine - contains status code + message (OK)
    - recordHeader - WARC-specific headers (see WARC/ARC section 5.1.)
    - payload - length of byte payload
    - httpHeader - HTTP-specific headers (only if HTTP response)
- *Usage:*
    - rdd.enrich(WarcPayload)

HttpPayload

- Extracts and appends HTTP-specific headers to the original CDX record
- !! Only works with Wayback Machine-based records !!
- *Result:* new JSON fields:
  - httpStatusLine - same as above
  - httpHeader - same as above
  - payload - same as above
- *Usage:*
  - rdd.enrich(HttpPayload)

StringContent

- Extracts payload as a string and appends it to the original CDX record
- *Result:* new nested JSON field:
  - payload.string - contains plain, unformatted string corresponding to WARC payload
- *Usage:*
  - rdd.enrich(StringContent)

Html(elementName)

- Generates HTML tree-like structure allowing to retrieve particular HTML element (both tag and content)
- *Result:* new nested JSON field:
  - payload.string.html.{elementName} : [ ] - for all
  - payload.string.html.{elementName} : value - for first
- *Usage:*
  - rdd.enrich(Html("div")) == rdd.enrich(Html.add("div"))
  - rdd.enrich(Html.first("div"))

HtmlText

- Extracts text/content of a particular HTML element
- !! Composable enrichment that depends on Html enrichment !!
- *Result:* new nested JSON field:
  - payload.string.html.{elementName}.text
- *Usage:*
  - rdd.enrich(HtmlText.of(Html.first("title")))

HtmlAttribute

- Extracts HTML attributes name/value pairs for a particular HTML element
- !! Composable enrichment that depends on Html enrichment !!
- *Result:* new nested JSON field:
  - payload.string.html.{elementName}.attributes.{attributeName} : {attributeValue}
- *Usage:*
  - rdd.enrich(HtmlAttribute("href").on(Html.first("a")))

Json[(objectProperty)]

- Parses string payload of WARC file creating dynamic element map, optionally filtered down to particular property
- !! Internally depends on StringContent that must return json-like string !!
- *Result:* new nested JSON field:
  - payload.string.json - just Json
  - payload.string.json.{objectProperty} - Json(objectProperty)
- *Usage:*
  - rdd.enrich(Json)
  - rdd.enrich(Json("Envelope.Payload-Metadata")

Entities

- Performs LDA analysis extracting entities from plain text
- !! Uses external library that doesn't come with ArchiveSpark - Stanford CoreNLP [30]. In order to use this enrichment this library must be on the classpath (included jar in spark-submit/spark-shell) !!
- *Result:* new nested JSON field:
  - Payload.string.entities
- *Usage:*
  - rdd.enrich(Entities.of(StringContent))

7.2. Custom

In addition to the above-mentioned enrichments, ArchiveSpark is flexible enough to allow users to define their own enrichments. The easiest way to achieve this is via the mapEnrich function call. The example below shows how mapEnrich functionality can be combined with one of the featured enrichments to create a totally new field on the record.

```
/** mapEnrich takes 2 parameters as well as actual function implementation (Scala partial function application.
    - 1st parameter - dependency field/function
    - 2nd parameter - target field

The result of this call is a new nested JSON field: payload.string.length that contains the length of string payload (result of StringContent)
**/
htmlOnline.mapEnrich(StringContent, "length") { content => content.length }
```

8. References

[1] ArchiveSpark, Efficient Web Archive Access, Extraction and Derivation; Helge Holzmann, Vinay Goel, Avishek Anand, JCDL ‘16 Proceedings of the 16th ACM/IEEE-CS on Joint Conference on Digital Libraries, Pages 83-92, http://dl.acm.org/citation.cfm?id=2910902
[2] Internet Archive; Accessed on December 8, 2016, https://archive.org/index.php
[3] L3S Research Center; Accessed on December 8, 2016, https://www.l3s.de/home
[4] Apache Spark; Accessed on December 8, 2016, http://spark.apache.org/
[5] ISO 28500; Publication date : 2009-05, Accessed on December 8, 2016, http://www.iso.org/iso/catalogue_detail.htm?csnumber=44717
[6] ARC file format; Publication date: September 15, 1996, Version 1.0, Authors: Mike Burner and Brewster Kahle, Accessed on December 8, 2016, http://archive.org/web/researcher/ArcFileFormat.php
[7] Sample WARC file; Accessed on December 8, 2016, http://netbootcamp.org/wp-content/uploads/WARC-file.png
[8] CDX writer; Accessed on December 8, 2016, https://github.com/rajbot/CDX-Writer
[9] Archive Metadata Extractor; Author: Brad Tofel, Accessed on December 8, 2016, https://webarchive.jira.com/wiki/display/Iresearch/archive-metadata-extractor.jar
[10] CDX and DAT legend; Accessed on December 8, 2016, https://archive.org/web/researcher/cdx_legend.php
[11] SBT, Scala interactive build tool; Accessed on December 8, 2016, http://www.scala-sbt.org/
[12] YARN; Publication date: September 4, 2015, Authors: Ray Chiang, Dennis Dawson, Accessed on December 8, 2016, http://blog.cloudera.com/blog/2015/09/untangling-apache-hadoop-yarn-part-1/
[13] Cloudera; Accessed on December 8, 2016, http://www.cloudera.com/
[14] Hortonworks; Accessed on December 8, 2016, http://hortonworks.com/
[15] ArchiveSpark Docker; Accessed on December 8, 2016, https://hub.docker.com/r/ibnesayeed/archivespark/
[16] Jupyter; Accessed on December 8, 2016, http://jupyter.org/
[17] Anaconda; Accessed on December 8, 2016, https://www.continuum.io/
[18] Warcbase; Accessed on December 8, 2016, https://lintool.github.io/warcbase-docs/
[19] Apache Maven; Accessed on December 8, 2016, https://maven.apache.org/
[20] Appassembler Maven plugin; Accessed on December 8, 2016, http://www.mojohaus.org/appassembler/appassembler-maven-plugin/
[21] HBase row key design; Accessed on December 8, 2016, http://archive.cloudera.com/cdh5/cdh/5/hbase-0.98.6-cdh5.3.8/book/rowkey.design.html
[22] Amazon AWS; Accessed on December 8, 2016, https://aws.amazon.com/?nc2=h_lg
[23] Cloudera Director Deployment Requirements; Accessed on December 8, 2016, https://www.cloudera.com/documentation/director/latest/topics/director_deployment_requirements.html#concept_fhh_ygd_nt
[24] SOCKS protocol; Accessed on December 8, 2016, https://en.wikipedia.org/wiki/SOCKS
[25] Deploying Cloudera Manager on AWS; Accessed on December 8, 2016, http://www.cloudera.com/documentation/director/latest/topics/director_get_started_aws_install_cm_cdh.html

[26] DLRL Hadoop cluster; Accessed on December 8, 2016, http://hadoop.dlib.vt.edu/
[27] Amazon EC2 Instance Types; Accessed on December 8, 2016, https://aws.amazon.com/ec2/instance-types/
[28] WIDE collection; Publication date: 2011-03-23, Accessed on December 8, 2016, https://archive.org/details/testWARCfiles
[29] jsoup HTML parser; Accessed on December 8, 2016, https://jsoup.org/
[30] Stanford CoreNLP; Accessed on December 8, 2016, http://stanfordnlp.github.io/CoreNLP/

9. Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 1319578: III: Small: Integrated Digital Event Archiving and Library (IDEAL); Grant No. 1619028, III: Small: Collaborative Research: Global Event and Trend Archive Research (GETAR); as well as work supported by the IMLS Grant LG-71-16-0037-16: Developing Library Cyberinfrastructure Strategy for Big Data Sharing and Reuse. I would like to thank all three Grants for their funding support, which had enabled the resources used in this project.

I especially wish to thank Dr. Edward A. Fox for his advice and guidance throughout the entire semester as well as Dr. Zhiwu Xie for his input and financial means allowing me to run my experiments on AWS.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.