Hadoop Project for IDEAL in CS5604

by

Jose Cadena
Mengsu Chen
Chengyuan Wen
{jcadena,mschen,wechyu88}@vt.edu

Completed as part of the course
CS5604: Information storage and retrieval
offered by
Dr. Edward Fox

Department of Computer Science
Virginia Tech
Blacksburg, VA
Spring 2015

Abstract

The Integrated Digital Event Archive and Library (IDEAL) system addresses the need for combining the best of digital library and archive technologies in support of stakeholders who are remembering and/or studying important events. It leverages and extends the capabilities of the Internet Archive to develop spontaneous event collections that can be permanently archived as well as searched and accessed. IDEAL connects the processing of tweets and web pages, combining informal and formal media to support building collections on chosen general or specific events. Integrated services include topic identification, categorization (building upon special ontologies being devised), clustering, and visualization of data, information, and context. The objective for the course is to build a state-of-the-art information retrieval system in support of the IDEAL project. Students were assigned to eight teams, each of which focused on a different part of the system to be built. These teams were Solr, Classification, Hadoop, Noise Reduction, LDA, Clustering, Social Networks, and NER. As the Hadoop team, our focus is on making the information retrieval system scalable to large datasets by taking advantage of the distributed computing capabilities of the Apache Hadoop framework. We design and put in place a general schema for storing and updating data stored in our Hadoop cluster. Throughout the project, we coordinate with other teams to help them make use of readily available machine learning software for Hadoop, and we also provide support for using MapReduce. We found that different teams were able to easily integrate their results in the design we developed and that uploading these results into a data store for communication with Solr can be done, in the best cases, in a few seconds. We conclude that Hadoop is an appropriate framework for the IDEAL project; however, we also recommend exploring the use of the Spark framework.

Acknowledgements

We would like to thank the Solr and Noise Reduction teams for their continuous collaboration throughout the semester. Even though, we worked with all the teams in the class, the Solr and Noise Reduction teams played a key role on helping us define the schemas for HBase and Avro. We thank Rich, Ananya, and Nikhil (Solr team) for patiently explaining the basics of Solr and helping us understand what data should and should not go into the system, and we thank Prashant and Xiangwen (Noise Reduction team) for giving us the initial idea of using Avro to standardize the output produced by the different teams and for later bearing with our numerous changes to the schema and adjusting their noise-reduction process accordingly.

We would like to thank Sunshin Lee, the graduate teaching assistant and cluster administrator, for his helpful suggestions and technical expertise in matters related to Hadoop. Especially in the early stages of the course, when we had very little experience with Hadoop, Sunshin patiently explained the different Hadoop tools to us and pointed us to the appropriate resources. Thank you for keeping us in the right track.

We would like to thank Dr. Fox for bringing this project to us and for his guidance throughout the semester. Dr. Fox helped us define tasks and focus areas for our team. Thank you for being the *guide on the side*. More generally, we thank Dr. Fox for making the IDEAL project part of the *Information Storage and Retrieval* course. By the end of the class, we felt we had a considerable understanding of the Hadoop framework, both in theory and practice. We doubt we would have gained a similar understanding in a traditional course format based on lectures and assignments.

Finally, we thank the sponsors for making the IDEAL project possible. This project was sponsored by the National Science Foundation **NSF grant IIS - 1319578, III: Small: Integrated Digital Event Archiving and Library (IDEAL).**

1. Project Overview

1.1 Project Effort

The efforts of our team are focused on designing and implementing a solution for the IDEAL project that runs on a Hadoop cluster and takes advantage of the distributed computing and storage capabilities of the cluster to handle large amounts of Twitter and web page data.

In coordination with the other teams and the instructor, we define the data workflow, from data ingestion and storage in the cluster to indexing into Solr. Every team in the class is going to be involved in different parts of this workflow. For example, each team will download web pages for two collections assigned to them, and the various teams in charge of data analysis tasks will produce results to be used by Solr. Our part in this workflow is loading data produced by the various teams into HBase (see Section 2), from where the data will later be indexed into Solr.

Our project efforts also include helping other teams to make use of the distributed computing capabilities of Hadoop, so that the collections that we are working with are processed efficiently.

1.2 Challenges

There are various challenges when it comes to designing a data workflow for our project. First, the design decisions have to consider how the many moving parts of this project fit together for our goal of building a state-of-the-art search engine. Second, putting the design together requires knowing what the different storage options for Hadoop are, understanding details of the architecture of these different options, and considering how the strengths and weaknesses of each alternative will affect our project. Third, designing the data workflow involves having awareness of the existing data processing tools for Hadoop and adjusting the design to make use of these tools when appropriate.

The point of understanding different options of data storage and file formats is raised once again in our task of loading data into HBase. Potentially, every team in charge of data analysis will have a preferred file format to take as input data and will produce output in a unique form. We have to implement tools and conventions to ensure that the different outputs produced by each team agree with the HBase schema, while, at the same time, minimizing the overhead of data processing for the other teams.

As mentioned in the previous section, one of our goals is efficient processing of the data collections in the Hadoop cluster. There are two main challenges associated with this goal. First, we do not want the tasks of other teams to be hindered or delayed by optimization concerns. As much as possible, optimization tasks should be transparent to the rest of the teams. Second, optimizing performance requires analyzing runtime statistics of the various jobs run on the cluster so that we can compare the effect of different set-ups. We have to understand how the logging protocols of Hadoop and be able to do basic analysis of the log files of the various tools used in the project.

1.3 Solutions Developed

In this section, we summarize our contributions. For architectural and implementation details, we point the reader to Sections 5, 6 and 7.

1.3.1 Data Workflow (Section 7.1.2)

Our solution starts by loading Twitter data into the Hadoop cluster from a relational database using Sqoop, a tool for bulk loading structured data into Hadoop. Web page data is fetched and stored in HDFS using Nutch, a web-crawling tool. Once the data is on disk, the Noise Reduction team will produce a *clean* version of the data that will 1) be loaded into HBase and 2) used by other teams for their respective data analysis tasks. Other teams will save their results in a predefined format (Avro files [18]) that we will later load into HBase. A description of our predefined format can be found in Section 7.2.3, and instructions on how to manipulate Avro files can be found in Section 6.2. As new data is added to HBase, it will be indexed into Solr in real-time using the hbase-indexer tool. When the data is indexed in Solr, it can be queried, and results will be produced according to a scoring function that incorporates all of the results from the data analysis phase.

1.3.2 Loading Data Into HBase (Section 7.4.2)

Over the course of the semester, we implemented three programs to load data into HBase. We first developed a centralized (i.e., not distributed) Java program that simply reads records from one of the Avro files produced by the teams and writes data to HBase through an HBase API for Java; we were able to load all the small tweet collections using this program, with each collection taking less than one minute. However, our implementation did not scale to the big collections ---we ran into memory limitations--- so we developed a MapReduce program that reads Avro files from HDFS and uploads records into HBase in a distributed fashion, again, through an HBase API. Our MapReduce implementation loads each collection in at most 30 minutes. Finally, we developed a solution that makes use of the bulk load tools that come with the HBase libraries. This solution involves transforming an Avro file into HFile format ---a native low-level HBase format--- determining an appropriate number of splits for the HBase tables into regions, and uploading the HFiles into the different regions. For this task, we write a MapReduce job where the Map function creates the HFiles and the Reduce function uses the HBase libraries to send the HFiles to the corresponding regions.

1.3.3 Collaboration With Other Teams (Section 5.4)

We interacted with all the other teams in the class. A big portion of these interactions involved helping the teams work with Avro files. Avro is a serialized format, so it is not immediately readable to a human. We instructed most teams on how to interpret these Avro files by using the libraries in the Hadoop cluster. Similarly, we helped the teams to adapt their workflow to read and write Avro files; through the semester, we went over multiple revisions of the output format with each team. Another part of part of our collaboration with other teams was sketching out the schema and workflow for the class and helping them with MapReduce programming for their respective tasks. Below, we give a brief summary of our interactions with each team.

1.4 Roadmap

The rest of this report is organized as follows. In Section 2, we give a **literature review** covering the **basic concepts of Hadoop** and the origins of this infrastructure, the **role of Hadoop in information storage and retrieval**, and a brief description of **useful tools for Hadoop**. We also point the user to some useful **resources to start learning Hadoop**. In Section 3, we state the **requirements** of our team for this class. In Section 4, we give a high-level overview of our **proposed design, background knowledge, and technical tools** used for the project. Then, in Section 5, we give details about the **implementation** of our project, including a detailed **timeline** of our work each week, an **evaluation** of our implementation, and details of our **collaboration with the other teams in the class**. Section 6 is a **user's manual,** which, we hope, will be valuable to the reader interested in learning how to **work with Avro files,** interact with **HBase,** and **use Nutch to crawl web pages.** Section 7 is a **developer's manual;** here, the reader can get **technical specifications of the Hadoop cluster used in the course.** This section is also of interest to readers who want to know how to **install Solr and Nutch,** two main tools that we used as part of the course. It is also in this section where we provide specific **details about our design for Avro, HBase, and the data flow for the IDEAL project.** We close by describing our solutions for **loading data into HBase**. Finally, in Section 8, we give some **conclusions** based on our experiences in the class, and we also pose some directions for **future work.**

2. Literature Review

Our literature review is divided into three parts. First, we describe the Hadoop framework and its main features. Then, we discuss how Hadoop interacts with information storage and retrieval tools. Finally, we present a non-exhaustive list of references that explains how to set up and start using Hadoop. After reading this section, the reader should 1) understand, at a high level, what Hadoop is and its advantages, 2) have a general awareness of the readily available software packages that can be used with Hadoop, and 3) know where to find resources to start using Hadoop and MapReduce.

2.1 What is Hadoop?

As described in the Apache Hadoop website [19], Hadoop is "a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models." Sometimes, we also call it the Hadoop (software) stack or Hadoop ecosystem. The two core components of the Hadoop ecosystem are the Hadoop Distributed File System (HDFS) and MapReduce, which were inspired by the Google File System [4] and Google MapReduce [2], respectively. HDFS provides a scalable, fault-tolerant way to store large volumes of data on a cluster. Hadoop MapReduce is a paradigm to process data in parallel in a distributed system.

Hadoop is useful to tackle computational tasks involving big datasets (i.e., on the scale of hundreds of gigabytes, terabytes, or more). When working with such datasets, there are two notable challenges. First, storing the data in its totality in one single disk becomes problematic; fitting the data into memory is even more challenging. Second, even if one can afford expensive hardware to store the data, it would take a prohibitively large amount of time to complete any kind of useful analysis. Hadoop solves the first problem by providing a distributed file system (HDFS). This way, a user is able to store big collections of data in a cluster of commodity software. Hadoop also provides a framework to tackle the second problem (computation on a big dataset), namely Hadoop MapReduce.

It is important to note that the operations related to distributing the data across nodes, reacting to errors in hardware, and scheduling details of MapReduce jobs, are transparent to the user. Hadoop takes care of most of the low-level tasks and provides simple interfaces to interact with the system. Other Apache projects, such as Hive [20], Pig [21], or Zookeeper [22], further abstract common programming tasks, making it even easier to analyze big datasets.

2.2 Hadoop for Information Storage and Retrieval

The distributed computing and storage capabilities of Hadoop have been used to make off-the-shelf information storage and retrieval tools more scalable. For the purposes of this project, we focus on HBase, Nutch, Solr, and Mahout.

HBase is a data storage system inspired by Google's BigTable [1]. HBase extends HDFS by allowing real-time random IO operations, whereas HDFS assumes that data will only be written once and always processed in bulk. Data in HBase is organized as a multi-dimensional map. A table is a map of *row ids* to column families; each *column family* in turn is a map from columns to values[1]. HBase has the properties of being distributed (over a Hadoop cluster), scalable, and sparse (i.e., a column only exists in a row if it has a value; there are no NULL columns as in a relational database). These properties have made HBase the tool of choice for companies like Facebook, Yahoo, and Twitter.

Solr [15] is an open source search platform developed by Apache. This tool is the core of the search engine that we develop in this class project. Solr can be configured to ingest and index data from a Hadoop cluster, allowing us to extend this powerful platform and all its readily-available tools to collections that do not fit in one single server.

Nutch [14] is a tool for large-scale web crawling. Given a collection of URLs, Nutch recursively traverses these URLs making it very simple to create large collections of web pages. Nutch includes tools for basic processing of the crawled data, and it readily supports integration with Hadoop and Solr. We use Nutch to efficiently fetch web pages from the URLs found in our tweet collections. However, we do not make use of the web-crawling capabilities of this tool.

Mahout [23] is a machine-learning library. The library contains many standard machine learning algorithms for classification, topic modelling, and clustering, among others. Furthermore, most of the algorithms in the library already have MapReduce implementations, so it is possible to run basic machine learning tasks on big datasets with little programming demand.

[1]To be accurate, a column maps to a map of timestamps to values. In other words, each column supports versioning by keeping track of its current and past values.

We finish this section by noting that, if there is not a readily-available implementation of a desired algorithm, a developer can always write the appropriate MapReduce program from scratch. The book *MapReduce: Design Patterns* [8] provides a good compilation of different MapReduce patterns to use for many data manipulation tasks. Also, a developer is not constrained to using Java as the programming language, since Hadoop has *streaming* packages to run code written in other languages. Therefore, a user can take advantage of other packages or libraries that he/she already knows how to use.

2.3 Getting Started with Hadoop

There are many tutorials on how to set up a Hadoop cluster and run basic programs. The Apache foundation offers a tutorial to set up a single node cluster [13]. For a more informal, "Quick Start" style tutorial, we suggest reading reference [12] to the interested reader. In order to take advantage of Hadoop, a user must learn how to interact with HDFS; the Yahoo! Developer Network [9] provides a good introduction. Understanding of the MapReduce paradigm is also a must. As stated above, the book *MapReduce: Design Patterns* provides a broad collection of examples and explanations of the basic MapReduce concepts. Readers specifically interested in text processing will benefit from the book *Data-Intensive Text Processing with MapReduce* [5]. Finally, users interested in the low-level details about index construction in Hadoop can use Chapter 4 of *Introduction to Information Storage and Retrieval* [7] as a reference.

3. Requirements

IDEAL is a Big Data project. One primary goal of the project is to make it possible for users to extract relevant content from collections on the scale of terabytes. There are various challenges to consider when working with this amount of data. As discussed in Section 2, it is not possible to store all the data in a single commodity disk, let alone load it into memory for any processing or data analytics task.

As the Hadoop team, our objective is to make information retrieval scalable in the IDEAL project. We work with the rest of the teams to help them parallelize their respective tasks as much as possible. Additionally, we are responsible for designing a general schema to store the data in the Hadoop cluster. The goal is that teams modify a unified data representation instead of producing disjoint results across the system. In designing the schema, we collaborate with the Solr team; we also work with the Solr team on indexing and loading the data from the cluster into Solr. On the user-support side, we help the teams to use tools in the cluster, such as Mahout, Nutch, and avro-tools. We also provide assistance on writing MapReduce programs for tasks that are not readily available in Mahout.

Below, we summarize our tasks for the project:

- Design a schema for the storage of Twitter data and web page data.
    - Decide on whether to use HDFS, HBase, or some other framework.
- Instruct other teams about the schema and propose data formatting standards.

- Load data into the cluster.
    - Coordinate with the cluster administrator (Sunshin Lee) for this requirement.
- Load data into HBase.
- Coordinate with the other teams to make sure that they take advantage of the parallel computing capabilities of Hadoop.
- Provide support to other teams for writing and running MapReduce jobs.

4. Design

4.1 Approach

Our approach is to have a workflow where teams in charge of data analysis read and write data from/to HDFS. Teams interact with HBase only through a data-loading tool that we provide. The data in HBase is a structured representation of all our collections containing only the data required by Solr for query processing. As the data is uploaded and updated in HBase, the changes are indexed in real time in Solr via the Lily HBase indexer [10].

4.2 Tools

Programming languages: Java and Python
HDFS (Hadoop File System): Distributed file system. Files are stored across a cluster of computers.
HBase: Non-relational database. HBase is sparse, scalable, and well-integrated in the Hadoop ecosystem.
Lily HBase Indexer: Tool for indexing HBase rows into Solr.
Sqoop: Tool for transferring data in bulk from a database to HDFS.
Nutch: Web crawler. We will use it to fetch web pages from a collection of URLs.

4.3 Methodology

Loading data into the cluster
The original Twitter data was stored in a relational database at the beginning of the semester. Sunshin Lee, the cluster administrator, used Sqoop to copy the data to HDFS as AVRO files, sequence files, and comma-separated values (CSV). After that, Nutch is to crawl web pages corresponding to the URLs extracted from the HDFS tweets. These webpages are stored in HDFS as WARC files. Web pages that are in text form (e.g., ending with .htm or .txt) also are stored in HDFS as HTML and text files. From here, the noise reduction team processes these files to discard irrelevant content as much as possible. Then, other teams can use the “clean” files for their respective machine learning tasks.
Communication with Solr
Each team in charge of data analysis reads data from HDFS and writes interim results back to HDFS. These results are then added to the corresponding tweet / webpage in HBase. The Lily HBase indexer automatically updates the Solr index.

4.4 Conceptual Background

- Fundamentals of HBase and Google Bigtable.
- Architecture of HBase and HDFS.
- MapReduce paradigm.
- Internal working of the HBase Indexer.

4.5 Deliverables

- Data workflow design.
- Avro file conventions.
- Programs for uploading data into HBase.
- Performance metrics and optimization suggestions.

5. Implementation

5.1 Timeline

Week 1: Get Solr running on our laptop. (see Section 7.3)
Week 2: Set up a Hadoop pseudo-cluster to practice Hadoop (Section 7.3)
Week 3: Reorganize the report of the previous week and start learning Mahout.
Weeks 4 and 5:

- Use Python script to download web pages mentioned in tweets.
- Index web pages and tweets into Solr.
- Research different options to store the data in the cluster.
    - Data will be stored in HDFS as HTML, WARC, or CSV fields.
    - We recommend HBase for communicating with Solr.
- Research Apache Nutch to crawl web pages instead of using a Python script. (see Section 7.2 Data)

Weeks 6 and 7 and 8:

- Created sample HBase tables via the HBase shell and the Java API.
- Researched different options and data formats for loading data into HBase.
- Finalized details of the data workflow with other teams.
- Implemented prototype workflow for indexing data from HBase to Solr.
- Implemented prototype web page fetching using Nutch.
- Learned how to use the Lily indexer to synchronize Solr and HBase.
- Defined details of the HBase schema with the Solr team.

Week 9:

- Talked to Sunshin about using Sqoop and Nutch to load data into HDFS. Sunshin loaded all the collections, and teams will extract web pages for their own collections.
- Learned how to interact with HBase programmatically in order to load data from HDFS into HBase.
- Designed Avro schemas for each team.
- Wrote documentation for the schemas and a tutorial on how to create Avro files using the schemas.
- Wrote a program to sequentially convert tweets in TSV format to Avro.

- Wrote a program to sequentially load Avro data into HBase. The program loaded our small tweet collection in about 1 minute (380,403 tweets in Avro format).

Week 10:

- Wrote a program to extract and expand shorten URLs from big collections.
- Crawled the web pages for our big collection.
- Wrote a MapReduce program to load Avro data into HBase.
- Loaded cleaned (i.e., after noise reduction) tweets into HBase.

Week 11

- Modified Avro schema due to changes in HBase schema.
- Loaded all the small collections into HBase.
- Continued working with other teams in producing Avro files.
    - **Noise reducing team**: Done with small collection.
    - **Classification team:** Discussed, waiting their output.
    - **NER team**: Provided help on modifying their output avro schema, waiting feedback.

Week 12

- Modified Avro schema for LDA team from conversations with LDA and Solr teams.
- Tested and debugged MapReduce HBase upload program.
- Continued working with other teams in producing Avro files.
    - **Noise reducing team**: They are working to produce clean web page data.
    - **Classification team:** Discussed, waiting for their output.
    - **NER team:** Provided help on modifying their output avro schema, waiting feedback.
    - **LDA team:** We agreed on the Avro schema for them. They have output to be loaded into HBase.
    - **Clustering:** They will produce files to be loaded into HBase.

Week 13

- Resolved issues of running our MapReduce HBase upload program.
- Loaded big tweet collections into HBase: 85,589,755 rows in total.
- Coordinated with Solr team to indexed data in HBase into Solr, the small tweet collection has been indexed.
- Continued working with other teams in producing Avro files.
    - **Noise reducing team**: Loaded cleaned small web page collections.
    - **Classification team:** Waiting for their output.
    - **NER team:** Loaded their output for tweets into HBase.
    - **LDA team:** We agreed on the Avro schema for them. They have output to be loaded into HBase. Waiting for their output.
    - **Clustering:** Loaded their output for tweets into HBase.
    - **Social Network:** We agreed on a format with them. We are waiting for their output.

Weeks 14 and 15

- Implemented a bulk-loading program to write data into HBase directly (i.e., bypassing the HBase write path).

- Integrated the *pangool* software library into our infrastructure for the Classification team.
- Continued working with other teams in loading HBase
    - **Noise reducing team**: Loaded cleaned big web page collections into HBase.
    - **Classification team:** Helped them with MapReduce programming and loaded their classification results into HBase.
    - **NER team:** Loaded their output for web pages into HBase.
    - **LDA team:** Loaded their output for tweets into HBase.
    - **Clustering:** loaded their output for web pages into HBase.
    - **Social Network:** Loaded their output for tweets and web pages into HBase.

5.2 Milestones and Deliverables

- Tools for loading data into HBase.
    - We developed a sequential program, a MapReduce program, and a bulk-loading program.
- HBase schema and synchronization with Solr.
    - We developed a schema for HBase and for each team based on the needs of the project.
- Optimization of other team's tasks.
    - We worked with the NER and Classification team to integrate their tools into our workflow and avoiding wasted disk space. Other teams used existing tools for MapReduce, but we did not optimize their jobs.

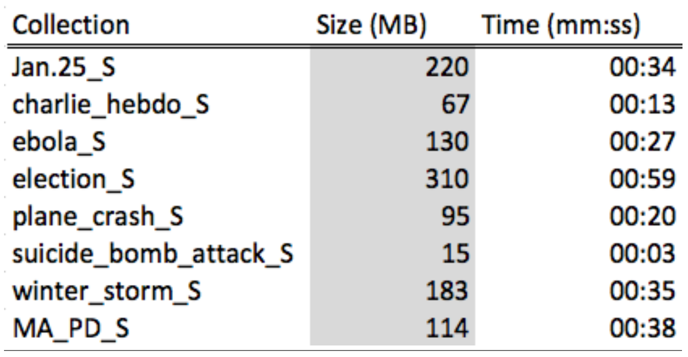
5.3 Evaluation

We report the performance (in terms of running time) of our three HBase loading programs. We focus on the tweet collections because they were larger and more challenging to handle. However, we provide loading times for the web page collections in Appendix D. Also, we just show the sizes of the collections that we upload and the time taken. A detailed description of the format of these collections can be found in Section 7.2, and description of the implementation of each program can be found in Section 7.4 and the Appendix.

5.3.1 Non-Distributed Program

Our first solution was a non-distributed Java program that reads Avro files from the main node of the cluster and writes data to HBase by invoking the HBase API for Java. The process should be familiar to anyone who has written programs to communicate with a database (relational or otherwise) through an API.
We were able to load all the small tweet collections, after being processed by the Noise Reduction team, using our non-distributed program. Table 1 reports the time to load each collection into HBase. Each collection took less than one minute to be uploaded, but most of the big collections could not be loaded with this program due to memory constraints in the main node.
We could use this program to load the big collections by adding more memory to the main node or breaking down the input files into smaller pieces ---effectively "distributing" the load by hand. However, both ideas are just temporary solutions; as the project grows, at some point, we are not going to be able scale anymore; furthermore, so far, we have not used the distributed

computing capabilities of the Hadoop cluster. In the next section, we show a distributed approach to load the data.



Collection	Size (MB)	Time (mm:ss)
Jan.25_S	220	00:34
charlie_hebdo_S	67	00:13
ebola_S	130	00:27
election_S	310	00:59
plane_crash_S	95	00:20
suicide_bomb_attack_S	15	00:03
winter_storm_S	183	00:35
MA_PD_S	114	00:38

Table 1. Time to load the small tweet collections into HBase using a non-distributed program

5.3.2 MapReduce Program

Our second implementation was a MapReduce program that reads Avro files from HDFS and writes data to HBase by using the HBase API for Java. This program only has a Map function (without a Reduce). Each mapper writes exactly one record to the data store. We were able to load all the big collections to HBase, with the largest collection taking 30 minutes. For comparison, we also tested loading the small collections with our MapReduce program. We note that, it is rather unnecessary to use the Mapreduce framework to process data that can be handled by a non-distributed program in only a few seconds. There is an appreciable cost in running a distributed program (i.e., communication between nodes, scheduling and supervising tasks across the cluster, etc.) that is not justified for small files. Table 2. reports the times for each collection; we also show the time taken by loading all the collections at once (i.e. all big and small collections in a single MapReduce job). There are two insights from this results that we want to emphasize. First, as discussed above, we don't gain anything by processing the small collections with MapReduce; in fact the log files for these jobs show that most of the small collections are being processed by a single mapper, which is no different than using a non-distributed program ---except that we still pay the overhead of managing the MapReduce job. Second, loading all the data at once is much faster than loading collections one at a time. It is preferable to wait until we have multiple collections before loading data into HBase.

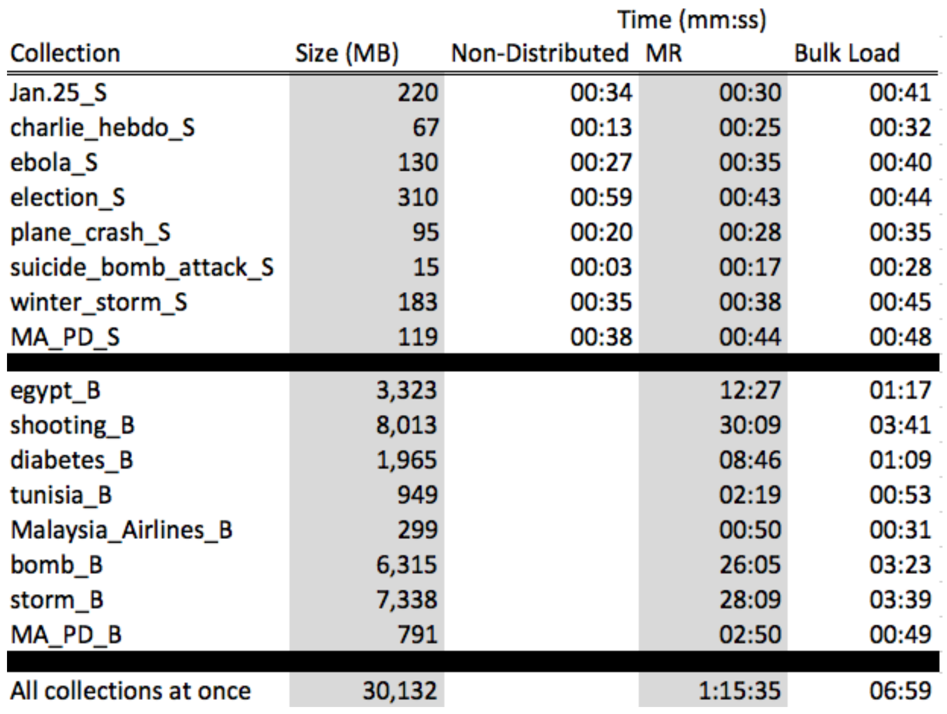
		Time (mm:ss)	
Collection	Size (MB)	Non-Distributed	MR
Jan.25_S	220	00:34	00:30
charlie_hebdo_S	67	00:13	00:25
ebola_S	130	00:27	00:35
election_S	310	00:59	00:43
plane_crash_S	95	00:20	00:28
suicide_bomb_attack_S	15	00:03	00:17
winter_storm_S	183	00:35	00:38
MA_PD_S	119	00:38	00:44
egypt_B	3,323		12:27
shooting_B	8,013		30:09
diabetes_B	1,965		08:46
tunisia_B	949		02:19
Malaysia_Airlines_B	299		00:50
bomb_B	6,315		26:05
storm_B	7,338		28:09
MA_PD_B	791		02:50
All collections at once	30,132		1:15:35

Table 2. Time to load all the tweet collections to HBase using MapReduce

5.3.3 Bulk-Load Method

HBase has a tool that allows a developer to directly upload HFiles, native HBase files, to the data store, effectively bypassing the normal workflow that is followed when writing to HBase through the API. We describe these ideas and its pros and cons in more detail in Section 7.4. For the purposes of this section, we just emphasize that bulk-loading is a much more efficient way to load large amounts of data into HBase than the API.

In order to use the bulk-loading tool, we first need to convert our Avro files to HFiles, which we did by using a MapReduce program. Then, we upload these files to HBase using the libraries included in the Hadoop distribution (see Section 7.3). Table 3 (rightmost column) reports the time taken to convert each collection to an HFile using our program. We are not including the time it takes to upload the HFiles to HBase because we consider it negligible. Loading the HFiles for individual collections takes two or three seconds, and loading the HFile for all the collections combined takes only six seconds. Even with this added upload time, the improvement from our MapReduce program is Section 5.3.2 is noticeable. With the bulk load approach, all the big collections can be loaded in around 7 minutes, whereas it takes well over one hour to do the same through the API.



		Time (mm:ss)		
Collection	Size (MB)	Non-Distributed	MR	Bulk Load
Jan.25_S	220	00:34	00:30	00:41
charlie_hebdo_S	67	00:13	00:25	00:32
ebola_S	130	00:27	00:35	00:40
election_S	310	00:59	00:43	00:44
plane_crash_S	95	00:20	00:28	00:35
suicide_bomb_attack_S	15	00:03	00:17	00:28
winter_storm_S	183	00:35	00:38	00:45
MA_PD_S	119	00:38	00:44	00:48
egypt_B	3,323		12:27	01:17
shooting_B	8,013		30:09	03:41
diabetes_B	1,965		08:46	01:09
tunisia_B	949		02:19	00:53
Malaysia_Airlines_B	299		00:50	00:31
bomb_B	6,315		26:05	03:23
storm_B	7,338		28:09	03:39
MA_PD_B	791		02:50	00:49
All collections at once	30,132		1:15:35	06:59

Table 3. Comparing our three solutions for uploading data to HBase

5.4 Collaboration With Other Teams

5.4.1 Solr

We were in constant collaboration with the Solr team at different stages of the project. In the initial weeks, we discussed workflows for the project and explored different alternatives to index the results of each team into Solr. Towards the end of the planning phase, we had to choose either to read data directly from HDFS or to add HBase to the workflow to make use of the real-time Lily Indexer tool. In class discussions, we decided to do the latter. After the workflow was decided, we collaborated with the Solr team for the design of the schema for the HBase tables and Avro files. We made many of our design decisions based on the information that the Solr team needed to have indexed into Solr. In the later stages of the semester, when we had most of the data loaded into HBase, we helped the Solr team with generating test tables (i.e., samples of the complete data) that they could use for prototyping and testing.

5.4.2 Noise Reduction

Similar to the Solr team, most of our work with the Noise Reduction (NR) team involved schema design and data formatting considerations. The NR team is responsible for generating the input files for most of the other teams; they are the first team in “touching” the data. Therefore, it was important to agree on a design and workflow with this team as early as possible. Furthermore, it was important to put special consideration into the schema of the NR team in order to avoid significant subsequent changes to their output files. The scenario that we wanted to prevent was having to ask the rest of the teams to redo their analysis of the data just because we forgot to include an important field in the initial output from NR. Fortunately, we did not encounter this problem.

5.4.3 Classification

The classification team had to use a third-party software library, *pangool* [3], for completing their task. We helped the classification team by adapting a Naive Bayes implementation in *pangool* to our infrastructure. To summarize, we modified the Naive Bayes MapReduce program to use our Avro libraries for reading and writing data according to our pre-specified schema. Details of these changes can be found in Section 7.

5.4.4 NER

Most of our interactions with the NER team were helping them to use Avro files and follow our schema conventions. We showed this team how to use *avro-tools,* an Avro utility, to convert Avro to a readable format, concatenate small Avro files in HDFS to avoid wasted space, and generate Avro from a given schema. We also helped this team to produce Avro files in a MapReduce program.

5.4.5 LDA, Clustering, and Social Networks

With the remaining teams, our interactions were mostly to decide details of their respective schemas and upload their results into HBase. The Clustering and LDA teams used Mahout, so they did not have to write MapReduce programs for their rspective tasks. It is likely that the Mahout jobs generated by these teams could be sped up; however, due to time limitations, we leave this performance tuning for future work. The Social Networks team did not use the MapReduce because this framework is not a good solution for the kind of analysis they were required to do (i.e., iterative graph algorithms). Instead, they used the Graphx library [24] for Spark [16].

6. User’s Manual

This section provides a guide for using the Hadoop cluster for this project. We present instructions and examples on how to use HBase, Avro files, and Nutch.

For the rest of the section, we assume that the user is working on the Cloudera Virtual Machine or the Hadoop cluster for this class.

6.1 The HBase Shell

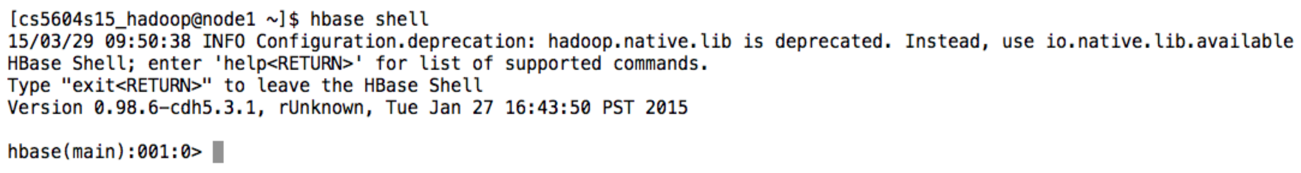
The HBase shell (or just “the shell”) is a command-line utility where a user can execute run commands to interact with HBase. Through the shell, a user can create or delete tables, update or remove data in the existing tables, get data stored in HBase, among other common operations. Users familiar with command-line utilities to interact with SQL databases should find the HBase shell familiar and easy to use despite the fact that HBase is a NoSQL data store.

6.2.1 Running the HBase Shell

We start the HBase shell by using the command

```
$ hbase shell
```

We should see the output shown in Figure 1.



```
[cs5604s15_hadoop@node1 ~]$ hbase shell
15/03/29 09:50:38 INFO Configuration.deprecation: hadoop.native.lib is deprecated. Instead, use io.native.lib.available
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.6-cdh5.3.1, rUnknown, Tue Jan 27 16:43:50 PST 2015

hbase(main):001:0>
```

Figure 1. Starting the HBase shell.

Now that the we are in the HBase shell, we are ready to interact with HBase.

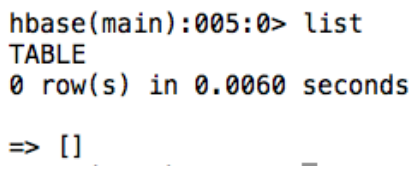
Note: If the commands in the next section do not work as expected, it may be because HBase is not running in your system. In the virtual machine, you can check whether HBase is running by pointing your browser to localhost:60010. If HBase is down, you will not be able to connect. You can start HBase by running the following commands:

```
/usr/lib/hbase/bin/hbase-daemon.sh start master
/usr/lib/hbase/bin/hbase-daemon.sh start regionserver
```

We check the browser again to see that HBase is running now.

6.2.2 Common Operations

First, let’s **get a list of the existing tables**. If this is your first time using HBase in your system, there should not be any tables yet. We get a list of the tables using the “list” command, which should return 0 rows as output (see Figure 2).



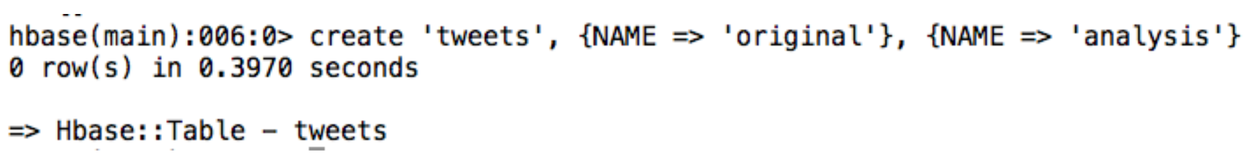
```
hbase(main):005:0> list
TABLE
0 row(s) in 0.0060 seconds

=> []
```

Figure 2. HBase does not have any tables yet.

Now, we will **create our first HBase table.** We will create a table for tweets with two column families: “original” and “analysis”. We will use the “create” command, which has syntax

```
create ‘table_name’ , [{NAME => ‘col_family_1’}, … , {NAME => ‘col_family_n’}]
```

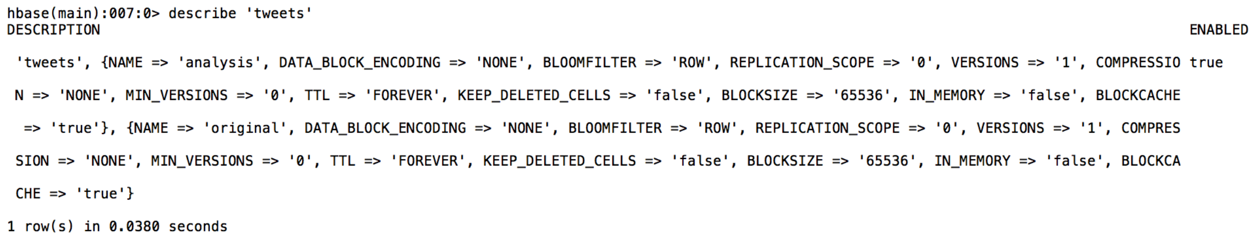
In this case, we are giving the name of the table and a list of column families. We note that there are other parameters for the “**create**” command, so you should check the HBase documentation. We show the command to create the “tweets” table in Figure 3.



```
hbase(main):006:0> create 'tweets', {NAME => 'original'}, {NAME => 'analysis'}
0 row(s) in 0.3970 seconds

=> Hbase::Table - tweets
```

Figure 3. Creating a table for tweets.

We can check that the table was created correctly using the **"describe"** command, which gives us **metadata of a table**, as shown in Figure 4.



```
hbase(main):007:0> describe 'tweets'
DESCRIPTION                                                                                                         ENABLED
 'tweets', {NAME => 'analysis', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSIO true
 N => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE
  => 'true'}, {NAME => 'original', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRES
 SION => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCA
 CHE => 'true'}
1 row(s) in 0.0380 seconds
```

Figure 4. Details of each column family for the 'tweets' table

The output of the "describe" command is the name of the table we are describing, its status (ENABLED or DISABLED), and metadata about each column family.

Now that we have a table to work with, we can **store data** in HBase. Let's add a tweet about Egypt. We will use the "put" command, which adds data for a specific column of a row. The syntax for the command is

```
put 'table_name', 'row_id', 'col_family:column', 'value'
```

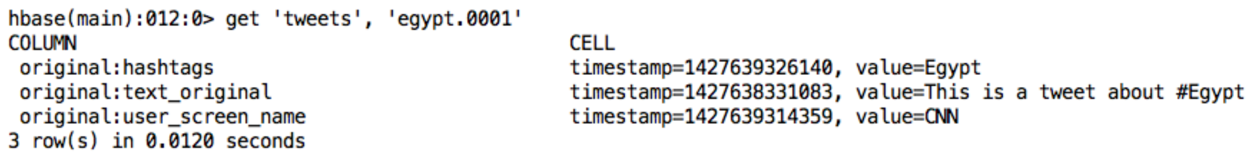
The "**put**" command adds a value for a specific column of a row in a table. If the column does not exist, a new column is create; otherwise, the column's value is updated. Similarly, if the row with ID "row_id" doesn't exist, the row gets created. Here is a concrete example:

```
put 'tweets', 'egypt.0001', 'original:text_original', 'This is a tweet about #Egypt'
```

Here, we are putting the value "This is a tweet about #Egypt" in the "text_original" column of the "original" column family of the "egypt.0001" row.

One big **limitation of the "put" command** is that it only allows us to add data for one column at a time. If we want to add data for 10 columns, we have to write 10 different "put" commands. Furthermore, the "put" command is the only way to add data in the shell. It is impractical to manually store data in HBase through the shell. Instead, one should add data programmatically using an HBase API, such as the one for Java.

As an exercise, try putting the value "CNN" in "original:user_screen_name" and "Egypt" in "original:hashtags" for the same row id. Once that is done, we can retrieve the data using the "**get**" command. "get" allows us to fetch data for a specific row id. Figure 5 shows an example of the output of the "get" command.



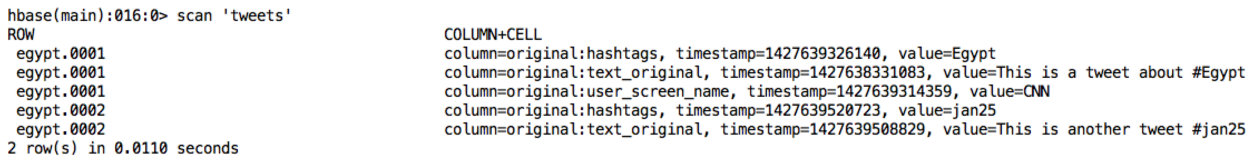
```
hbase(main):012:0> get 'tweets', 'egypt.0001'
COLUMN                                          CELL
 original:hashtags                              timestamp=1427639326140, value=Egypt
 original:text_original                         timestamp=1427638331083, value=This is a tweet about #Egypt
 original:user_screen_name                      timestamp=1427639314359, value=CNN
3 row(s) in 0.0120 seconds
```

Figure 5. Using the “get” command to get data about a tweet

Now, we will add one more row to the table:

```
put 'tweets', 'egypt.0002', 'original:text_original', 'This is another tweet #jan25'
put 'tweets', 'egypt.0002', 'original:hashtags', 'jan25'
```

Again, we can retrieve the data for this new row using “get”. If instead, we want to **get all the data in the table**, we use the “scan” command, as shown Figure 6.:



```
hbase(main):016:0> scan 'tweets'
ROW                                     COLUMN+CELL
 egypt.0001                             column=original:hashtags, timestamp=1427639326140, value=Egypt
 egypt.0001                             column=original:text_original, timestamp=1427638331083, value=This is a tweet about #Egypt
 egypt.0001                             column=original:user_screen_name, timestamp=1427639314359, value=CNN
 egypt.0002                             column=original:hashtags, timestamp=1427639520723, value=jan25
 egypt.0002                             column=original:text_original, timestamp=1427639508829, value=This is another tweet #jan25
2 row(s) in 0.0110 seconds
```

Figure 6. Retrieving all the data in an HBase table.

The “**scan**” command also supports options for scanning only a range of row IDs. Suppose that we have row IDs of the form “egypt.XXXX” and “malasya.XXXX” for two different collections. Then, using the “scan” command, we can retrieve the data for one entire collection only.

This is the end of this section. There is much more to the HBase shell, but the examples above should get you started.

6.2 Working with Avro Files

Every team processing data in HDFS will have an Avro schema. For the purposes of loading HBase and standardization, we ask teams to output their results of data processing in Avro format, according to the schemas in the Appendix.

This section shows how to read and write Avro files. Even though Avro is a convenient file format to work on Hadoop, Avro files are serialized, so they cannot be read and written like normal text files. However, Apache provides a package to interact with Avro.

For the remaining of the section, we **assume that the user is working on the Cloudera Virtual Machine**, and we will use the schema for the Noise Reduction team to illustrate how to read and write Avro. This schema is the following:

```
Noise Reduction:
{"namespace": "cs5604.tweet.NoiseReduction",
 "type": "record",
 "name": "TweetNoiseReduction",
 "fields": [
     {"name": "doc_id", "type": "string"},
     {"doc": "original", "name": "tweet_id", "type": "string"},
     {"doc": "original", "name": "text_clean", "type": "string"},
     {"doc": "original", "name": "text_original", "type": "string"},
     {"doc": "original", "name": "created_at",  "type": "string"},
     {"doc": "original", "name": "user_screen_name", "type": "string"},
     {"doc": "original", "name": "user_id", "type": "string"},
     {"doc": "original", "name": "source", "type": ["string", "null"]},
     {"doc": "original", "name": "lang", "type": ["string", "null"]},
     {"doc": "original", "name": "favorite_count", "type": ["int", "null"]},
     {"doc": "original", "name": "retweet_count", "type": ["int", "null"]},
     {"doc": "original", "name": "contributors_id", "type": ["string", "null"]},
     {"doc": "original", "name": "coordinates", "type": ["string", "null"]},
     {"doc": "original", "name": "urls", "type": ["string", "null"]},
     {"doc": "original", "name": "hashtags", "type": ["string", "null"]},
     {"doc": "original", "name": "user_mentions_id", "type": ["string", "null"]},
     {"doc": "original", "name": "in_reply_to_user_id", "type": ["string", "null"]},
     {"doc": "original", "name": "in_reply_to_status_id", "type": ["string", "null"]},

     {"doc": "original", "name": "text_clean2", "type": ["null", "string"], "default": null},
     {"doc": "original", "name": "collection", "type": ["null", "string"], "default": null}

 ]
}
```

Writing Avro Files (Java)

The schema above was compiled into a Java class using a standard Avro tool. We will use this Java class to write Avro. The examples in this section write Avro files sequentially. We are currently working on the MapReduce version, but the code below can be used for prototyping.

```
package cs5604.hadoop;

import java.io.File;
import java.io.IOException;
import org.apache.avro.file.DataFileWriter;
import org.apache.avro.io.DatumWriter;
import org.apache.avro.specific.SpecificDatumWriter;
// The import below is the class compiled from the schema. These classes will be distributed to each team
import cs5604.tweet.NoiseReduction.TweetNoiseReduction;

public class CreateDummyTweets {

  /**
   * @param args
   * @throws IOException
   */
  public static void main(String[] args) throws IOException {
    // object representing a tweet from the noise reduction team
    TweetNoiseReduction tweet = new TweetNoiseReduction();

    DatumWriter<TweetNoiseReduction> tweetDatumWriter = new
SpecificDatumWriter<TweetNoiseReduction>(TweetNoiseReduction.class);
    DataFileWriter<TweetNoiseReduction> dataFileWriter = new
DataFileWriter<TweetNoiseReduction>(tweetDatumWriter);
    dataFileWriter.create(tweet.getSchema(), new File("tweets.avro")); // data will be saved to this file

    // notice that there are getter methods corresponding to the fields in the schema
    // write first tweet
    tweet.setDocId("egypt.1");
    tweet.setTweetId("123k25lse");
    tweet.setUserId("123");
    tweet.setUserScreenName("CNN");
    tweet.setCreatedAt("2014-12-01 13:00:12");
    tweet.setTextOriginal("This is the original tweet #freedom #egypt http://twi.tter.com");
    tweet.setTextClean("This is the original tweet");
    tweet.setHashtags("egypt|freedom");
    tweet.setUrls("http://twi.tter.com");
    tweet.setLang("English");
    dataFileWriter.append(tweet);
    // write second tweet
    tweet = new TweetNoiseReduction();
    tweet.setDocId("egypt.2");
    tweet.setTweetId("123k25lse");
    tweet.setUserId("235");
    tweet.setUserScreenName("MSNBC");
    tweet.setCreatedAt("2014-12-01 13:00:12");
    tweet.setTextOriginal("This is another tweet");
    tweet.setTextClean("This is another tweet");
    tweet.setLang("English");
    dataFileWriter.append(tweet);
    // write third tweet
    tweet = new TweetNoiseReduction();
    tweet.setDocId("egypt.3");
    tweet.setTweetId("12413edsf2");
    tweet.setUserId("421");
    tweet.setUserScreenName("paul");
    tweet.setCreatedAt("2014-12-01 13:00:12");
    tweet.setTextOriginal("RT @CNN: \"This is the original tweet #freedom #egypt http://twi.tter.com\"");
    tweet.setTextClean("This is the original tweet");
    tweet.setHashtags("egypt|freedom");
    tweet.setUrls("http://twi.tter.com");
    tweet.setRetweetCount(1);
    tweet.setInReplyToUserId("123k25lse");
    tweet.setLang("English");
    dataFileWriter.append(tweet);

    dataFileWriter.close();
  }
}
```

Let’s walk through this code example. First, we import the Avro libraries (from /usr/lib/avro) and the TweetNoiseReduction class corresponding to the Avro schema:

```
import java.io.File;
import java.io.IOException;
import org.apache.avro.file.DataFileWriter;
import org.apache.avro.io.DatumWriter;
import org.apache.avro.specific.SpecificDatumWriter;
// The import below is the class compiled from the schema. These classes will be distributed to each team
import cs5604.tweet.NoiseReduction.TweetNoiseReduction;
```

Then, we instantiate a TweetNoiseReduction object that we will use to write the data to the Avro file. In order to write data, we instantiate a DatumWriter and a DataFileWriter for the TweetNoiseReduction class. We also specify the name of the output file ("tweets.avro"):

```
 // object representing a tweet from the noise reduction team
        TweetNoiseReduction tweet = new TweetNoiseReduction();

        DatumWriter<TweetNoiseReduction> tweetDatumWriter = new
SpecificDatumWriter<TweetNoiseReduction>(TweetNoiseReduction.class);
        DataFileWriter<TweetNoiseReduction> dataFileWriter = new
DataFileWriter<TweetNoiseReduction>(tweetDatumWriter);
        dataFileWriter.create(tweet.getSchema(), new File("tweets.avro")); // data will be saved to this file
```

Now, we fill in the data for the first tweet:

```
// write first tweet
        tweet.setDocId("egypt.1");
        tweet.setTweetId("123k25lse");
        tweet.setUserId("123");
        tweet.setUserScreenName("CNN");
        tweet.setCreatedAt("2014-12-01 13:00:12");
        tweet.setTextOriginal("This is the original tweet #freedom #egypt http://twi.tter.com");
        tweet.setTextClean("This is the original tweet");
        tweet.setHashtags("egypt|freedom");
        tweet.setUrls("http://twi.tter.com");
        tweet.setLang("English");
```

We notice that not all fields are mandatory. For the Noise Reduction team, in particular, the only data that they need to provide are the fields that do not have a "null" indicator in the schema.

After filling in the data, we write the tweet to file:

```
dataFileWriter.append(tweet);
```

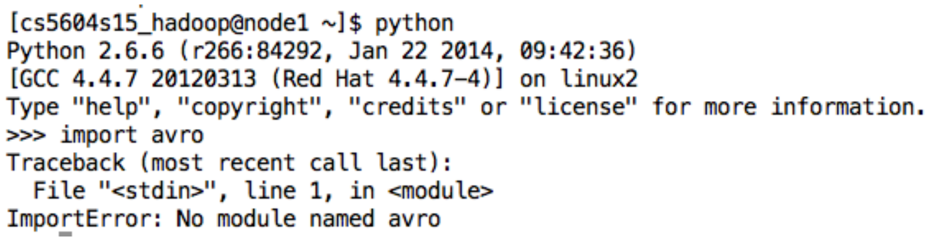
We repeat for the other two tweets. At the end, we have to close the file to save our changes:

```
dataFileWriter.close();
```

Writing Avro Files (Python)

We can also read and write Avro using Python. In this case, however, we will not use pre-compiled classes, so it is the responsibility of the developer to keep the data consistent with the schema. This subsection assumes that the user is comfortable with python and can write non-trivial data-processing scripts.

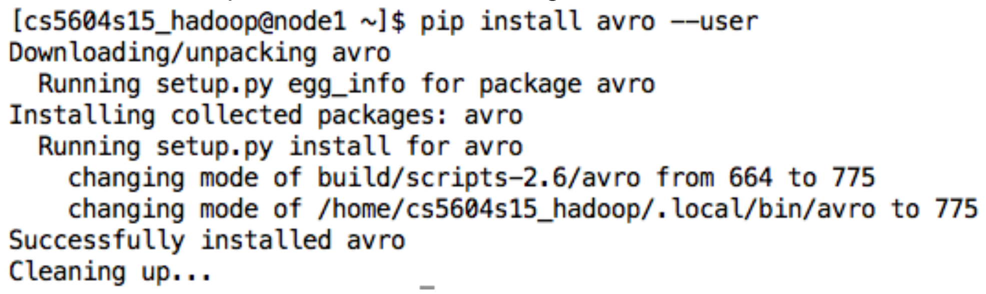
First, open the Python shell and check that you have the Avro library installed (Figure 7).



```
[cs5604s15_hadoop@node1 ~]$ python
Python 2.6.6 (r266:84292, Jan 22 2014, 09:42:36)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import avro
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named avro
```

Figure 7. Trying to import Avro in Python. We see an error because the Avro libraries are not installed yet.

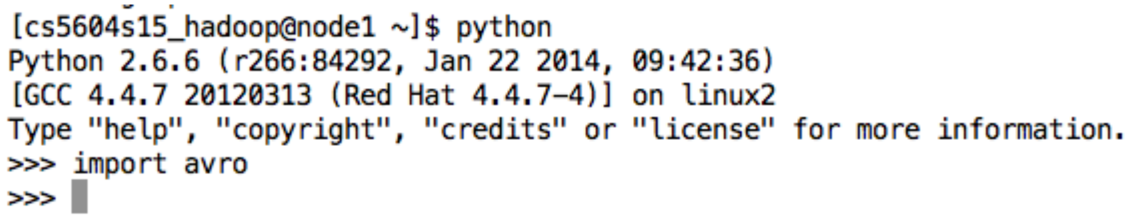
The error in Figure 7 indicates that I don't have Avro in my system. We can install the library using the pip utility. Since, I don't have root access in the Hadoop cluster, I have to install Avro for my local user, and the procedure is shown in Figure 8:



```
[cs5604s15_hadoop@node1 ~]$ pip install avro --user
Downloading/unpacking avro
  Running setup.py egg_info for package avro
Installing collected packages: avro
  Running setup.py install for avro
    changing mode of build/scripts-2.6/avro from 664 to 775
    changing mode of /home/cs5604s15_hadoop/.local/bin/avro to 775
Successfully installed avro
Cleaning up...
```

Figure 8. Installing the Avro libraries for Python.

Let's try again:



```
[cs5604s15_hadoop@node1 ~]$ python
Python 2.6.6 (r266:84292, Jan 22 2014, 09:42:36)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import avro
>>> 
```

Figure 9. Successfully importing the Avro library.

As shown in Figure 9, there is no problem this time.

Now, we can write a script to convert a TSV file to Avro. We will use the script below to convert our small tweet collection to an Avro file.

```
import avro.schema
from avro.datafile import DataFileWriter
from avro.io import DatumWriter

def main():
  # load schema from file
  schema = avro.schema.parse(open("noise-reduction.avsc").read())
  # instantiate writer
  writer = DataFileWriter(open("z4t.avro", "w"), DatumWriter(), schema)

  doc_id = 0  # the id of the tweet
  with open("data/z4t.csv") as f:
    # skip header
    f.readline()
    # iterate the TSV file
    for line in f:
        line = line.decode('utf-8').strip()
        doc_id += 1
        tokens = line.split("\t")

        text_original = tokens[0]
        user_screen_name = tokens[1]
        tweet_id = tokens[2]
        created_at = tokens[3]

        # hashtags are words that start with a “#”
        hashtags = [word for word in text_original.split() if word.startswith("#")]
        # URLs are words that start with “http”
        urls = [word for word in text_original.split() if word.startswith("http")]
        # make a json object for the tweet. The keys have to map to fields in the Avro schema
        json_tweet = {"doc_id": "egypt" + str(doc_id), "text_original": text_original,
                    "created_at": created_at, "tweet_id": tweet_id, "user_screen_name": user_screen_name,
                    "hashtags": "|".join(hashtags), "urls": "|".join(urls)}
        # write to file
        writer.append(json_tweet)

    writer.close()
    print "%s records written to avro" % doc_id


if __name__ =="__main__":
    main()
```

Let’s walk through the code. First, we load the Noise Reduction schema and instantiate a DataFileWriter to save the Avro data in “z4t.avro”.

```
# load schema from file
    schema = avro.schema.parse(open(“noise-reduction.avsc").read())
    # instantiate writer
    writer = DataFileWriter(open("z4t.avro", "w"), DatumWriter(), schema)
```

Then, we open our small collection fle, “z4t.csv”. In this tab-separated file, the first row is the text of the tweet, the second row has the user screen name, the third row has the tweet id, and the fourth row has the creation time of the tweet.

We iterate through each line splitting the text by tab and extracting the corresponding data:

```
    for line in f:
        line = line.decode('utf-8').strip()
        doc_id += 1
        tokens = line.split("\t")

        text_original = tokens[0]
        user_screen_name = tokens[1]
        tweet_id = tokens[2]
        created_at = tokens[3]
```

We can extract hashtags and URLs, by checking which words start with “#” and “http”, respectively (Note: this is not a perfect filter):

```
  # hashtags are words that start with a “#”
        hashtags = [word for word in text_original.split() if word.startswith("#")]
        # URLs are words that start with “http”
        urls = [word for word in text_original.split() if word.startswith("http")]
```

Once we have all the data, we put it in a JSON object, and we write to the file. **Notice that we store lists as “|”-separated strings.**

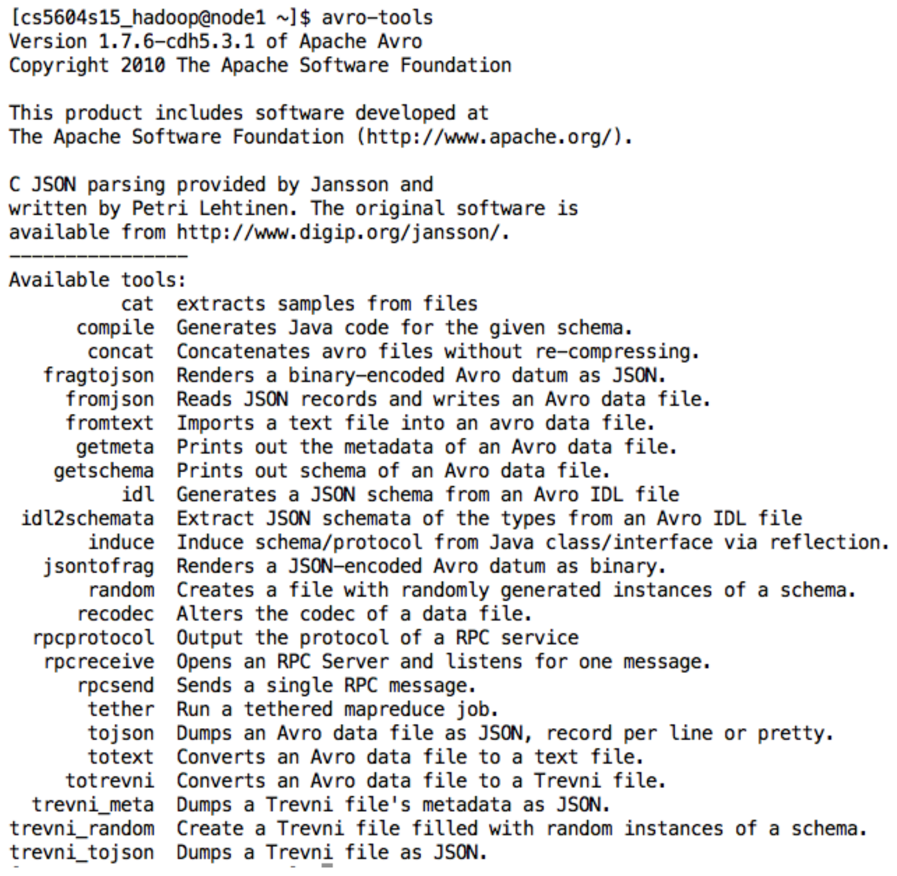
```
 json_tweet = {"doc_id": "egypt" + str(doc_id), "text_original": text_original,
                      "created_at": created_at, "tweet_id": tweet_id, "user_screen_name": user_screen_name,
                      "hashtags": "|".join(hashtags), "urls": "|".join(urls)}
        # write to file
        writer.append(json_tweet)
```

Finally, we close the output file once we are done iterating the input file:

```
writer.close()
```

Reading Avro Files

Reading Avro files programmatically in Java and Python is similar to writing files. We plan to include detailed code examples in a future edition, but, for now, we will show how to convert Avro data to Json using the “avro-tools” utility. Json is human-readable and easy to process by most programming languages.
By typing “avro-tools” in the terminal, we get the output shown in Figure 10:



```
[cs5604s15_hadoop@node1 ~]$ avro-tools
Version 1.7.6-cdh5.3.1 of Apache Avro
Copyright 2010 The Apache Software Foundation

This product includes software developed at
The Apache Software Foundation (http://www.apache.org/).

C JSON parsing provided by Jansson and
written by Petri Lehtinen. The original software is
available from http://www.digip.org/jansson/.
----------------
Available tools:
          cat  extracts samples from files
      compile  Generates Java code for the given schema.
       concat  Concatenates avro files without re-compressing.
   fragtojson  Renders a binary-encoded Avro datum as JSON.
     fromjson  Reads JSON records and writes an Avro data file.
     fromtext  Imports a text file into an avro data file.
      getmeta  Prints out the metadata of an Avro data file.
    getschema  Prints out schema of an Avro data file.
          idl  Generates a JSON schema from an Avro IDL file
 idl2schemata  Extract JSON schemata of the types from an Avro IDL file
       induce  Induce schema/protocol from Java class/interface via reflection.
   jsontofrag  Renders a JSON-encoded Avro datum as binary.
       random  Creates a file with randomly generated instances of a schema.
      recodec  Alters the codec of a data file.
  rpcprotocol  Output the protocol of a RPC service
   rpcreceive  Opens an RPC Server and listens for one message.
      rpcsend  Sends a single RPC message.
       tether  Run a tethered mapreduce job.
       tojson  Dumps an Avro data file as JSON, record per line or pretty.
       totext  Converts an Avro data file to a text file.
     totrevni  Converts an Avro data file to a Trevni file.
  trevni_meta  Dumps a Trevni file's metadata as JSON.
trevni_random  Create a Trevni file filled with random instances of a schema.
trevni_tojson  Dumps a Trevni file as JSON.
```

Figure 10. The options available in *avro-tools*.

For now, the options that we are interested in are “tojson” and “fromjson” , which allows us to write Avro files to json format and vice versa.

As an example, we will convert the Avro file that we generated above in Java to JSON. We run the following command

```
avro-tools tojson tweets.avro > tweets.json
```

This command creates a file containing three tweets:

{"doc_id":"egypt.1","tweet_id":"123k25lse","text_clean":"This is the original tweet","text_original":"This is the original tweet #freedom #egypt http://twi.tter.com","created_at":"2014-12-01 13:00:12","user_screen_name":"CNN","user_id":"123","source":null,"lang":{"string":"English"},"favorite_count":null,"retweet_count":null,"contributors_id":null,"coordinates":null,"urls":{"string":"http://twi.tter.com"},"hashtags":{"string":"egypt|freedom"},"user_mentions_id":null,"in_reply_to_user_id":null,"in_reply_to_status_id":null}
{"doc_id":"egypt.2","tweet_id":"123k25lse","text_clean":"This is another tweet","text_original":"This is another tweet","created_at":"2014-12-01 13:00:12","user_screen_name":"MSNBC","user_id":"235","source":null,"lang":{"string":"English"},"favorite_count":null,"retweet_count":null,"contributors_id":null,"coordinates":null,"urls":{"string":"http://twi.tter.com"},"hashtags":{"string":"egypt|freedom"},"user_mentions_id":null,"in_reply_to_user_id":null,"in_reply_to_status_id":null}

{"doc_id":"egypt.3","tweet_id":"12413edsf2","text_clean":"This is the original tweet","text_original":"RT @CNN: \"This is the original tweet #freedom #egypt http://twi.tter.com\"","created_at":"2014-12-01 13:00:12","user_screen_name":"paul","user_id":"421","source":null,"lang":{"string":"English"},"favorite_count":null,"retweet_count":{"int":1},"contributors_id":null,"coordinates":null,"urls":{"string":"http://twi.tter.com"},"hashtags":{"string":"egypt|freedom"},"user_mentions_id":null,"in_reply_to_user_id":{"string":"123k25lse"},"in_reply_to_status_id":null}

6.3 Loading Avro Data into HBase

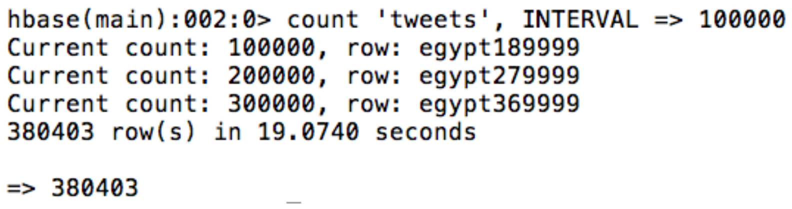
We have written a command-line program to load Avro data into HBase. Teams can use this utility to easily import their data. The current version loads the data sequentially; we could load our small collection into HBase in about one minute using this program. However, we don't expect it to scale to the big collections, so we are working on a MapReduce implementation.

The program takes two arguments: 1) an Avro file generated from any of the schemas provided to the teams and 2) either the word "tweets" or "webpages" indicating what kind of data to write.

As an example, we will load the avro file created in the "Writing Avro Files (Python)" section. The example assumes HBase is currently running and that there is a table called 'tweets' with column families "original" and "analysis" (see Section 6.2). Run the command:

```
java -jar hbase-loader.jar z4t.avro tweets
```

After about one minute, the program should report the number of rows inserted to HBase. You can see that the data has been loaded using the HBase shell (Figure 11).



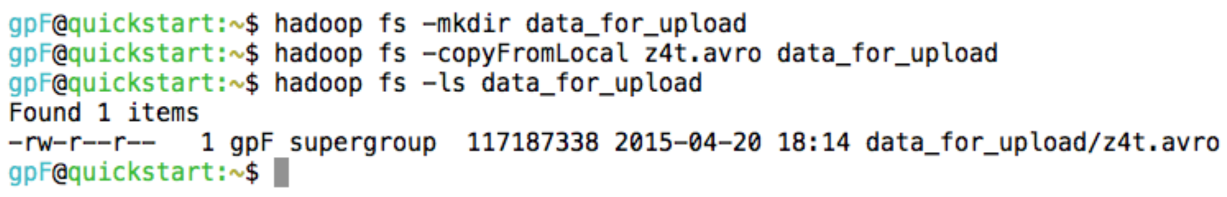
```
hbase(main):002:0> count 'tweets', INTERVAL => 100000
Current count: 100000, row: egypt189999
Current count: 200000, row: egypt279999
Current count: 300000, row: egypt369999
380403 row(s) in 19.0740 seconds

=> 380403
```

Figure 11. 'tweets' table after importing a sample collection.

We have also developed a MapReduce version of this program. The MapReduce JAR takes three arguments: 1) an Avro file generated from any of the schemas provided to the teams, 2) an output directory name in HDFS, and 3) the name of an HBase table with column families "original" and "analysis" where the data from 1) will be stored. For example, we will load the z4t.avro file that we created above. First, we store the file in HDFS:

```
hadoop fs -mkdir data_for_upload
hadoop fs -copyFromLocal z4t.avro data_for_upload
hadoop fs -ls data_for_upload
```

After running these commands, we should see the file stored in HDFS, as shown in Figure 12:



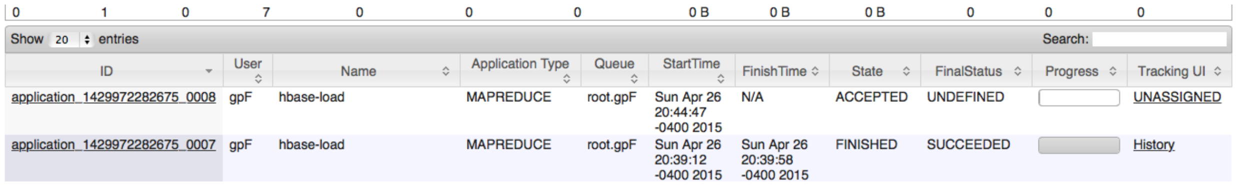
```
gpF@quickstart:~$ hadoop fs -mkdir data_for_upload
gpF@quickstart:~$ hadoop fs -copyFromLocal z4t.avro data_for_upload
gpF@quickstart:~$ hadoop fs -ls data_for_upload
Found 1 items
-rw-r--r--   1 gpF supergroup  117187338 2015-04-20 18:14 data_for_upload/z4t.avro
gpF@quickstart:~$ 
```

Figure 12. Loading a file from the main Hadoop node into HDFS.

Then, we start the MapReduce job as follows:

```
hadoop jar mr-hbase-loader.jar data_for_upload output tweets
```

We can see the progress of the MapReduce job either in Hue or in the Resource Manager node of the cluster. For the cluster used in the class, the Resource Manager is at http://128.173.49.66:8088/cluster . Figure 13 is a screenshot of the Resource Manager view:



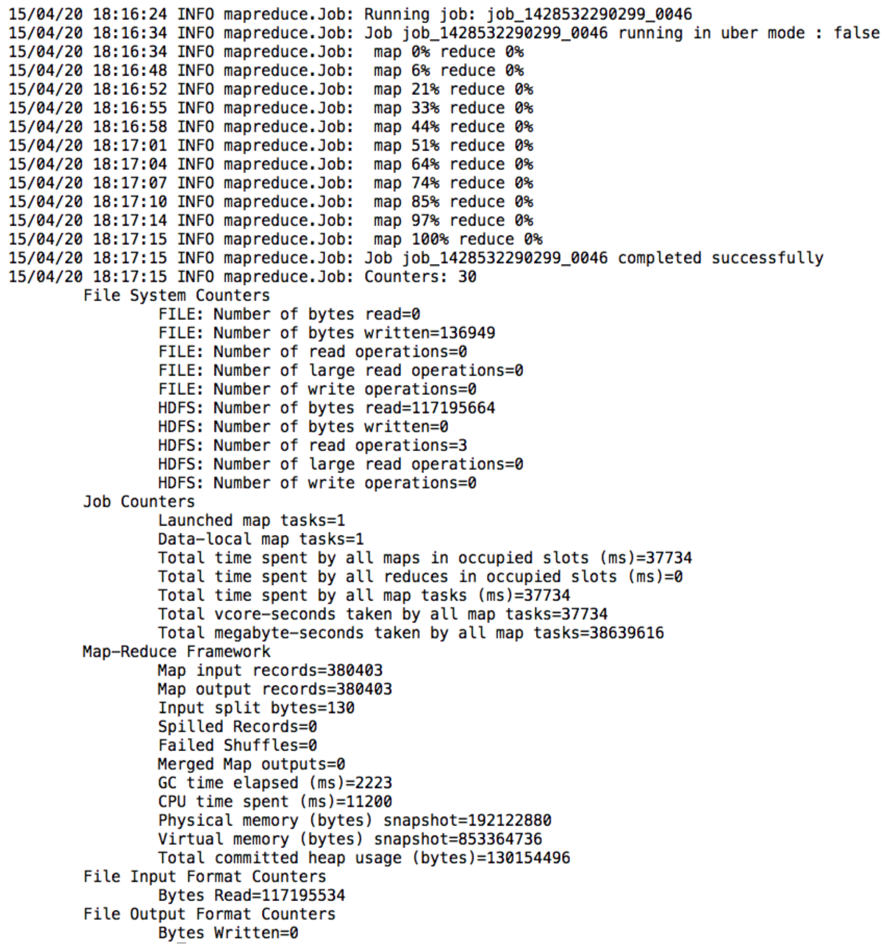
0	1	0	7	0	0	0	0 B	0 B	0 B	0	0	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1429972282675_0008	gpF	hbase-load	MAPREDUCE	root.gpF	Sun Apr 26 20:44:47 -0400 2015	N/A	ACCEPTED	UNDEFINED		UNASSIGNED
application_1429972282675_0007	gpF	hbase-load	MAPREDUCE	root.gpF	Sun Apr 26 20:39:12 -0400 2015	Sun Apr 26 20:39:58 -0400 2015	FINISHED	SUCCEEDED		History

Figure 13. Our MapReduce job is being processed.

This shows that our job (first row) is queued to start. As the job progresses, we can see the status in the terminal, as shown in Figure 14:



```
15/04/20 18:16:24 INFO mapreduce.Job: Running job: job_1428532290299_0046
15/04/20 18:16:34 INFO mapreduce.Job: Job job_1428532290299_0046 running in uber mode : false
15/04/20 18:16:34 INFO mapreduce.Job:  map 0% reduce 0%
15/04/20 18:16:48 INFO mapreduce.Job:  map 6% reduce 0%
15/04/20 18:16:52 INFO mapreduce.Job:  map 21% reduce 0%
15/04/20 18:16:55 INFO mapreduce.Job:  map 33% reduce 0%
15/04/20 18:16:58 INFO mapreduce.Job:  map 44% reduce 0%
15/04/20 18:17:01 INFO mapreduce.Job:  map 51% reduce 0%
15/04/20 18:17:04 INFO mapreduce.Job:  map 64% reduce 0%
15/04/20 18:17:07 INFO mapreduce.Job:  map 74% reduce 0%
15/04/20 18:17:10 INFO mapreduce.Job:  map 85% reduce 0%
15/04/20 18:17:14 INFO mapreduce.Job:  map 97% reduce 0%
15/04/20 18:17:15 INFO mapreduce.Job:  map 100% reduce 0%
15/04/20 18:17:15 INFO mapreduce.Job: Job job_1428532290299_0046 completed successfully
15/04/20 18:17:15 INFO mapreduce.Job: Counters: 30
        File System Counters
                FILE: Number of bytes read=0
                FILE: Number of bytes written=136949
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=117195664
                HDFS: Number of bytes written=0
                HDFS: Number of read operations=3
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=0
        Job Counters
                Launched map tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=37734
                Total time spent by all reduces in occupied slots (ms)=0
                Total time spent by all map tasks (ms)=37734
                Total vcore-seconds taken by all map tasks=37734
                Total megabyte-seconds taken by all map tasks=38639616
        Map-Reduce Framework
                Map input records=380403
                Map output records=380403
                Input split bytes=130
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=2223
                CPU time spent (ms)=11200
                Physical memory (bytes) snapshot=192122880
                Virtual memory (bytes) snapshot=853364736
                Total committed heap usage (bytes)=130154496
        File Input Format Counters
                Bytes Read=117195534
        File Output Format Counters
                Bytes Written=0
```

Figure 14. Progress of a MapReduce job.

As a technical note, this job does not have a Reduce task. All the data is uploaded in the Map task.

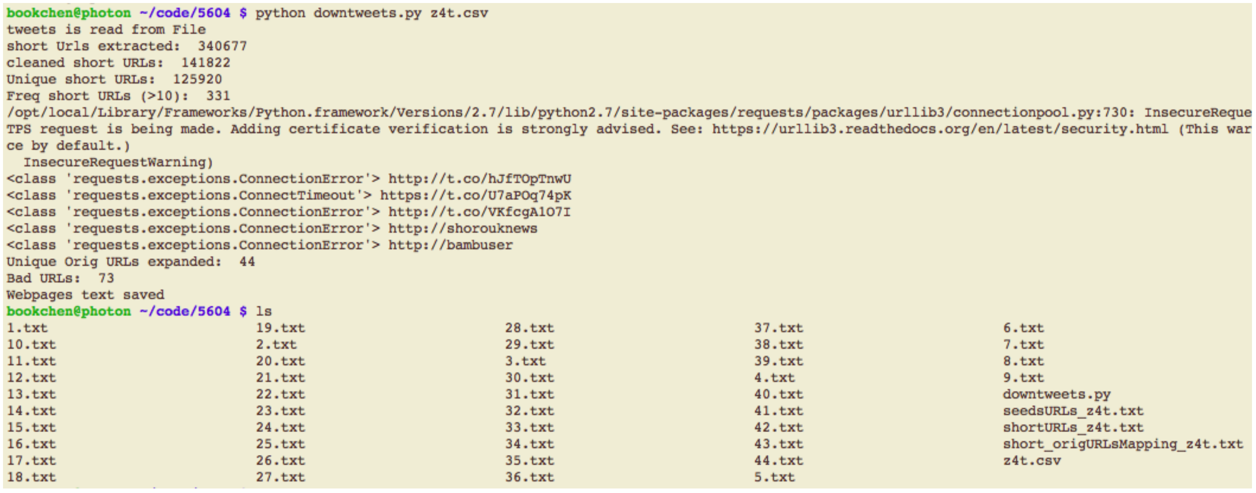
6.3 Crawling web pages

In this project, we initially only have tweet data. We obtain web page data by crawling the URLs mentioned in tweets. Generally, to crawl web pages, one can simply use a URL library, such as, urllib in Python. However, in this project, we have terabytes of tweets, so we want to crawl data in parallel and store the crawled data in a distributed manner. After obtaining the web page dataset, we still have to parse and process the data. It is hard to achieve these goals with a simple Python script. Luckily, there is an open source program called Apache Nutch [14] that provides all the features we are seeking.

Below, we will briefly introduce how to crawl web pages using a Python script. Then, we will introduce Apache Nutch, including its architecture and how to install, and a quick start guide. Finally, we will demonstrate using Nutch to crawl the web pages mentioned in billions of tweets.

6.5.1 Python Script Approach

There is a Python script provided by the TAs of the class. We ran the script on our small tweet collection. Our tweets collection is about the Egyptian revolution, and it has 380,404 tweets. Extracting the URLs from these tweets and downloading the corresponding web pages takes about 1 minute on a 2012 Mac Air. The screenshot in Figure 15 shows the output of the script and the list of text files generated.



```
bookchen@photon ~/code/5604 $ python downtweets.py z4t.csv
tweets is read from File
short Urls extracted:  340677
cleaned short URLs:  141822
Unique short URLs:  125920
Freq short URLs (>10):  331
/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/requests/packages/urllib3/connectionpool.py:730: InsecureReque
TPS request is being made. Adding certificate verification is strongly advised. See: https://urllib3.readthedocs.org/en/latest/security.html (This war
ce by default.)
  InsecureRequestWarning)
<class 'requests.exceptions.ConnectionError'> http://t.co/hJfTOpTnwU
<class 'requests.exceptions.ConnectTimeout'> https://t.co/U7aPOq74pK
<class 'requests.exceptions.ConnectionError'> http://t.co/VKfcgA1O7I
<class 'requests.exceptions.ConnectionError'> http://shorouknews
<class 'requests.exceptions.ConnectionError'> http://bambuser
Unique Orig URLs expanded:  44
Bad URLs:  73
Webpages text saved
bookchen@photon ~/code/5604 $ ls
1.txt                         19.txt                        28.txt                        37.txt                        6.txt
10.txt                        2.txt                         29.txt                        38.txt                        7.txt
11.txt                        20.txt                        3.txt                         39.txt                        8.txt
12.txt                        21.txt                        30.txt                        4.txt                         9.txt
13.txt                        22.txt                        31.txt                        40.txt                        downtweets.py
14.txt                        23.txt                        32.txt                        41.txt                        seedsURLs_z4t.txt
15.txt                        24.txt                        33.txt                        42.txt                        shortURLs_z4t.txt
16.txt                        25.txt                        34.txt                        43.txt                        short_origURLsMapping_z4t.txt
17.txt                        26.txt                        35.txt                        44.txt                        z4t.csv
18.txt                        27.txt                        36.txt                        5.txt
```

Figure 15. Crawling web page data using Python.

Although the running time of the script is very short for our collection, it was reported by some teams that the script would take about half an hour to run for some collections. This is a huge performance problem if we are going to process several TBs of tweets.

6.5.2 Apache Nutch Approach

There are many web crawlers out there. The reason we choose Nutch is that it fits the purpose of large-scale crawling (ultimately, we want to fetch terabytes of web pages). Furthermore, Nutch nicely integrates with our Hadoop cluster.

Nutch takes a plain text file as input. The file is just a list of URLs serving as the starting point of the crawling loop. Crawling is an iterative process:

$$\text{URL} \rightarrow \text{web page} \rightarrow \text{new URL} \rightarrow \text{web page} \rightarrow \ldots$$

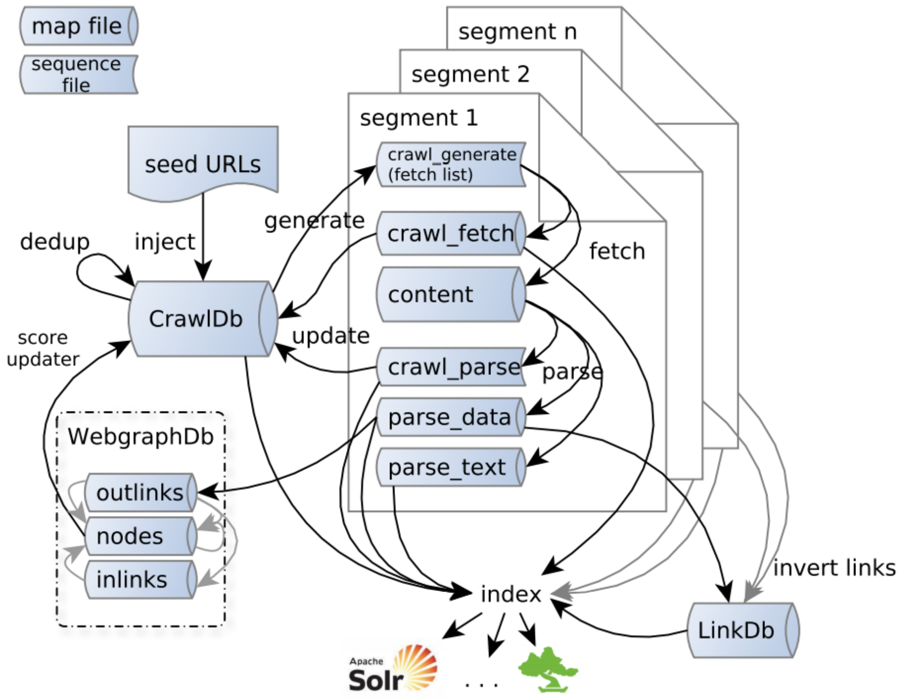


Figure 16. Nutch Architecture

Now, we want to document our experience of installing Nutch 1.9 and crawling web page a Hadoop cluster.

Quick Start Guide

We can use the crawl script at “bin/crawl” to start crawling.

- save all the urls to a plain text file seed.txt user a folder (eg. urls)
- load the that folder into HDFS: `hdfs dfs -copyFromLocal urls urls`
- use the crawl script to start crawling
  `bin/crawl urls SitesFromTweets <solr_url> 1`
  - where `urls` is the path to the directory containing URL list
  - `SitesFromTweets` is the directory to store the crawled web pages, will created automatically if not exist.
  - `<solr_url>` is the Solr instance to index the crawled web pages. Because in this project, we don’t want the web pages to be indexed directly by Solr, we disable these function by commenting out the related code in bin/crawl script. Therefore, one can provide any string at this place.
  - number 1 is the number of round Nutch will perform crawling. Because we only crawl the web pages in the URL list, this number should be set to 1.

Extracting URLs From a Big Tweet Collection

All the big collections are currently stored as Avro files, so the Python script used to extract URLs from CSV files needs to be modified to work with Avro. Python can directly read Avro files, but for simplicity, we use *avro-tool* to convert an Avro file to JSON; then, Python will read tweets from a JSON file.

To expand the shorten URLs, we use the *urllib* of Python as follows:

```
resp = urllib.urlopen(shourt_url)
long_url = resp.url
```

Expanding short URLs turns out to be the bottleneck of the entire web page crawling process because the operation of expanding URLs requires us to visit the web pages through an Internet connection, and we have little control over how long the communication takes. What we can do is to run the Python script in parallel to fire multiple connections, but, in a cluster setting, we access the internet through only one IP address. Multiple connections may be rejected by the URL expanding server.

Given the above, our current script uses only one connection. The script expands one URL in slightly less than one second. In other words, our script expands 36,000 URLs in 9.5 hours in our testing on the head node of the cluster hadoop.dlib.vt.edu. In our big collection, there are 11,747,983 tweets mentioning 9,093,437 shortened URLs; 4,510,250 of them are unique. To expand all these 4,510,250 URLs using the current serial implementation would take 49 days. Therefore, we can only expanded some of them, the most frequently mentioned URLs.

Below, are the instructions to run the script:

- Run `hdfs dfs -copyToLocal /class/CS5604S15/dataset/#egypt_B_AVRO ~/dataset`
- Run `avro-tools to json ~/dataset/#egypt_B_AVRO/part-m-0000.avro > egypt_B.json`
- Run `python extractURLs.py egypt_B.json 10`
    - where `10` means extract only the URLs that appear at least 10 times

Here is the extractURLs.py script:

```
import sys
import re
import urllib
from collections import defaultdict

# run this script like:
# python extractURLs.py egypt_tweets.json 3
# where egypt_tweets.json can be extracted from egypt_tweets.avro by "avro-tool tojson"
# where 3 means only extract those URLs that appear at least 3 times
# You can try this script on a very large json file
# Use Ctrl + c to stop this script, the already expanded URLs will still be saved.
```

```
tweetFile = sys.argv[1]
archiveID = tweetFile.split(".")[0]
min_repeat = int(sys.argv[2])

#### load tweets from file
f = open(tweetFile,"r")
f.close()
tweets = []
with open(tweetFile,"r") as f:
    for line in f.readlines():
        t = re.findall(r'"text":\{"string":"(.+)"\},"to_user_id"',line)[0]
        tweets.append(t)

n_tweets = len(tweets)
print "# of tweets read from %s: %d" % (tweetFile, len(tweets))

#### Extract URLs from Tweets
urls_dct = defaultdict(int)
n_urls = 0
for tweet in tweets:
    regExp = "(?P<url>https?://[a-zA-Z0-9\./-]+)"
    url_li = re.findall(regExp, tweet)
    while (len(url_li) > 0):
        url = url_li.pop()
        n_urls += 1
        urls_dct[url] += 1

print "# of short URLs in tweets:", n_urls
print "# of Unique short URLs in tweets:", len(urls_dct.keys())

#### filter out the more frequent URLs
uniq_urls_high_freq = []
for url in urls_dct:
    if urls_dct[url] >= min_repeat:
        uniq_urls_high_freq.append(url)
print '# of unique URLs repeat at least %d times: %d' % (min_repeat,
len(uniq_urls_high_freq))

#### expand shorten URLs and save to file
urls_fname = "%s_urls.txt" % archiveID
with open(urls_fname,"w") as f_urls:
    expanded_urls_lst = []
    n_uniq_urls_high_freq = len(uniq_urls_high_freq)
    i_url = 0
    for url in uniq_urls_high_freq:
        print "%d/%d" % (i_url,n_uniq_urls_high_freq),
        i_url += 1
        print "expanding",url,
        if 'http://t.co/' or 'https://t.co/' in url:
            try:
                resp = urllib.urlopen(url)
            except KeyboardInterrupt:
                print ""
                print "-------------"
                print "%d short URLs in %d tweets:" % (n_urls, n_tweets)
                print "%d unique short URLs" % (len(urls_dct.keys()))
                print '%d unique URLs repeat at least %d times' %
(len(uniq_urls_high_freq),min_repeat)
                print "URLs save to:",urls_fname
                exit()
            except:
                print " FAILED!"
                continue
            print ""
            if url == resp.url:
                continue
            url = resp.url
        expanded_urls_lst.append(url)
        f_urls.write("%s\n" % url)

print "------------"
print "%d short URLs in %d tweets:" % (n_urls, n_tweets)
print "%d unique short URLs" % (len(urls_dct.keys()))
print '%d unique URLs repeat at least %d times' % (len(uniq_urls_high_freq),min_repeat)
print "URLs save to:",urls_fname
```

7. Developer's Manual

7.1 Technical Specifications

7.1.1 Hadoop Cluster Specifications

In this course, we use two different Hadoop systems for development and production.

Development is done in a virtual machine provided by Cloudera. This virtual machine runs a Red Hat operating system and simulates a single-node Hadoop cluster installation. The virtual machine comes installed with all the tools that we need in the production cluster, except for Nutch, for which we describe the installation in Section 7.2.
The **production** cluster has the CDH 5.3.1 Cloudera version of Hadoop installed. The cluster has the following specifications:

- Number of nodes
    - 19 Hadoop nodes
    - 1 Manager node
    - 2 Tweet DB nodes
    - 1 HDFS backup node

- CPU
    - Intel i5 Haswell Quad core 3.3 Ghz Xeon
- RAM
    - 660 GB in total
    - 32 GB in each of the 19 Hadoop nodes
    - 4 GB in the manager node
    - 16 GB in the tweet DB nodes
    - 16 GB in the HDFS backup node
- Storage
    - 60 TB across Hadoop, manager, and tweet DB nodes
    - 11.3 TB for backup

7.1.2 Architecture and Data Workflow

In coordination with the other teams, we designed the data workflow depicted in Figure 17. The main stages of this workflow are the following:

1. **HDFS Data Loading**
    a. Twitter data is loaded from a relational database into the Hadoop cluster using Sqoop, a tool for bulk loading structured data into HDFS.
    b. Web page data is fetched using Nutch, a web-crawler, and stored into HDFS as plain text, HTML, and WARC files.
2. **Noise Reduction**
    a. Tweets and web pages are processed by the Noise Reduction team. The *noise-reduced* data is stored in HDFS as plain text, HTML, and in Avro format.
3. **Data Analysis**
    a. The data analysis teams take the noise-reduced data as input for their respective tasks. Every team will produce (in addition to their other outputs), a file in Avro format to be loaded into HDFS.
    b. As necessary, every team will also produce data to be shared with other teams. For example, the Clustering team can produce a file to be used by the Social Networks team.
4. **HBase Data Loading**
    a. The data produced at the end of Steps 2 and 3 will be loaded into HBase by the creator of the data using a MapReduce program written by our team. The noise-reduced data will be loaded first, and the rest of the teams will subsequently load data in the form of updates or additions to the existing noise-reduced data in Solr.
5. **Indexing into Solr**
    a. Data in HDFS will be indexed into Solr using the *hbase-indexer* tool.

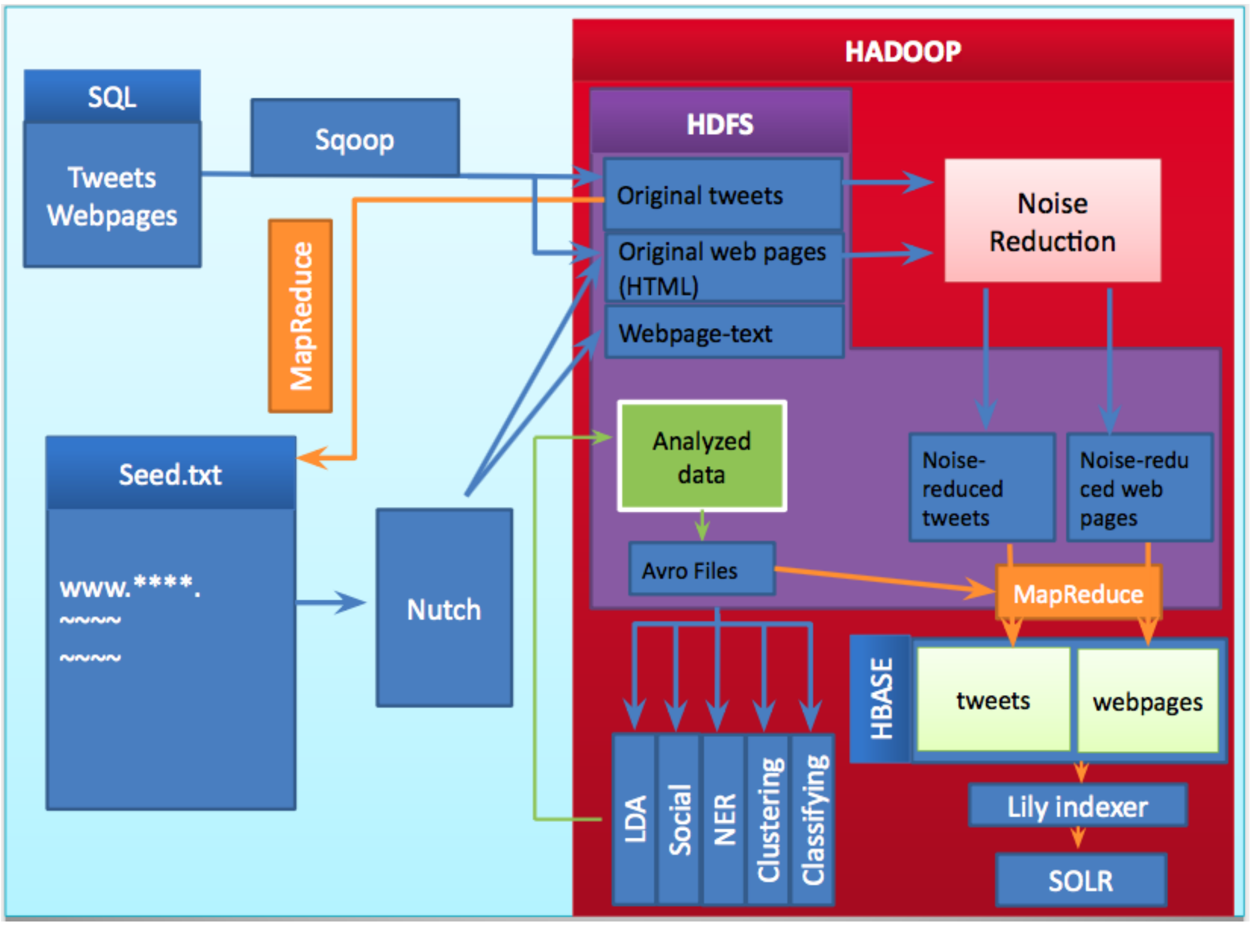

HADOOP
SQL
Tweets
Webpages
Sqoop
MapReduce
HDFS
Original tweets
Original web pages (HTML)
Webpage-text
Noise Reduction
Seed.txt
www.****.
~~~~
~~~~
Nutch
Analyzed data
Avro Files
Noise-reduced tweets
Noise-reduced web pages
MapReduce
HBASE
tweets
webpages
Lily indexer
SOLR
LDA
Social
NER
Clustering
Classifying

Figure 17. Data workflow for our course project.

7.2 Data

7.2.1 Data Description

Tweets

The initial tweet data loaded into the cluster is in three formats: Avro, CSV, and Sequence Files. However, in this class, we mostly used the Avro version, so we just show an example of this format (after converting to JSON) below.

```
{
   "archivesource":{
      "string":"twitter-search"
   },
   "text":{
      "string":"RT @WilliamMScherer: Colorado Rockies winter storm. Central Mtns: 4\"-8\"
&gt;10,000'kft. 1\"-3\" snow:9000'-10000'kft. Pikes Peak:8\"-14\"+. #COwx â€¦"
   },
   "to_user_id":{
      "string":""
   },
```

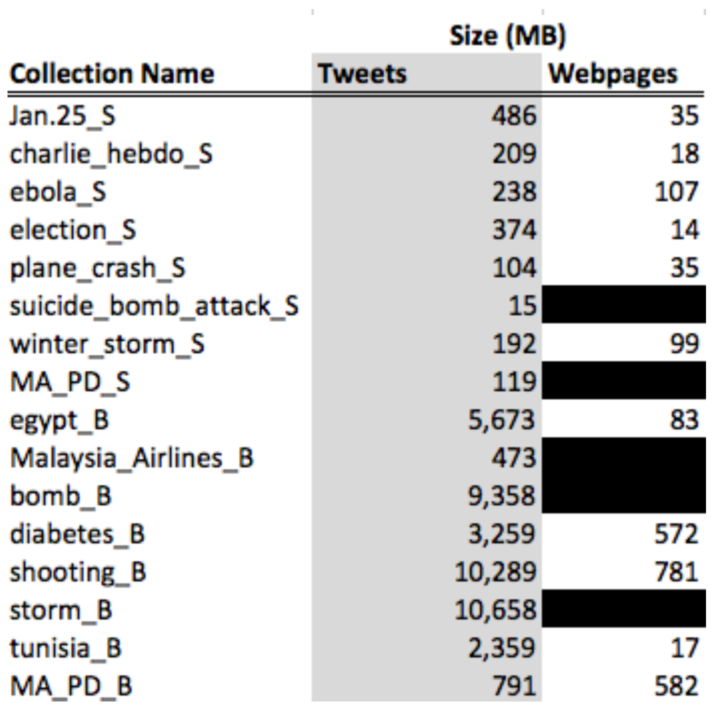
```
   "from_user":{
      "string":"gjeni_u"
   },
   "id":{
      "string":"520330344818819073"
   },
   "from_user_id":{
      "string":"1027114068"
   },
   "iso_language_code":{
      "string":"en"
   },
   "source":{
      "string":"<a href=\"http://twitter.com\" rel=\"nofollow\">Twitter Web Client</a>"
   },
   "profile_image_url":{
      "string":"http://abs.twimg.com/images/themes/theme10/bg.gif"
   },
   "geo_type":{
      "string":""
   },
   "geo_coordinates_0":{
      "double":0.0
   },
   "geo_coordinates_1":{
      "double":0.0
   },
   "created_at":{
      "string":"Thu Oct 09 21:49:56 +0000 2014"
   },
   "time":{
      "int":1412891396
   }
}
```

Web pages

Web pages crawled from Nutch come in a serialized format. We do not show this format here, but, below, we describe what data is extracted from these pages in our Avro schemas.

Sizes of the collections

Table 4 reports the sizes (in megabytes) of the tweet and web page collections. The “S” and “B” suffixes indicate that a collection is small or big, respectively. The black cells in the table correspond to data that we did not have at time of writing.



	Size (MB)	
Collection Name	Tweets	Webpages
Jan.25_S	486	35
charlie_hebdo_S	209	18
ebola_S	238	107
election_S	374	14
plane_crash_S	104	35
suicide_bomb_attack_S	15	■
winter_storm_S	192	99
MA_PD_S	119	■
egypt_B	5,673	83
Malaysia_Airlines_B	473	■
bomb_B	9,358	■
diabetes_B	3,259	572
shooting_B	10,289	781
storm_B	10,658	■
tunisia_B	2,359	17
MA_PD_B	791	582

Table 4. Sizes of the collections used in this course.

7.2.2 HBase Schemas

The HBase schema for both tweets and web pages was designed in collaboration with the Solr team. We decided to have two separate column families (i.e., column groups) for the content and metadata of a document and for the analysis data produced by each team. Below, we show the schema for tweets; the schema for web pages can be found in Appendix E.

Column Family	Column Qualifier
original	
	collection
	text_original
	text_clean
	text_clean2
	created_at
	source
	user_screen_name
	user_id
	lang
	retweet_count
	favorite_count
	contributors_id
	coordinates
	urls
	hashtags

	user_mentions_id
	in_reply_to_user_id
	in_reply_to_status_id
analysis	ner_people
	ner_locations
	ner_dates
	ner_organizations
	cluster_id
	cluster_label
	class
	social_importance
	lda_vectors
	lda_topics

An important consideration when designing an HBase table is to decide on an appropriate convention for the row ID (i.e., unique identifier) of an HBase row. Data in HBase is stored in lexicographical order; one can take advantage of this fact to design the row ID in such a way that common operations become efficient. For example, at the time of design and implementation of our project, we considered that it would be of interest for a user to get data for a particular collection. With that in mind, we decided to use the collection name as a prefix for each document. The format of the row ID is **[collection_Name]--[UID]**, where collection_name is the collection that the document belongs to and UID is a unique identifier for the document. An example of a row ID for a tweet is “Jan.25_S--100003”, indicating that the tweet is part of the Jan.25_S collection. This format for the row ID is more useful than just using a unique identifier, such as the tweet ID. However, a disadvantage of this naming convention is that bulk-loading can be inefficient and cause load balancing problems because it is not immediately obvious how to split an HBase table into balanced regions with such specific prefixes.

7.2.3 Avro Schemas

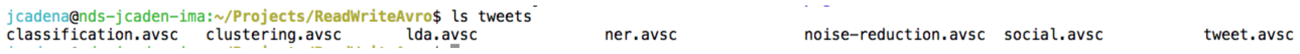
Every team processing data in HDFS has an Avro schema. For the purposes of loading HBase and standardization, we asked teams to output their results of data processing in Avro format according to our proposed schemas.
The schemas simply reflect the HBase schema. The only required field for each Avro object is “doc_id”, since we need to know the id of the document to be updated in HBase (the Noise Reduction team has some additional required fields). Null fields or empty strings will not be uploaded to HBase.
As said above, every team has a separate schema. This separation has two purposes: 1) avoiding accidental overwrites between teams and 2) being able to make changes to the schema of one team without affecting the rest of the class.
The detailed schemas for each team can be found in Appendix A.

Field Definitions

Most fields in the schema are text strings, except for a few integer fields in the Twitter schema. Fields that can be interpretable as lists (i.e., hashtags, list of clusters, list of topics, list of NER locations) are written as pipe-separated strings in Avro. For example, the list of hashtags ("#Egypt", "#Jan25", "#revolution") should be written as "Egypt | Jan25 | revolution".

Compiling the Avro Schemas

We show how to compile the Twitter schemas. The process for web pages is the same. First, we put all the schemas from the Appendix in the same directory shown in Figure. 18:



```
jcadena@nds-jcaden-ima:~/Projects/ReadWriteAvro$ ls tweets
classification.avsc   clustering.avsc      lda.avsc            ner.avsc             noise-reduction.avsc  social.avsc          tweet.avsc
```

Figure 18. Directory containing a set of Avro schemas to be compiled

Then, we run the avro-tools utility:

```
avro-tools compile schema tweets ./
```

This command generates the Java classes corresponding to the schema in the current directory, and we obtain the directory structure shown in Figure. 19:

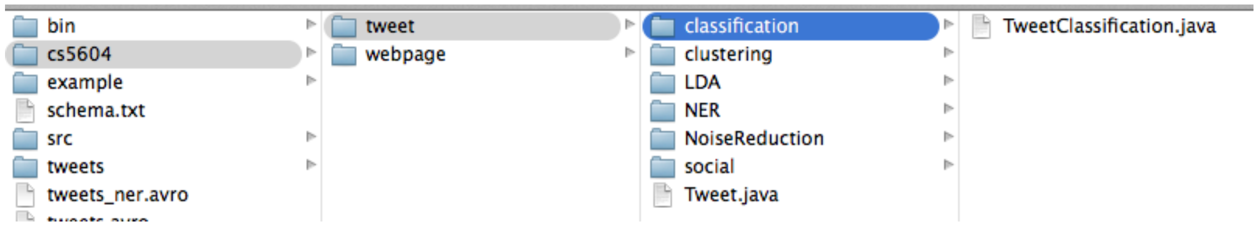


Figure 19. Directory structure of the Java classes generated from an Avro schema

Instructions on how to use the generated classes for reading and writing Avro can be found in the User's manual.

7.3 Installation

7.3.1 Installation of Solr

Solr can run on any platform as long as Java is installed. As of Solr 4.10.3, it requires Java 1.7 or greater. To check the version of Java, run `Java -version` in terminal.

Get Solr running (on Mac OS X)

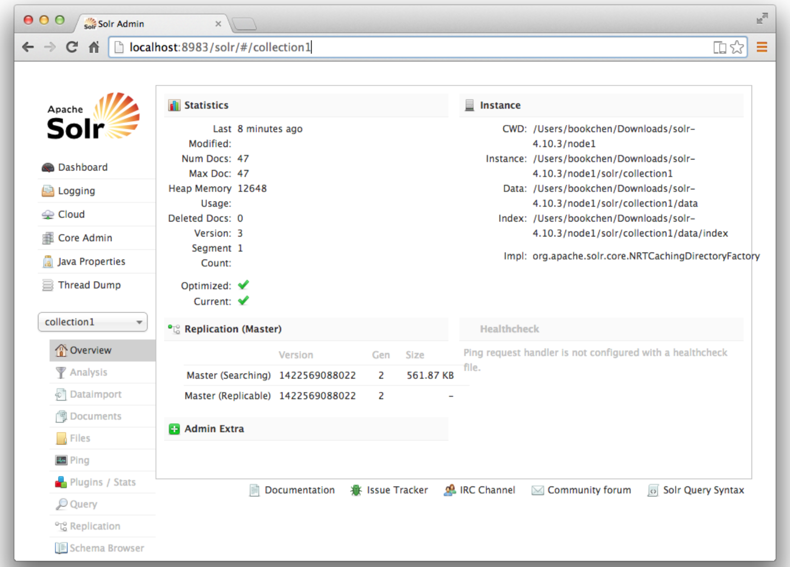
1. Download Solr 4.10.3 from http://lucene.apache.org/solr/
1. Extract the downloaded file. You will get a folder named solr-4.10.3 containing bin, docs, and example folders
2. Run `bin/solr start -e cloud -noprompt` to start

3. There are two collections created automatically, collection1 and gettingstarted. We will only use collection1 here.
4. Indexing Data: Install SimplePostTool. Set $CLASSPATH environment variable: `export CLASSPATH=dist/solr-core-4.10.3.jar`
5. Indexing a directory of "rich" files. Run `java -Dauto -Drecursive org.apache.solr.util.SimplePostTool docs/`
6. Indexing Solr XML: `java org.apache.solr.util.SimplePostTool exampledocs/*.xml`
7. Indexing JSON: `java -Dauto org.apache.solr.util.SimplePostTool example/exampledocs/books.json`
8. Indexing CSV (Comma/Column Separated Values): `java -Dauto org.apache.solr.util.SimplePostTool example/exampledocs/books.csv`

Loading data into Solr

Run `java -Dauto -Drecursive org.apache.solr.util.SimplePostTool ~/5604/data/` where `~/5604/data/` is the location of the sample collection in my computer.

Now look at the administration page of Solr at http://localhost:8983/solr/#/collection1



Statistics

```
         Last  8 minutes ago
     Modified:
    Num Docs:  47
     Max Doc:  47
 Heap Memory   12648
       Usage:
Deleted Docs:  0
     Version:  3
     Segment   1
       Count:

   Optimized:  ✓
     Current:  ✓
```

Instance

```
      CWD:  /Users/bookchen/Downloads/solr-
            4.10.3/node1
 Instance:  /Users/bookchen/Downloads/solr-
            4.10.3/node1/solr/collection1
     Data:  /Users/bookchen/Downloads/solr-
            4.10.3/node1/solr/collection1/data
    Index:  /Users/bookchen/Downloads/solr-
            4.10.3/node1/solr/collection1/data/index

     Impl:  org.apache.solr.core.NRTCachingDirectoryFactory
```

Replication (Master)

	Version	Gen	Size
Master (Searching)	1422569088022	2	561.87 KB
Master (Replicable)	1422569088022	2	-

Healthcheck

Ping request handler is not configured with a healthcheck file.

Admin Extra

Figure 20. Administration page of Solr

In Figure. 20 you can see all the 47 files have been indexed.

By using the *Query* tab, you can search all the indexed documents. The results will be shown in JSON format.

7.3.2 Installation of Hadoop

Use the virtual machine version of Hadoop from
http://www.cloudera.com/content/cloudera/en/downloads/quickstart_vms/cdh-5-3-x.html

After you enter the cluster, run `source /etc/my.sh`. This will set up all the environment variables. To run a simple example, execute the following commands:

```
cd $HADOOP_PREFIX            # which is /usr/local/hadoop
bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.6.0.jar grep input
output 'dfs[a-z.]+'
```

Run an example of HDFS:

We follow the tutorial from [11]:

1. Download example input data
2. Restart the Hadoop cluster:
   `$/usr/local/hadoop/bin/start-all.sh`
3. Copy local example data to HDFS:

```
      /bin/hadoop dfs -copyFromLocal /Download_datapath/ /HDFS directory store input
data/
```

Run the MapReduce job

```
      $bin/hadoop jar hadoop*examples*.jar command /HDFS directory store input data/
/HDFS Directory store output/
```

Retrieve the job results from HDFS

1. Read it directly:

```
      $bin/hadoop dfs -cat /HDFS Directory store output/part-r-00000
```

2. Or copy it from HDFS directory to local directory:

```
      $mkdir /local directory/
      $bin/hadoop dfs -getmerge /HDFS Directory store output/  /local directory/
```

7.3.3 Installation of Apache Nutch

Since we want to use Hadoop cluster, the binary distribution of Nutch doesn't work in this case. The distribution works well in "local mode", i.e., not using Hadoop, but from what we could determine, there is not an "apache-nutch-1.9.job" file in the binary distribution, which is required

to be deployed to all the data nodes in the cluster. For this reason, we build Nutch from the source code.
To start crawling web pages, we must declare a crawler in the Nutch configuration file, “conf/nutch-site.xml”. We add the following lines:

```
<property>
 <name>http.agent.name</name>
 <value>MyCrawler</value>
</property>
```

7.4 Operation

7.4.1 MapReduce Introduction

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster[6].

The name MapReduce comes from functional programming:

- **Map** is the name of a higher-order function that applies a given function to each element of a list. Example:

```
val numbers = List(1,2,3,4,5)

numbers.map(x => x * x) == List(1,4,9,16,25).
```

- **Reduce** is the name of a higher-order function that analyze a recursive data structure and recombines, through use of a given combining operation, the results of recursively processing its constituent parts, building up a return value. Example

```
val numbers = List(1,2,3,4,5)

numbers.reduce(_ + _) == 15.
```

MapReduce takes an input, splits it into smaller parts, executes the code of the mapper on every part, shuffles and sorts all the results, then gives them to one or more reducers that merge all the results into one.
The canonical first example of a MapReduce program is counting words in a big collection of documents. An implementation of this example can be found in the Apache Hadoop website [17] and is illustrated in Figure 21. The Mapper task maps each word in a line to the number 1 and sends this pair to the Shuffler and Sorter. Once the keys are sorted, the Reducer task simply aggregates the number of times a unique key appears to obtain the final word count.

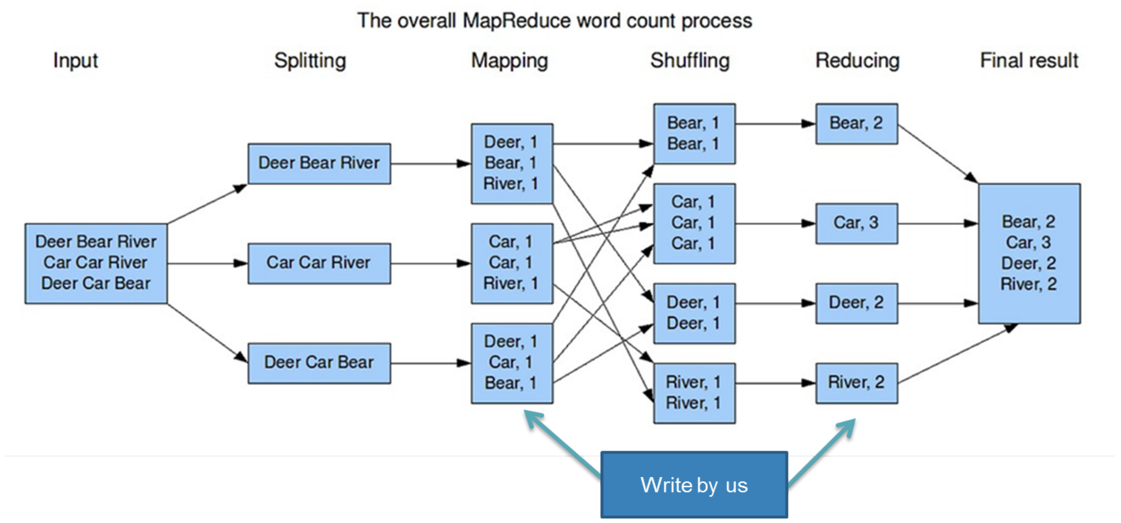


Figure 21. Example of a simple MapReduce program for counting words. Each mapper emits a <word, 1> key-value pair. The shufflers sort these pairs lexicographically by key and send them to the reducers, which simply add the counts for each word to obtain a final count.

7.4.2 Loading Data Into HBase

Non-Distributed Program

We wrote a program to load Avro data into HBase sequentially: *hbase-loader.jar*. This program can load a small tweet collection (around 500 MBs of data) into HBase in under two minutes. The code can be found in Appendix B. The code was compiled into a runnable jar file called hbase-loader.jar. Instructions on how to run the jar file can be found in the User's manual.

MapReduce Program

hbase-loader.jar does not scale to the big collections, which are between one and two orders of magnitude larger in size than the small collections. Trying to load a big collection from the main node with *hbase-loader.jar* throws the Java *OutOfMemory* exception, even when changing the default settings of the Java Virtual Machine to use all the available memory in the node. We developed a MapReduce version of the program to load the big collections: *mr-hbase-loader.jar*. This program is a Map-only task that reads Avro files from HDFS and writes the deserialized data to HBase. The code can be found in the Appendix.

We note that the developer should set cluster-specific configuration in the main function of the MapReduce program. For example, in the Hadoop cluster used in this class, Zookeeper was configured to run in nodes 1, 2, and 3. The default HBase configuration expects Zookeeper to be running in each node. We can adjust configuration settings in the *config* object of the MapReduce program (See Appendix C) for an example.

Bulk-Loading Program

When using HBase programmatically, as in the two implementations above, the data to be loaded traverses the entire write-path depicted in Figure 22. First, data for a row is received by the Region Server, which is a program that determines the region in the cluster that the row belongs to; we note that data in HBase is stored in lexicographical order by key, so a region contains keys in this order, and, when amount of rows in the regions becomes too large, the Region Server splits it into two. After the row gets to a region, it then goes to a local MemStore, where the data is kept in memory temporarily until a certain number of operations have been submitted to the region. At that time, the row data is written to an HFile for long-term storage. It is possible to bypass the HBase workflow by writing HFiles and directly writing those files to the appropriate region in HBase, as described in [26]. The three main parts of the process are

1. Pre-splitting the HBase table, so that each region has a roughly balanced number of rows.
2. Writing the HFiles using a MapReduce program (our program is in Appendix F).
3. Using the HBase libraries to load the HFiles.

Disadvantages of bulk-loading

One main challenge in the bulk-loading process is pre-splitting the destination table into an appropriate number of regions. When we create the HFiles to be uploaded into HBase (step 2 above), the number of reducers in the MapReduce job is equal to the number of regions we create in step 1. In fact there is a one-to-one mapping between reducers and regions in that each reducer is in charge of processing all the data that is going to ultimately reside in the corresponding region. If the HBase table is pre-split in such a way that most of the input data goes to one single region (i.e., the load is unbalanced), then one of the reducers in the MapReduce job will be heavily loaded, and we will essentially be creating the HFiles sequentially, losing the distributed computing advantage.

We have explained why a good initial split of an HBase table is important, but not how to get good splits. Before splitting the destination table, it is necessary to have a good understanding of the distribution of the row IDs in our data. Sometimes, the data is naturally well-distributed. For example, suppose that the row IDs of our dataset are 8-digit numbers uniformly distributed in the range [00000000, 99999999]. Then, it is sufficient to pre-split the HBase table into 10 regions with the split points being the natural numbers from 0 to 9. If, on the other hand, our row IDs are words in the English dictionary, pre-splitting the table into 26 regions, one for each letter of the alphabet, would not give us a balanced load, since some starting letters are more common than others (i.e., there are more words starting with “T” than words starting with “X”). Therefore, pre-splitting requires us to do a preliminary analysis of the data to estimate the row ID distribution.

An HBase schema designer who foresees using bulk-loading should incorporate “uniformity” as one aspect to consider when deciding on a row ID. For example, in our schema for the IDEAL project, we are using the collection name as a prefix for the row ID of a document –for instance, “egypt_B--00012”. Retrieving data for a particular collection can be done efficiently with this ID format –that was one of our goals in the design. However, it is going to be very hard to keep the load balanced across regions with these row IDs, unless we know in advance the names of most of the collections that we are interested in. If we knew that Solr is going to be the only “user” of HBase and that we are not interested on optimizing any specific user queries, we could change the ID to something better distributed yet meaningless to a person. A key that balances the opposing goals of being well-distributed yet informative is left as an open question.

Another disadvantage of bulk-loading specific to our project is that, by skipping the HBase write-path, we are also avoiding the table replication mechanism. Briefly, table replication is a process by which the data of an HBase table in one cluster is copied to another cluster. The real-time Lily Indexer that we are using to load data from HBase into Solr critically depends on table replication. Therefore, if we use bulk-loading, we are limiting to only loading Solr in batches of data and not in real time.

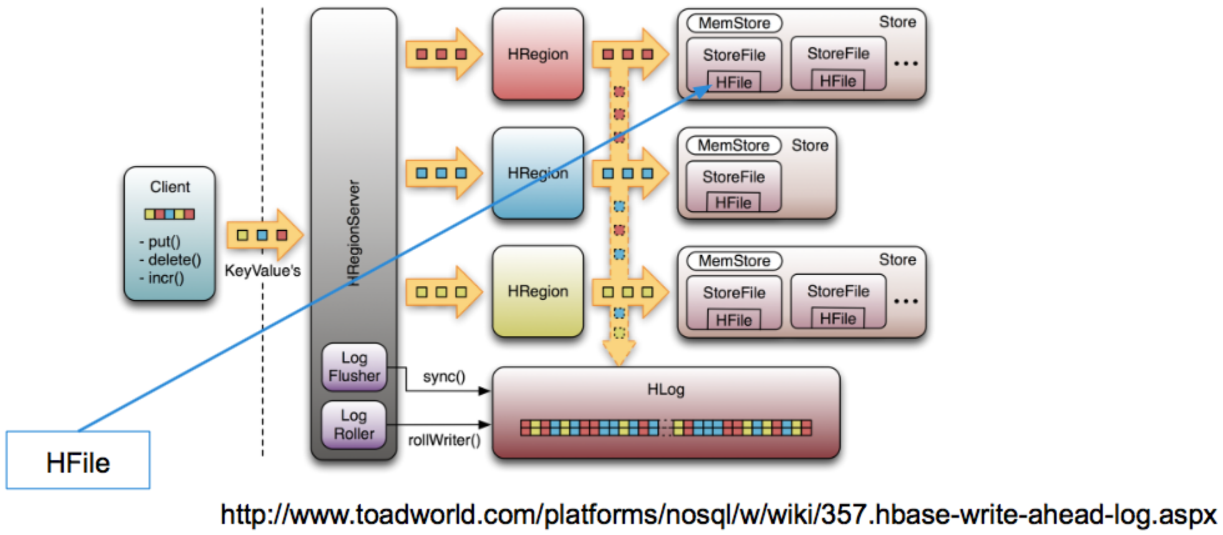


Figure 22. Depiction of the HBase write-path. Data to be inserted into an HBase table is first received by a Region Server that is in charge of sending the data to its appropriate Region in the cluster (based on row ID). The Region server handles a local memory space and decides when to write data to disk in the form of HFiles.

8. Conclusions and Future Work

8.1 Conclusions

Our proposed solution makes it easy to integrate the results from different teams working independently. We found that we could combine results from all the teams in HBase incrementally very quickly. Our ultimate goal, of course, is loading data into Solr efficiently and indexing new results as we get them. We could not scale the indexing process to the big collections by the end of the course, but we believe that it is simply a matter of giving the Solr installation more computing resources.

Regarding our use of Avro for standardization of the data, we conclude that it was a good design decision to use this format. Avro is well-supported by the Hadoop infrastructure (i.e., easy to handle in HDFS and via MapReduce) and supports versioning, which is a very important property for a young project like IDEAL. Having a schema for each team was valuable during the initial stages of the class where we still had to settle details about the output of each team and the data needed in Solr, and we believe that this property will also be valuable at all points of the project as new ways to analyze the collections are proposed.

As a general, Hadoop is an appropriate framework for a project like IDEAL. Most teams were able to do their analysis in a matter of minutes using existing tools for Hadoop, and the few teams that had to write their own MapReduce programs did not have any major difficulties –an exception being the Social Networks team, who had to use a different framework.

8.2 Future Work

We name just a few open challenges. First, we wrote programs for most of the data handling tasks, such as pre-processing data and uploading to HBase. Writing specialized programs has a development cost that could be avoided using existing off-the-shelf tools. For example, we could use the Apache Pig [21] platform to move data in Avro format from HDFS to HBase. Apache Pig also abstracts many common data analysis tasks, such as summarization. Teams can make use of this tool instead of writing their own programs. Second, even though Hadoop was a good fit in the class project, we recommend researching Spark. Spark is an engine for large-scale data processing. The engine can run either in standalone mode or in a distributed system like Hadoop. According to the Spark website, this framework can run in as little as 8 GBs of memory per node (in a distributed setting), and scale to "hundreds of gigabytes." Thus, the Hadoop cluster that we used has more than enough resources for accommodating Spark. When possible, Spark works with data in memory, thus avoiding costly disk I/O operations. In practice, Spark has been shown to be faster than MapReduce. Recently, Spark won that Daytona Graysort Contest, being the fastest open-source engine to sort 1 petabyte (1000 terabytes) of data [25]. Furthermore, the engine is unarguably a superior choice when it comes to running iterative algorithms ---which have to be scheduled as a sequence of jobs in MapReduce. For our use case, running iterative algorithms efficiently is very important, since many machine-learning tasks, such as K-means clustering and PageRank, are iterative in nature. Another place for improvement is performance tuning in the Hadoop cluster. Analyzing the log files produced by the many MapReduce jobs run through the semester would give us insights on how to make the jobs run faster. For this course, speed was *generally* not a problem, so we leave performance tuning for future work. Finally, as mentioned in Section 7.4.2, if we want to use a bulk-loading approach for loading data into HBase, it will be necessary to formulate a row ID format that gives a better load balancing than the current format.

9. Inventory of VTechWorks Files

1. Final report: HadoopFinalReport.pdf
2. Final report for editing: HadoopFinalReport.docx
3. Final presentation: HadoopPresentation.pdf
4. Final presentation for editing: HadoopPresentation.pptx
5. Code: hadoop_team_code.tar.gz
    a. Avro schema for tweets and webpages
    b. Our three programs for HBase loading

10. References

[1] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A distributed storage system for structured data." *ACM Transactions on Computer Systems (TOCS)* 26, no. 2 (2008): 4.

[2] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51, no. 1 (2008): 107-113.

[3] Ferrera, Pedro, Ivan de Prado, Eric Palacios, Jose Luis Fernandez-Marquez, and Giovanna Di Marzo Serugendo. "Tuple MapReduce: Beyond Classic MapReduce." In *ICDM*, pp. 260-269. 2012.

[4] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." In *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 29-43. ACM, 2003.

[5] Lin, Jimmy, and Chris Dyer. "Data-intensive text processing with MapReduce." *Synthesis Lectures on Human Language Technologies* 3, no. 1. "Morgan & Claypool Publishers", (2010): 1-177.

[6] Iacono, Andrea. "MapReduce by examples." Accessed March 22, 2015, http://www.slideshare.net/andreaiacono/mapreduce-34478449.

[7] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Vol. 1. "Cambridge: Cambridge University Press", 2008.

[8] Miner, Donald, and Adam Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. " O'Reilly Media, Inc.", 2012.

[9] Yahoo! Developer Network. "Module 2: The Hadoop File System", Accessed February 12, 2015, https://developer.yahoo.com/hadoop/tutorial/module2.html

[10] Reid, Gabriel "Lily Indexer by NGDATA." NGDATA, Accessed May 5, 2015, http://ngdata.github.io/hbase-indexer/

[11] Noll, Michael. "Running Hadoop on Ubuntu Linux (Single-Node Cluster)." Accessed February 12, 2015, http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/

[12] Rathbone, Matthew. "A Beginners Guide to Hadoop", Accessed February 12, 2015, http://blog.matthewrathbone.com/2013/04/17/what-is-hadoop.html

[13] *Apache Hadoop 2.6.0 - Hadoop MapReduce Next Generation 2.6.0 - Setting up a Single Node Cluster*. “The Apache Software Foundation”, Accessed February 5, 2015, http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SingleCluster.html.

[14] "Apache Nutch." The Apache Software Foundation, Accessed March 28, 2015, http://nutch.apache.org

[15] “Apache Solr” The Apache Software Foundation, Accessed February 12, 2015, http://lucene.apache.org/solr/

[16] "Apache Spark." The Apache Software Foundation, Accessed April 12, 2015, https://spark.apache.org

[17] “MapReduce Tutorial.” The Apache Software Foundation, Accessed February 12, 2015, http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[18] “Welcome to Apache Avro.” The Apache Software Foundation, Accessed May 5, 2015, https://avro.apache.org/

[19] “Welcome to Apache Hadoop.” The Apache Software Foundation, Accessed February 12, 2015, http://hadoop.apache.org

[20] “Welcome to Apache Hive.” The Apache Software Foundation, Accessed February 12, 2015, http://hive.apache.org

[21] “Welcome to Apache Pig.” The Apache Software Foundation, Accessed February 12, 2015, http://pig.apache.org

[22] “Welcome to Apache ZooKeeper.” The Apache Software Foundation, Accessed February 12, 2015. http://zookeeper.apache.org

[23] “What is Mahout?” The Apache Software Foundation, Accessed February 12, 2015, http://mahout.apache.org

[24] Xin, Reynold S., Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. "Graphx: A resilient distributed graph system on spark." In *First International Workshop on Graph Data Management Experiences and Systems*, p. 2. ACM, 2013.

[25] Xin, Reynold. “Spark the fastest open source engine for sorting a petabyte.” Databricks, Accessed April 12, 2015, http://databricks.com/blog/2014/10/10/spark-petabyte-sort.html

[26] Cryans, Jean-Daniel. “How-to: Use HBase Bulk Loading, and Why.” Cloudera, Accessed May 5, 2015, http://blog.cloudera.com/blog/2013/09/how-to-use-hbase-bulk-loading-and-why/

Appendix

A. Avro Schemas

In this section, we list the Avro schemas that we designed for use by the different teams (except for Solr and Hadoop). See Section 7.2.3 for more information about the design decisions.

Tweets

Noise Reduction:

```
{"namespace": "cs5604.tweet.NoiseReduction",
 "type": "record",
 "name": "TweetNoiseReduction",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "original", "name": "tweet_id", "type": "string"},
    {"doc": "original", "name": "text_clean", "type": "string"},
    {"doc": "original", "name": "text_original", "type": "string"},
    {"doc": "original", "name": "created_at",  "type": "string"},
    {"doc": "original", "name": "user_screen_name", "type": "string"},
    {"doc": "original", "name": "user_id", "type": "string"},
    {"doc": "original", "name": "source", "type": ["string", "null"]},
    {"doc": "original", "name": "lang", "type": ["string", "null"]},
    {"doc": "original", "name": "favorite_count", "type": ["int", "null"]},
    {"doc": "original", "name": "retweet_count", "type": ["int", "null"]},
    {"doc": "original", "name": "contributors_id", "type": ["string", "null"]},
    {"doc": "original", "name": "coordinates", "type": ["string", "null"]},
    {"doc": "original", "name": "urls", "type": ["string", "null"]},
    {"doc": "original", "name": "hashtags", "type": ["string", "null"]},
    {"doc": "original", "name": "user_mentions_id", "type": ["string", "null"]},
    {"doc": "original", "name": "in_reply_to_user_id", "type": ["string", "null"]},
    {"doc": "original", "name": "in_reply_to_status_id", "type": ["string", "null"]},
    {"doc": "original", "name": "text_clean2", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "collection", "type": ["null", "string"], "default": null}

 ]
}
```

Clustering:

```
{"namespace": "cs5604.tweet.clustering",
 "type": "record",
 "name": "TweetClustering",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "cluster_label", "type": ["string", "null"]},
    {"doc": "analysis", "name": "cluster_id", "type": ["string", "null"]}
 ]
}
```

NER:

```
{"namespace": "cs5604.tweet.NER",
 "type": "record",
 "name": "TweetNER",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "ner_people", "type": ["string", "null"]},
    {"doc": "analysis", "name": "ner_locations", "type": ["string", "null"]},
    {"doc": "analysis", "name": "ner_dates", "type": ["string", "null"]},
    {"doc": "analysis", "name": "ner_organizations", "type": ["string", "null"]}
 ]
}
```

Social Network:

```
{"namespace": "cs5604.tweet.social",
 "type": "record",
 "name": "TweetSocial",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "social_importance", "type": ["double", “null”], “default”: 0},
 ]
}
```

Classification:

```
{"namespace": "cs5604.tweet.classification",
 "type": "record",
 "name": "TweetClassification",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "class", "type": ["string", “null”]},
 ]
}
```

LDA

```
{"namespace": "cs5604.tweet.LDA",
 "type": "record",
 "name": "TweetLDA",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "lda_topics", "type": ["string", “null”]},
    {"doc": "analysis", "name": "lda_vectors", "type": ["string", “null”]}
 ]
}
```

Web pages

Noise Reduction

```
{"namespace": "cs5604.webpage.NoiseReduction",
 "type": "record",
 "name": "WebpageNoiseReduction",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "original", "name": "text_clean", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "text_original", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "created_at",  "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "accessed_at",  "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "author", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "subtitle", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "section", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "lang", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "coordinates", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "urls", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "content_type", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "text_clean2", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "collection", "type": ["null", "string"], "default": null} ,
    {"doc": "original", "name": "url", "type": ["null", "string"], "default": null},
    {"doc": "original", "name": "appears_in_tweet_ids", "type": ["null", "string"], "default": null}
 ]
}
```

Clustering

```
{"namespace": "cs5604.webpage.clustering",
 "type": "record",
 "name": "WebpageClustering",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "cluster_label", "type": ["string", "null"]},
    {"doc": "analysis", "name": "cluster_id", "type": ["string", "null"]}
 ]
}
```

NER

```
{"namespace": "cs5604.webpage.NER",
 "type": "record",
 "name": "WebpageNER",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "ner_locations", "type": ["string", "null"]},
    {"doc": "analysis", "name": "ner_people", "type": ["string", "null"]},
    {"doc": "analysis", "name": "ner_dates", "type": ["string", "null"]},
    {"doc": "analysis", "name": "ner_organizations", "type": ["string", "null"]}
 ]
}
```

Social Network:

```
{"namespace": "cs5604.webpage.social",
 "type": "record",
 "name": "WebpageSocial",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "social_importance", "type": ["double", “null”], “default”: 0},
 ]
}
```

Classification:

```
{"namespace": "cs5604.webpage.classification",
 "type": "record",
 "name": "WebpageClassification",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "class", "type": ["string", “null”]},
 ]
}
```

LDA

```
{"namespace": "cs5604.webpage.LDA",
 "type": "record",
 "name": "WebpageLDA",
 "fields": [
    {"name": "doc_id", "type": "string"},
    {"doc": "analysis", "name": "lda_topics", "type": ["string", “null”]},
    {"doc": "analysis", "name": "lda_vectors", "type": ["string", “null”]},
 ]
}
```

B. AvroToHBase.java

In this section, we show the centralized program that we use to load data into HBase (See Section 7.4.2). Our program takes as input an Avro file. The main routine is a while loop that reads each record in the input file and stores it in an array. When all the records have been read, the data is put into HBase with a single call to the API.

```
package cs5604.hadoop;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import org.apache.avro.Schema;
import org.apache.avro.file.DataFileReader;
import org.apache.avro.io.DatumReader;
import org.apache.avro.specific.SpecificDatumReader;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.conf.Configuration;
import cs5604.tweet.Tweet;
import cs5604.webpage.Webpage;

public class AvroToHBase {

	public static enum DataType {
		tweets,
		webpages
	};

	/**
	 * @param args
	 * @throws IOException
	 */
	public static void main(String[] args) throws IOException {
		// check command line arguments
		if (args.length != 2) {
			System.out.println("Number of arguments should be 2. " +  args.length + " arguments 
found.");
			System.out.println("usage: java -jar hbase-loader.jar AVRO_FILE {tweets, webpages}");
			System.out.println("example: java -jar hbase-loader.jar tweet_clusters.avro tweets");
			System.out.println("example: java -jar hbase-loader.jar webpage_importance.avro 
webpages");
			System.exit(1);
		}
		String dataFile = args[0];
		DataType dt = null;

		try {
			dt = DataType.valueOf(args[1]);
		} catch (IllegalArgumentException exception) {
			System.out.println("Data type must be either \"tweets\" or \"webpages\". \"" + args[1] + "\" 
was found.");
```

```
                System.exit(1);
            }
            String tableName = args[1];

            // connect to HBase
            Configuration config = HBaseConfiguration.create();
            HTable table = new HTable(config, tableName);
            ArrayList<Put> puts = new ArrayList<Put>();

            switch(dt) {
                case tweets:
                    DatumReader<Tweet> datumReader = new
SpecificDatumReader<Tweet>(Tweet.class);
                DataFileReader<Tweet> dataFileReader = new DataFileReader<Tweet>(new File(dataFile),
datumReader);
                Tweet record = null;


                while (dataFileReader.hasNext()) {
                    // Reuse user object by passing it to next(). This saves us from
                    // allocating and garbage collecting many objects for files with
                    // many items.
                    record = dataFileReader.next(record);
                    //System.out.println(record);

                    Put p = new Put(Bytes.toBytes(record.getDocId().toString()));

                    for (Schema.Field field : record.getSchema().getFields()) {
                        // skip row id
                        if (field.name().equals("doc_id")) {
                            continue;
                        }
                        String columnFamily = field.doc();
                        String columnQualifier = field.name();
                        assert(columnFamily.length() > 0);

                        Object value = record.get(columnQualifier);

                        if (value != null && ! (value.toString().isEmpty())) {
                            //System.out.println("Value is " + value);
                            p.add(Bytes.toBytes(columnFamily), Bytes.toBytes(columnQualifier),
Bytes.toBytes(value.toString()));
                        }
                    }
                    if (!p.isEmpty()) {
                        // add to buffer
                        puts.add(p);
                    } else {
                        System.out.println("WARN: All fields are null for row with id " + record.getDocId()
+
                                    ". Nothing to add.");
                    }
                }
                dataFileReader.close();
```

```
                        break;
                            case webpages:
                                    DatumReader<Webpage> daReader = new
SpecificDatumReader<Webpage>(Webpage.class);
                        DataFileReader<Webpage> daFileReader = new DataFileReader<Webpage>(new
File(dataFile), daReader);
                        Webpage rec = null;


                        while (daFileReader.hasNext()) {
                                // Reuse user object by passing it to next(). This saves us from
                                // allocating and garbage collecting many objects for files with
                                // many items.
                                rec = daFileReader.next(rec);
                                //System.out.println(record);

                                Put p = new Put(Bytes.toBytes(rec.get("doc_id").toString()));

                                for (Schema.Field field : rec.getSchema().getFields()) {
                                    //System.out.println("Field is " + field.name());
                                    //System.out.println("Family is " + field.doc());
                                    // skip row id
                                    if (field.name().equals("doc_id")) {
                                            continue;
                                    }
                                    String columnFamily = field.doc();
                                    String columnQualifier = field.name();
                                    assert(columnFamily.length() > 0);

                                    Object value = rec.get(columnQualifier);

                                    if (value != null) {
                                            //System.out.println("Value is " + value);
                                            p.add(Bytes.toBytes(columnFamily), Bytes.toBytes(columnQualifier),
Bytes.toBytes(value.toString()));
                                    }
                                }
                            // add to buffer
                            puts.add(p);
                        }
                        daFileReader.close();
                        break;
                      default:
                            System.out.println("Reached unexpected switch statement case. Quitting.");
                            System.exit(1);
                    }

        // write to HBase
                    table.put(puts);
                    table.flushCommits();
                    table.close();

        System.out.println("" + puts.size() + " rows were written to the " + tableName + " table");
        }
}
```

C. WriteHBaseMR.java

Here, we present a MapReduce program to insert data into HBase through the HBase API. The program takes as input an Avro file (from HDFS) and writes records to HBase in the Mapper task. A lot of the code below is for setting up the job and validating the user input; the most interesting parts are the TweetMapper and WebpageMapper methods, which process tweets and web pages, respectively.

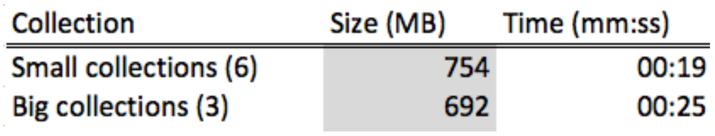
```
package cs5604.hadoop;

import java.io.IOException;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.avro.Schema;
import org.apache.avro.mapred.AvroKey;
import org.apache.avro.mapreduce.AvroJob;
import org.apache.avro.mapreduce.AvroKeyInputFormat;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import cs5604.tweet.Tweet;
import cs5604.webpage.Webpage;

public class WriteHBaseMR extends Configured implements Tool{
        public static String tableName;
        // allowed HBase table names
        public static enum TableName {
                tweets,
                webpages,
                test_tweets,
                test_webpages
        };

        public static class TweetMapper extends
                Mapper<AvroKey<Tweet>, NullWritable, ImmutableBytesWritable, Put> {
```

```
			protected void map(AvroKey<Tweet> key, NullWritable value, Context context)
					throws IOException, InterruptedException {
				Tweet t = key.datum();
				ImmutableBytesWritable s = new
ImmutableBytesWritable(Bytes.toBytes(t.getDocId().toString()));

				Put p = datumToPut(t);
				if (!p.isEmpty()) {
					context.write(s, datumToPut(t));
			 } else {
				System.out.println("WARN: All fields are null for row with id " +
t.getDocId() +
							      ". Nothing to add.");
			 }
			}

			private static Put datumToPut(Tweet t) throws IOException {
				Put p = new Put(Bytes.toBytes(t.getDocId().toString()));
			 for (Schema.Field field : t.getSchema().getFields()) {
				// skip row id
				if (field.name().equals("doc_id")) {
					continue;
				}
				String columnFamily = field.doc();
				String columnQualifier = field.name();
				assert(columnFamily.length() > 0);

				Object field_value = t.get(columnQualifier);

				if (field_value != null && ! (field_value.toString().isEmpty())) {

					p.add(Bytes.toBytes(columnFamily),
Bytes.toBytes(columnQualifier), Bytes.toBytes(field_value.toString()));
				}
			 }
			 return p;
		}

		}


		public static class WebpageMapper extends
			Mapper<AvroKey<Webpage>, NullWritable, ImmutableBytesWritable, Put> {
			protected void map(AvroKey<Webpage> key, NullWritable value, Context context)
					throws IOException, InterruptedException {
				Webpage w = key.datum();
				ImmutableBytesWritable s = new
ImmutableBytesWritable(Bytes.toBytes(w.getDocId().toString()));

				Put p = datumToPut(w);
				if (!p.isEmpty()) {
					context.write(s, datumToPut(w));
```

```
                } else {
                        System.out.println("WARN: All fields are null for row with id " +
w.getDocId() +
                                                        ". Nothing to add.");
                }


            }

            private static Put datumToPut(Webpage w) throws IOException {
                    Put p = new Put(Bytes.toBytes(w.getDocId().toString()));
                for (Schema.Field field : w.getSchema().getFields()) {
                    // skip row id
                    if (field.name().equals("doc_id")) {
                            continue;
                    }
                    String columnFamily = field.doc();
                    String columnQualifier = field.name();
                    assert(columnFamily.length() > 0);

                    Object field_value = w.get(columnQualifier);

                    if (field_value != null && ! (field_value.toString().isEmpty())) {

                            p.add(Bytes.toBytes(columnFamily),
Bytes.toBytes(columnQualifier), Bytes.toBytes(field_value.toString()));
                    }
                }
                    return p;
            }
        }

        @Override
        public int run(String[] rawArgs) throws Exception {
                if (rawArgs.length != 3) {
                        System.err.printf("Usage: %s [generic options] <input> <output>
<tableName>\n",
                                        getClass().getName());
                        ToolRunner.printGenericCommandUsage(System.err);
                        return -1;
                }

                TableName tn = null;
                String[] args = new GenericOptionsParser(rawArgs).getRemainingArgs();
                try {
                        tn = TableName.valueOf(args[2]);
                } catch (IllegalArgumentException exception) {
                        System.out.println("Data type must be either \"tweets\" or \"webpages\".
\"" + args[2] + "\" was found.");
                        return -1;
                }
                tableName = args[2];
                Configuration config = HBaseConfiguration.create();
                config.set("mapreduce.task.timeout", "300000");
                // cluster-specific configuration. In the production cluster
                // zookeeper only runs on nodes 1, 2, and 3
                config.set("hbase.zookeeper.quorum",
"node1.dlrl:2181,node2.dlrl:2181,node3.dlrl:2181");
                 // set the job name to be “hbase-load” and the name of input file
                Job job = Job.getInstance(config, "hbase-load - " + args[0]);

                job.setJarByClass(WriteHBaseMR.class);

                Path inPath = new Path(args[0]);
                Path outPath = new Path(args[1]);

                FileInputFormat.setInputPaths(job, inPath);
                FileOutputFormat.setOutputPath(job, outPath);
                outPath.getFileSystem(super.getConf()).delete(outPath, true);

                job.setInputFormatClass(AvroKeyInputFormat.class);

                // call appropriate mapper for tweets or webpages
                switch(tn) {
                        case test_tweets:
                        case tweets:
                                job.setMapperClass(TweetMapper.class);
                                AvroJob.setInputKeySchema(job, Tweet.getClassSchema());
                                break;
                        case test_webpages:
                        case webpages:
                                job.setMapperClass(WebpageMapper.class);
                                AvroJob.setInputKeySchema(job, Webpage.getClassSchema());
                                break;
                        default:
                                System.out.println("Reached unexpected switch statement case.
Quitting.");
                        return -1;
                }

                TableMapReduceUtil.addDependencyJars(job);
                TableMapReduceUtil.initTableReducerJob(tableName, null, job);

                // Map only job. Set number of reducers to 0
                job.setNumReduceTasks(0);

                return (job.waitForCompletion(true) ? 0 : 1);
        }

        public static void main(String[] args) throws Exception {
                int result = ToolRunner.run(new WriteHBaseMR(), args);
                System.exit(result);
        }
}
```

D. Loading time for webpages

At the time of writing, we only had access to cleaned web page data for six small collections (LDA was missing) and three big collections (Clustering, Noise Reduction, and Solr teams). Table D.1 reports the time taken to upload the data using our MapReduce program.



Collection	Size (MB)	Time (mm:ss)
Small collections (6)	754	00:19
Big collections (3)	692	00:25

Table D.1 Time taken to load the web page collections into HBase using MapReduce and the HBase API

E. HBase Schema for webpages

Below is the HBase schema for web pages. The "analysis" column family is identical to the column family for tweets with the same name. The "document" family is meant to store data specific to a web page, such as the content or the domain.

Table: webpages [rowkey: uuid]

Column Family	Column Qualifier
document	title
	collection
	domain
	text_original
	text_clean1
	text_clean2
	author
	subtitle
	created_at
	section
	urls
	twitter_link
	facebook_link
	google_plus_link
	pinterest
	coordinates
analysis	ner_people
	ner_locations
	ner_dates
	ner_organizations
	cluster_id
	cluster_label
	class
	social_importance
	lda_vectors

lda_topics

F. HFile Generation

Here, we show our MapReduce program to generate HFiles for the bulk-loading process (See Section 7.4.2). The mappers (for tweet and web pages) convert the input file (Avro) to a sequence of “Put” HBase operations. A reducer that comes with the HBase distribution handles these “Put” objects, but that part is opaque to the programmer. The programmer simply calls the right reducer using this statement: HFileOutputFormat.configureIncrementalLoad(job, htable);

```
package cs5604.hadoop;

import java.io.IOException;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.mapreduce.HFileOutputFormat;
import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.avro.Schema;
import org.apache.avro.mapred.AvroKey;
import org.apache.avro.mapreduce.AvroJob;
import org.apache.avro.mapreduce.AvroKeyInputFormat;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import cs5604.tweet.Tweet;
import cs5604.webpage.Webpage;

public class HBaseBulkload extends Configured implements Tool{
        public static String tableName;

        public static enum TableName {
                tweets,
                webpages,
                test_tweets,
                test_webpages,
                bulk_tweets,
                bulk_webpages
        };

        public static class TweetMapper extends
                Mapper<AvroKey<Tweet>, NullWritable, ImmutableBytesWritable, Put> {
                protected void map(AvroKey<Tweet> key, NullWritable value, Context context)
                                throws IOException, InterruptedException {
                        Tweet t = key.datum();
                        ImmutableBytesWritable s = new
ImmutableBytesWritable(Bytes.toBytes(t.getDocId().toString()));

                        Put p = datumToPut(t);
                        if (!p.isEmpty()) {
                                context.write(s, datumToPut(t));
              } else {
                System.out.println("WARN: All fields are null for row with id " + t.getDocId() +
                                        ". Nothing to add.");
              }
                }

                private static Put datumToPut(Tweet t) throws IOException {
                        Put p = new Put(Bytes.toBytes(t.getDocId().toString()));
              for (Schema.Field field : t.getSchema().getFields()) {
                // skip row id
                if (field.name().equals("doc_id")) {
                        continue;
                }
                String columnFamily = field.doc();
                String columnQualifier = field.name();
                assert(columnFamily.length() > 0);

                Object field_value = t.get(columnQualifier);

                if (field_value != null && ! (field_value.toString().isEmpty())) {

                        p.add(Bytes.toBytes(columnFamily), Bytes.toBytes(columnQualifier),
Bytes.toBytes(field_value.toString()));
                }
              }
              return p;
        }

        }


        public static class WebpageMapper extends
                Mapper<AvroKey<Webpage>, NullWritable, ImmutableBytesWritable, Put> {
                protected void map(AvroKey<Webpage> key, NullWritable value, Context
context)
                                throws IOException, InterruptedException {
                        Webpage w = key.datum();
                        ImmutableBytesWritable s = new
ImmutableBytesWritable(Bytes.toBytes(w.getDocId().toString()));

                        Put p = datumToPut(w);
                        if (!p.isEmpty()) {
                                context.write(s, datumToPut(w));

            } else {
                System.out.println("WARN: All fields are null for row with id " + w.getDocId() +
                                        ". Nothing to add.");
            }



                }

                private static Put datumToPut(Webpage w) throws IOException {
                        Put p = new Put(Bytes.toBytes(w.getDocId().toString()));
            for (Schema.Field field : w.getSchema().getFields()) {
                // skip row id
                if (field.name().equals("doc_id")) {
                        continue;
                }
                String columnFamily = field.doc();
                String columnQualifier = field.name();
                assert(columnFamily.length() > 0);

                Object field_value = w.get(columnQualifier);

                if (field_value != null && ! (field_value.toString().isEmpty())) {

                        p.add(Bytes.toBytes(columnFamily), Bytes.toBytes(columnQualifier),
Bytes.toBytes(field_value.toString()));
                }
```

```
				}
					return p;
			}
		}

		@Override
		public int run(String[] rawArgs) throws Exception {
			if (rawArgs.length != 3) {
				System.err.printf("Usage: %s [generic options] <input> <output>
<tableName>\n",
						getClass().getName());
				ToolRunner.printGenericCommandUsage(System.err);
				return -1;
			}

			TableName tn = null;
			String[] args = new GenericOptionsParser(rawArgs).getRemainingArgs();
			try {
				tn = TableName.valueOf(args[2]);
			} catch (IllegalArgumentException exception) {
				System.out.println("Data type must be either \"tweets\" or \"webpages\".
\"" + args[2] + "\" was found.");
				return -1;
			}
			tableName = args[2];

			Configuration config = HBaseConfiguration.create();
			config.set("mapreduce.task.timeout", "300000");
			config.set("hbase.zookeeper.quorum",
"node1.dlrl:2181,node2.dlrl:2181,node3.dlrl:2181");
			Job job = Job.getInstance(config, "hbase-bulk-load - " + args[0]);

			job.setJarByClass(HBaseBulkload.class);

			Path inPath = new Path(args[0]);
			Path outPath = new Path(args[1]);

			FileInputFormat.setInputPaths(job, inPath);
			FileOutputFormat.setOutputPath(job, outPath);
			outPath.getFileSystem(super.getConf()).delete(outPath, true);

			job.setInputFormatClass(AvroKeyInputFormat.class);
			job.setMapOutputKeyClass(ImmutableBytesWritable.class);
			job.setMapOutputValueClass(Put.class);
		switch(tn) {
			case test_tweets:
			case tweets:
			case bulk_tweets:
				job.setMapperClass(TweetMapper.class);
				AvroJob.setInputKeySchema(job, Tweet.getClassSchema());
				break;
			case test_webpages:
			case webpages:
			case bulk_webpages:
				job.setMapperClass(WebpageMapper.class);
				AvroJob.setInputKeySchema(job, Webpage.getClassSchema());
				break;
			default:
				System.out.println("Reached unexpected switch statement case.
Quitting.");
			return -1;
		}

		TableMapReduceUtil.addDependencyJars(job);
		HTable htable = new HTable(config, tableName);
		HFileOutputFormat.configureIncrementalLoad(job, htable);
		//TableMapReduceUtil.initTableReducerJob(tableName, null, job);


		return (job.waitForCompletion(true) ? 0 : 1);
	}

	public static void main(String[] args) throws Exception {
		int result = ToolRunner.run(new HBaseBulkload(), args);
		System.exit(result);
	}
}
```