Sifter: A Generalized, Efficient, and Scalable Big Data Corpus Generator

by

Sherwin Zhang Wu

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
September 19, 2014

Signature redacted

Certified by ....
Professor Samuel Madden
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by ....
Professor Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Sifter: A Generalized, Efficient, and Scalable Big Data Corpus Generator

by

Sherwin Zhang Wu

Submitted to the Department of Electrical Engineering and Computer Science
on September 19, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Big data has reached the point where the volume, velocity, and variety of data place significant limitations on the computer systems which process and analyze them. Working with very large data sets has becoming increasingly unweildly. Therefore, our goal was to create a system that can support efficient extraction of data subsets to a size that can be manipulated on a single machine. Sifter was developed as a big data corpus generator for scientists to generate these smaller datasets from an original larger one. Sifter's three-layer architecture allows for client users to easily create their own custom data corpus jobs, while allowing administrative users to easily integrate additional core data sets into Sifter. This thesis presents the implemented Sifter system deployed on an initial Twitter dataset. We further show how we added support for a secondary MIMIC medical dataset, as well as demonstrate the scalability of Sifter with very large datasets.

Thesis Supervisor: Professor Samuel Madden
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to give thanks to my family and friends who have supported me through the highs and lows of my MIT career. Without their unconditional support, I definitely would not have been able to get to where I am today.

I am also deeply grateful to Professor Samuel Madden for his excellent support as a thesis advisor. Without his guidance, generosity, and kindness, this thesis would not have been possible.

I would also like to thank Dr. Vijay Gadepally for his patience and willingness to help me throughout this project. His ability to put up with me was second to none, and his mentorship has been nothing short of extraordinary. Lastly, I would like to thank Dr. Jeremy Kepner for his high level guidance and for his involvement in the technologies involved in this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents the Sifter data corpus generation system. In this introductory chapter, we will explain the motivations for building a system like Sifter in the context of the current state of big data. Then we will provide a high level overview of the Sifter system before diving into the details.

1.1 Motivations for Sifter

An overarching and high-level vision for big data research is to ultimately allow scientists to process, analyze, and gain insights into any big data corpus without being limited by the computer systems that they use. However, there are many factors that are blocking the realization of this vision.

As technology becomes more and more widespread in our world today, the amount of data collected by computer systems is increasing at an unprecendented rate. This meteoric rise of big data has created technological challenges along three facets of data known as the 3 V's of big data — Volume, Velocity, and Variety [1] [2]. Big data *volume* has increased as larger raw amounts of data strains the storage, memory, and computational capacity of existing systems, resulting in increased reliance on cloud computing systems. Big data *velocity* has stressed the rate at which data can be ingested and meaningfully processed. Finally, big data *variety* places limitations on algorithms and tools that can address a large variety of input data.

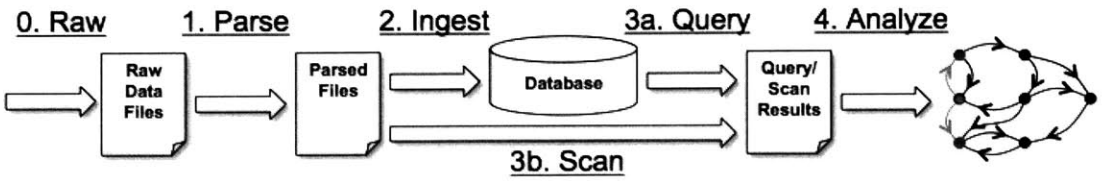


Figure 1-1: Big data pipeline

Because of these multifaceted challenges, researchers have realized that many system architectures are becoming size, weight, and power (SWaP) limited [3]. Large datasets spread across many distributed machines are difficult to move around, and lengthy processing times make it very difficult to make quick computations or interactions with the data. As a result, it has become much more convenient to work with data that fits into a single machine.

With these challenges in mind, research is being done on two fronts in regards to the general big data pipeline of reading, parsing, ingesting, querying, and analyzing data (Figure 1-1):

1. Improving the pipeline processes, such as finding ways to parse or ingest data faster and better.

2. Creating better data systems to address the SWaP challenges, such as finding innovative ways to create distributed database systems.

This brings us to the Sifter data corpus generation system, which we believe is a step forward for the first front — improving the processes in the big data pipeline. Particularly, Sifter is targeted towards optimizing the processes involved in the ingest stage (Step 2 in Figure 1-1). It solves the problem of SWaP limited architectures stumbling with very large datasets by allowing researchers to create their own custom smaller data corpuses from a larger original one.

This thesis presents several of our contributions to this effort through the Sifter system. In particular, we:

1. Architected, designed, and implemented the Sifter system.

2. Deployed the Sifter system on the MIT Lincoln Laboratory's E1 LLGrid supercomputing cluster [4], on top of a Twitter data corpus.

3. Extended the Sifter system by incorporating the MIMIC dataset as a secondary dataset.

4. Benchmarked the Sifter system to demonstrate its scalability and performance.

1.2 Overview of Sifter

Sifter is a system for processing existing big data corpuses and filtering out a smaller and higher-quality subset of data for further data processing. It relies on the Dyamic, Distributed, Dimensional Data Model (D4M) data format to store its data — this allows for extremely efficient processing and preparing of data corpuses. Sifter is built on a three-layer architecture in which different types of users will interact with different layers. It consists of a client layer, a server layer, and a data storage layer.

Normal client users of Sifter interact with only the top client layer. They are the users who have an interest in a large data corpus currently housed in Sifter, and they want to use Sifter to create a smaller custom corpus for their own research purposes. They interact with a web application interface built in Meteor.js, which provides them with a dynamic and responsive job submission form. It allows them to create a custom data corpus creation job, while giving them live feedback on data estimates of their data corpus as they interact with the form. After job submission, they can view the status of their job, as well as finally download the custom data corpus they made.

A smaller subset of administrative, or admin, users will interact with all three layers, mostly touching the server and data storage layers. These users want to integrate their own large data corpus into Sifter so that they themselves and other people can use Sifter to create custom data corpuses from the newly integrated data. These admin users can integrate their data through a two step process. They first process their data to be in the correct format, and then make incremental changes in

the client and server layer to support usage of the new data corpus.

With this overview, we will now dive into the rest of Sifter. For the rest of this thesis, we will first provide some background information on the technologies used by Sifter, as well as discuss past related works. Then we will show how users can interact with Sifter and use Sifter. From there, we will present the architecture for Sifter and discuss key architectural decisions, as well as discuss the generalizability of Sifter's data sources and show how a researcher could integrate new data corpuses into Sifter. Then we will present some performance benchmarks of the Sifter system and identify system bottlenecks. Finally, we discuss future directions of Sifter and what existing work can be done on the system.

Chapter 2

Background Information

In this section I will provide background information to lay the foundation for some of the core technologies used in the Sifter system. At the very heart of Sifter is the central data that is stored in it — the data that is being "sifted" from. Sifter requires that all this central data be in a particular data format — the Dynamic Distributed Dimensional Data Model (D4M) [5]. We chose the D4M data format because it gives us many advantages for Sifter's use cases. Here we will discuss the D4M format, and show how it provides these advantages to the Sifter system.

2.1 Associative Arrays

Sifter relies on the D4M data format for much of its job processing, but D4M itself is built on top of the notion of associative arrays [5].

2.1.1 Definition and Usage

Associative arrays are data structures that represent associations between multidimensional entities using keys and values. Each tuple stores "positional" multidimensional data keys, along with a value associated at that position. For example, in a two-dimensional associative array, every column and row is associated with a particular value. A trivial two-dimensional associative array might be:

	col1	col2
row1	1	2
row2	3	4

Table 2.1: Trivial Associative Array Example

where the values each correspond to a specific row and column:

```
A["row1","col1"] = 1
A["row1","col2"] = 2
A["row2","col1"] = 3
A["row2","col2"] = 4
```

Associative arrays are already used in many data processing and storage systems today. Spreadsheets, which perhaps might be the original and most widely used data structure, are essentially glorified associative arrays. Similarly, triple store databases such as Big Table, Dynamo, Cassandra, and HBase, use many of the concepts of associative arrays [5].

2.1.2 Composability

A useful property of associative arrays are that query operations carry nice composability properties, allowing for the ability to construct complex composable query operations [5]. The following operations all result in an output which is an associative array, allowing for nice chaining of these operations.

```
A('alice',:)  // alice row
A(:,'age') // age column
A('al*',:)  // rows that begin with al
A(1:3,:)  // first three rows
A == 47.0 // subarray with values 47.0
```

This composability of associative arrays is built upon the fact that many mathematical operations performed over associative arrays return associative arrays [5]. The following operations also all return associative arrays.

```
A + B     A - B     A & B     A | B     A * B
```

This composability proves to be very useful in data processing, especially when it is used in conjunction with the D4M data format. By allowing us to chain query and mathematical operations, associative arrays empower Sifter to perform complex query and filter operations on data corpuses very elegantly.

2.2 D4M Data Format

The D4M data format was developed by the MIT Lincoln Labs as a general purpose schema for the Accumulo database system. It was meant to be highly general, built upon the advantages of associative arrays (by storing data in associative arrays), and allowed for the fastest published ingest rates of data into a database system [1]. The D4M schema can also be used for many noSQL-style databases as well. Regardless, for Sifter we wanted to avoid the overhead of an entire database system and just wanted to interact with the raw data in main memory.

The MIT Lincoln Laboratory currently only has an implementation of D4M in Matlab [1]. It includes a library with their own custom associative array datatype, as well as processing tools and functions that would be useful for data stored in the D4M format. All data is stored in associative array data structures, and can be exported and loaded by Matlab into files on disk. In order to work with just the raw data, we decided let Sifter work with only those raw Matlab D4M files.

The D4M format was designed as a schema for use with the Accumula database system [1]. However, because Sifter does not use a deployed database system, we only use a subset of D4M. Regardless, we now explain D4M in full for comprehensiveness and complete understanding.

2.2.1 D4M Details

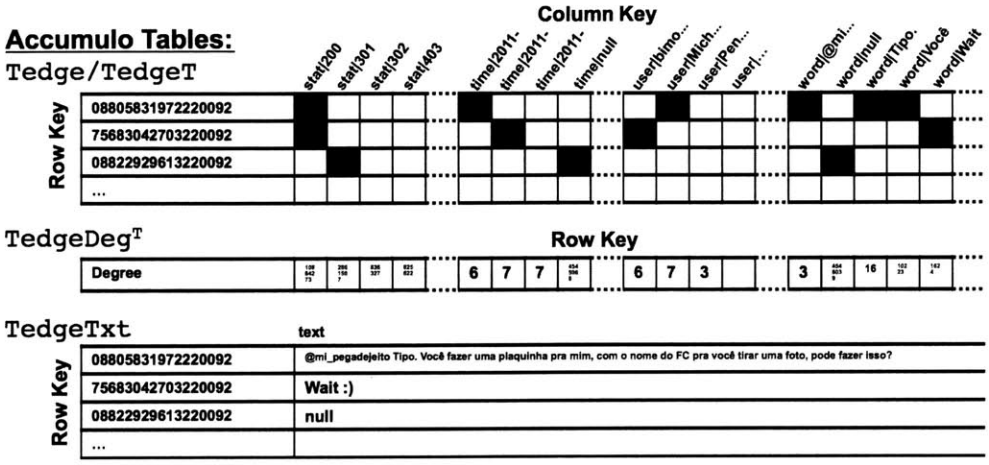


Figure 2-1: D4M Schema applied to Twitter data

D4M uses a four table schema in Accumulo to store data (Figure 2-1, adapted from [1]). As a walkthrough example, we can use a Twitter dataset consisting of a large corpus of tweets to look at each part of the D4M schema.

Tedge and TedgeT

The `Tedge` table contains all the semantic information of the raw dataset [1]. As displayed in Figure 2-1 for the Twitter dataset, each row represents a tweet itself, with each row key representing the tweet's tweet ID stored in a big endian format. The remainder of the tweet's attributes are stored as column keys. This means that each unique attribute value will be present as a column. As a result, it is common for the `Tedge` table to have significantly more columns than rows. As we discuss later, this representation has many empty cells (as a sparse matrix), which can be compressed away.

Notice that for the text of the tweet, it is actually split by words into 1-grams by the D4M schema and each unique 1-gram is stored as a column key. The values themselves can only have a value of 0 or 1, which is represented in the figure as a box colored in white or black, respectively. Thus, a tweet with the text "hello world" would generate the columns "word|hello" and "word|world", and have values of 1 for

them (or have values of 1 for those columns if they already exist).

In order to query efficiently on not only the row keys, but also the column keys, the schema stores both `Tedge` and its transpose, `TedgeT`. Therefore, `TedgeT` contains the same information as `Tedge` but is a transposed copy which is used for row-oriented databases. This effectively allows for indexing on both the rows and columns, and allows for fast lookups in both dimensions.

TedgeDeg

The `TedgeDeg` table contains the degree distribution of `Tedge` [1]. It essentially stores a histogram of the counts of non-zero values for each column. This information helps in planning queries by giving an estimate on size.

TedgeTxt

Lastly, the `TedgeTxt` table is an auxiliary table whose purpose is to allow us to reconstruct original data without too much difficulty [1]. In the case of the Twitter dataset, we broke up the text of a tweet into individual words for the column keys in `Tedge`. Thus, we use the `TedgeTxt` table to store the whole text with the row key. For this example it stores only the tweet text, but it can be used to store and recover any data attribute that was broken apart for querying purposes in `Tedge`.

Sifter doesn't use an actual Accumulo instance to store its core data because of the overhead involved with importing data into it and, more importantly, exporting data out of it. Therefore, we just stored the core `Tedge` and `TedgeT` table on disk for data processing.

2.2.2 Advantages of D4M

The D4M format was developed to be used in association with the Accumulo database system. As a result, there were many design decisions of D4M which optimizes it for usage with the Accumulo database system [1]. However, Sifter was meant to be

much more general purpose — to allow users to obtain data in a variety of forms not limited to a particular database system. Luckily, many of these features and properties of D4M greatly benefited its usage in Sifter and were not limited to just use with Accumulo. We describe these features as follows.

Column and Row Lookup

One of the most important features of D4M for usage in Sifter is that is allows for constant time lookup for any row or column stored in its format [1]. This essentially allows for constant time lookup and filtering on any attribute of a data object in the large data corpus. This is because each cell is one bit, so a given cell will be at a fixed, known position marked by its row and column.

A common example for the Twitter dataset might be to query a large data corpus for all tweets that contain the keyword "hello". With the D4M format, we could just look for all rows which have a value of 1 for the column "word|hello". This is very fast and efficient with the D4M format, and allows for very fast querying and filtering which Sifter takes full advantage of.

Traditional and Non-Traditional Database Support

D4M also has excellent support for a broad range of database types. For traditional databases, rows and columns intuitively translate into the D4M format. D4M has also been used without modification to handle cyber, bioinformatics, scientific citation, free text, and social media data [1]. NoSQL database or non-rigid JSON objects could easily be translated and stored in the D4M format.

This support of a broad range of data types is a key feature of D4M for us, as we wanted Sifter to be highly general purpose and able to support a large variety of datatypes. It would have been highly undesirable if D4M could only be used with a limited number of datatypes, but fortunately it was highly general.

Sparse Matrix Storage

Data stored in the D4M format tends to be highly sparse. In the example Twitter dataset, it was shown that the 16M x 30M storage associative array was $> 99.9999\%$ empty [1]. This means that the vast majority of the values in D4M associative arrays are null or non-existent. Storage of D4M in Matlab is conscious of this, and only stores non-empty columns for a row [1]. This way, the entire associative array structure is preserved, but only the minimal amount of space is used to store the entire structure.

Arbitrary Text Support

The D4M format also is able to support storing arbitrary byte strings [1], allowing it to store numeric data as well as multilingual data such as unicode. This makes the D4M format help support generalizability and doesn't limit storage on particular character encodings.

Parallelization

The D4M format can also support parallelization very easily, with multiple processes able to read and process data [1]. This is particularly useful for Sifter, because it must process massively large data corpuses, and parallelization would help alleviate some of the performance load.

Furthermore, because the D4M implementation was in Matlab, we would be able to take advantage of the pMatlab parallel Matlab toolbox. pMatlab is a library that was created and developed at the MIT Lincoln Laboratory for the purpose of parallel programming in Matlab [6]. It abstracts away the messy parts of parallel programming, such as keeping track of which processor the data is on, determining which processor the data needs to be delivered to or received from, and distributing computation to maximize computation on each processor. Instead, there is a very simple API to work with.

2.3 Related Work

The contributions that Sifter aims to have on big data systems are similar to that of batch processing systems. Similar batch processing results could be obtained from running Google MapReduce [7] jobs on large corpuses of data.

One could imagine that users who wanted a more manageable subset of their data could write a MapReduce program to process their large dataset. They would first load their dataset into a distributed file system that MapReduce can read from. They would then write their own map and reduce functions to process the data in their own way to get their custom subset of data. Finally, they would run their program on a large cluster of computers in a parallel and efficient manner.

However, there are some undesirable properties with this MapReduce approach. For one, if another user wanted to generate a slightly different subset of data from the same original large data corpus, they would have to read the previously written MapReduce code and slightly modify it for their own purpose, or write their own new MapReduce program entirely. Every subsequent user who wanted a slightly different data subset would have to go through these same steps.

Further, the actual data processing in MapReduce is not guaranteed to be efficient. Consider the problem where the original data corpus consists of data objects with a long text field attribute, and a user wants to generate a subset that consists of data objects that contain a particular keyword in the text field. More likely than not, the text will be stored in its most natural format — as a string of text. However, this would make filtering out data objects that contain a particular keyword to be extremely inefficient, as testing for presence of a keyword would involve scanning through the entire string of text.

Sifter presents what we believe is a better solution to this problem. Code is highly reusable — if an administrative user integrates a dataset into Sifter, other users can generate their own subsets from this data from a common interface without even writing any code. This absence of writing code makes Sifter accessible even to non-technically inclined users — they would be interacting with a simple web interface

instead of learning how to write MapReduce programs. Further, because Sifter stores data in the D4M data format, querying of data is extremely efficient. Searching for presence of a keyword in a string of text is a constant time operation.

Chapter 3

Sifter Interface

Sifter is a system that is meant to be easy and helpful to users. This section describes the client layer and how a user can interact with the interface. It also just shows what the client interface displays. Future users of Sifter should use this section to familiarize themselves with the system and client interface. For this section, we display screenshots of usage while creating a custom data corpus from the Twitter dataset (Figure 3-1). Any further explanation of how we calculate certain values displayed to the user, or how data is processed under the hood, is explained in Chapter 4.

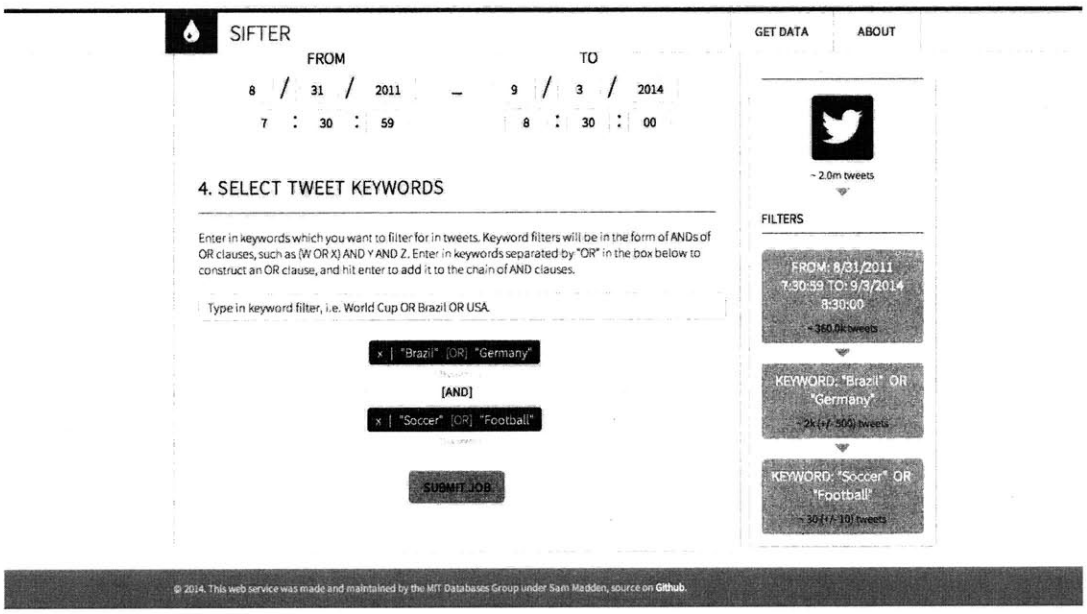


Figure 3-1: Screenshot of the job submission form in progress.

3.1 Data Corpus Submission Form

The heart of the Sifter client layer is a responsive web form which allows users to submit their own custom data corpus jobs.

The form renders itself dynamically — once a step is completed, the next step in the form will render itself. The form also offers dynamic error catching — the moment an input contains an error, the form will notify the user, and the moment the error is fixed, the form will remove the error. As a result, the form is inherently error proof, and only correctly formatted jobs will be submitted to the server layer.

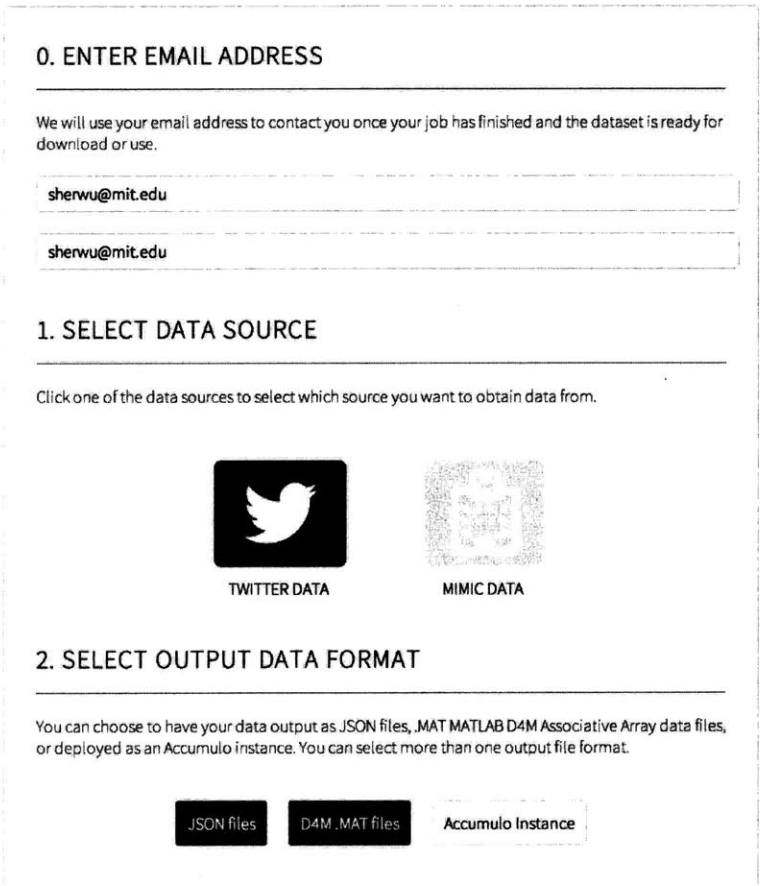


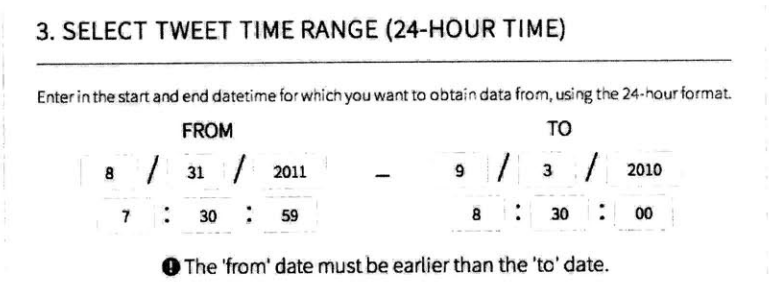
Figure 3-2: Screenshot of email input, data source selection, and data output format selection.

To start off the form, users input their email addresses, select an input data source to filter data from, and select an output data format (Figure 3-2).

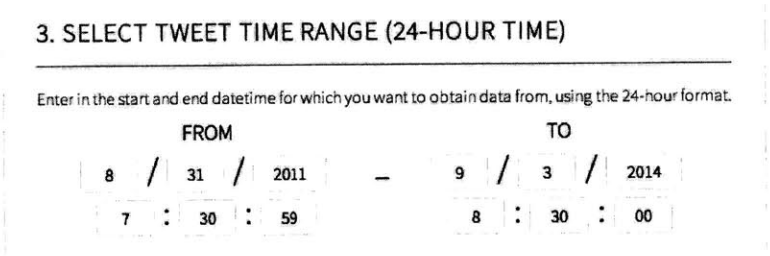
Users' email addresses are used to send job confirmation and status emails to the user after they submit a job. Currently, only two emails are sent: one when Sifter starts processing the job (giving the user a link to a job status page), and also when Sifter finishes processing the job (giving the user a link to view and download the new data corpus).

Users can also select which data source they want to select data from. Clicking on one large icon or the other selects the data source.

Sifter also offers a variety of data output options, and the form allows users to select one or more output formats. JSON files have become a standard for data gathered and used in web applications today, so users can have their data output as files consisting of rows of JSON objects. Alternatively, users can have their output data in the format of raw D4M files, which can be loaded by Matlab as associative arrays. Users can also choose to have their data ingested into an Accumulo database deployment for them to query and use to their choosing.



(a) Invalid date time input where the "from" date is later than the "to" date.



(b) Date time input with correct inputs.

Figure 3-3: Screenshots of date time input with dynamic error catching.

Users can then input the primary index filter, which for the Twitter dataset is the

creation time of the tweet (Figure 3-3). Users enter in the start and end dates for the Twitter data they want to obtain. Whenever the input is invalid, an error message appears, and when the input is corrected, the error message disappears.

The last step of the form is to input the secondary index filters, which for the Twitter dataset are keywords in the tweet content (Figure 3-4). The form allows users to enter in general logic queries for keywords in the form of ANDs of OR clauses. Users enter in each OR clause (where each keyword is separated with a capitalized "OR"), and entering them appends them to the running list of clauses. Users can also delete OR clauses that they have already entered as well.

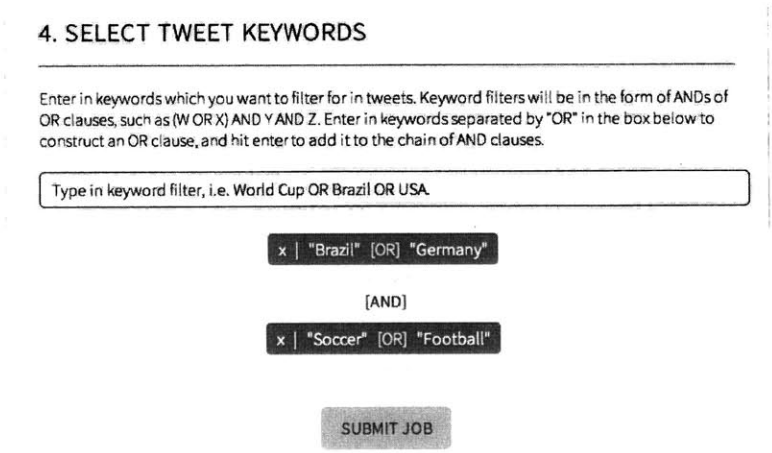


Figure 3-4: Screenshot of Twitter keyword input.

3.2 Data Corpus Preview Panel

Alongside the job submission form is a preview panel that updates responsively with users as they input and change data on the job submission form. As users choose data sources and add or remove filters, the side panel will dynamically update with the filters and with data estimates for the filters (Figure 3-5).

Once users choose a dataset, the side panel will reflect their selection and show them an estimate of how many data objects are currently in the entire data corpus they selected. In addition, when users select a primary index filter (time for the

Twitter dataset), the preview panel will also live update and provide a data estimate of the number of data objects that will pass through the primary index filter.

Similarly, when users add or remove secondary index filters (keywords for the Twitter dataset), the preview panel will also reflect these changes. The preview panel will also show them a sample data estimate with confidence intervals for each secondary index filter they add on, with estimates updating any time they remove or add a filter.

Figure 3-5: Screenshot of the Preview Panel.

3.3 After Job Submission

Right after users submit a job, they are given a job ID associated with the job they just submitted, as well as a link to view the status of the job. When Sifter begins processing their job, it will also send them an email with their job ID and a link to the status page.

The status page shows the three steps associated with a job:

1. Removing the job from the job queue.
2. Processing the job and filtering out the new data corpus.
3. Emailing the user about the finished job.

The second step typically takes by far the largest amount on time, and so the status page will show the user what percentage of the data has been processed as a better measurement of job progress. As each step is completed, the status page checks it off (Figure 3-6). Finally, the user can download their data corpus (Figure 3-7).

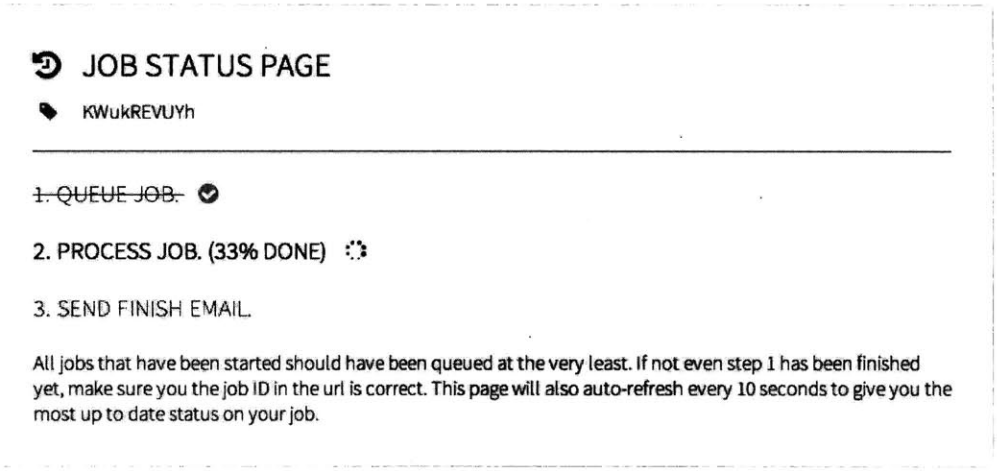


Figure 3-6: Screenshot of the job status page.

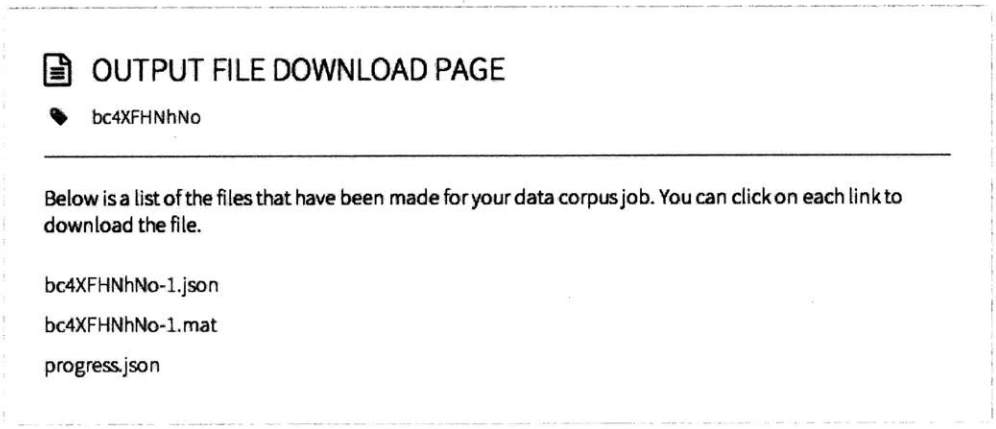


Figure 3-7: Screenshot of the file download page.

Chapter 4

Sifter Architecture

In this section we describe the architecture of the Sifter system. We first give a high level overview of the three-layer architecture. Then we will dive into the details of each of the three layers, describing the details of the client layer, server layer, and the data storage layer. We will describe how the three layers interact, as well as provide insights for key design decisions and trade-offs made with this architecture.

4.1 Architecture Overview

The architecture of Sifter is split into three layers — a client layer which users interact with, a server layer which processes the data corpuses and responds to client requests, and a data storage layer which hosts the actual data stored in the Sifter system (Figure 4-1). These three layers interact with each other to compose up the system architecture.

4.2 Client Web Application Layer

The client layer is a responsive Meteor.js web application that is exposed to the user. Its purpose is to interface with the user for the entire process involved with creating a custom data corpus job. This includes creating a job and entering details for a creation job, showing data estimations for a current job, providing statuses on

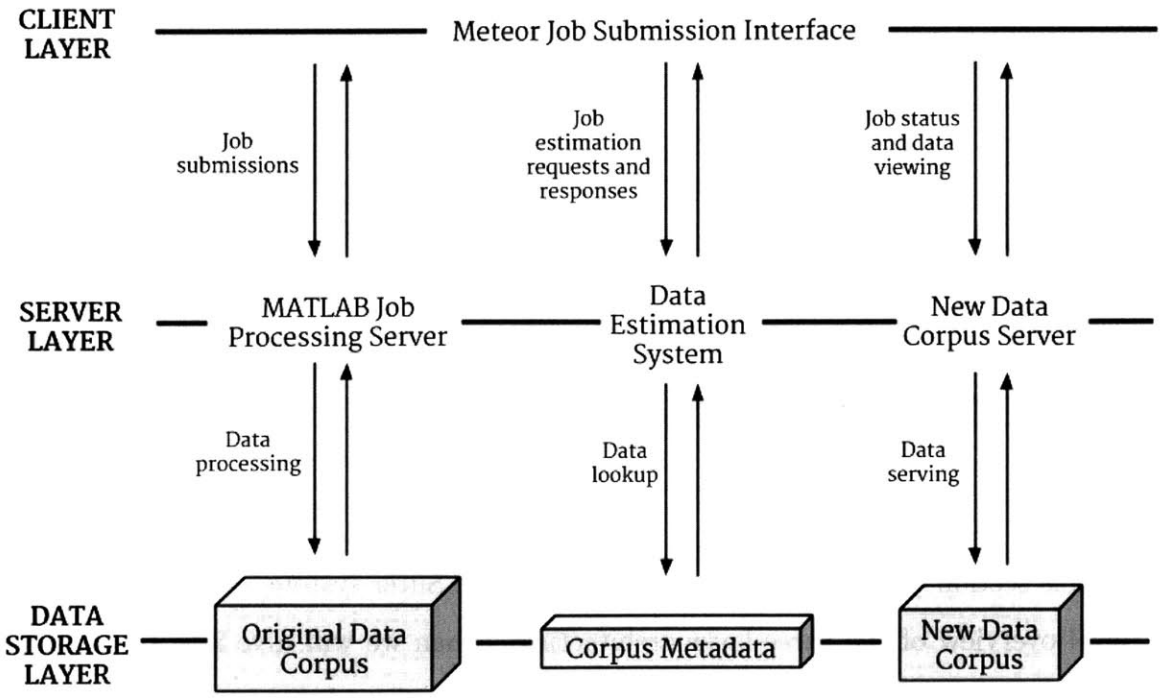


Figure 4-1: Sifter Architecture Overview

job processing, and allowing users to view and download data from jobs that have finished.

Because the other two layers (the server-side and data storage layers) are deployed on the beefy E1 supercomputing cluster, the client layer was designed to be lightweight and thin since the other two layers could do the bulk of the data processing work. Therefore, throughout the entire data corpus creation process, the client layer provides end-to-end support for a user as just an interface — it channels information it receives from the user to the server layer, and it displays data received from the server layer to the user.

The client layer provides a responsive web form for users to fill out a custom corpus creation job, choosing primary and secondary index filters for their dataset. The web form provides users with real time estimation of their dataset size by sending requests to a data estimation endpoint on the server layer.

When a user submits a job, the client layer takes the information for a corpus creation job and queues it for processing by the server layer. After queuing the job,

the client layer can query the server layer to show the user a live status page for their new submitted job. This status page tracks what steps have been completed and shows an updating percentage of data processing progress. Finally, when the job is done, the client layer allows the user to view their new data corpus and download it.

4.3 Server Backend Layer

The bulk of the work and processing power of Sifter lies in the server backend layer. This layer consists of three subcomponents which the client layer interfaces with, and each of which use data stored in the data storage layer.

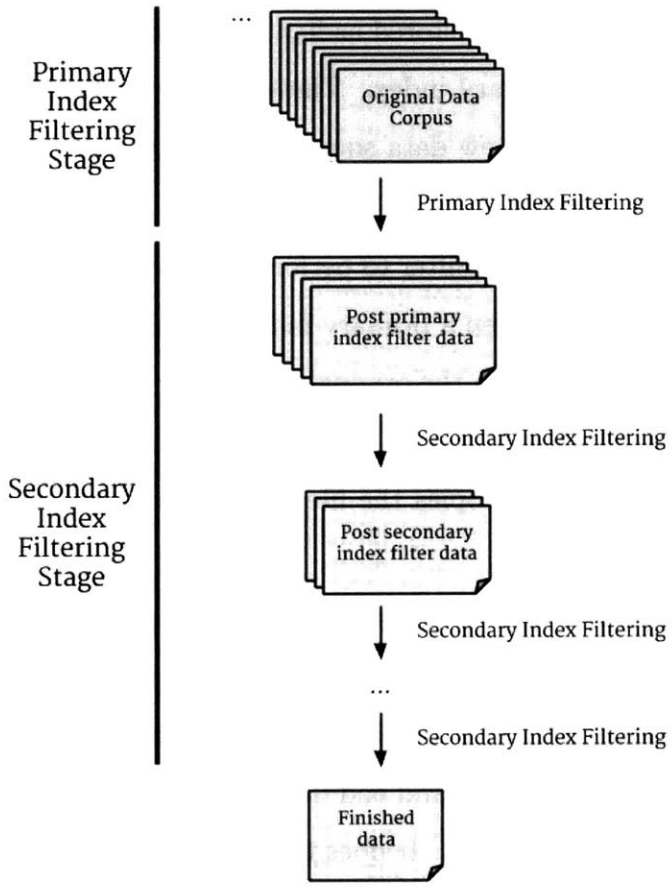


Figure 4-2: Sifter Data Filtering Overview

4.3.1 Matlab Job Processing Server

At the heart of the corpus creation process is the processing and filtering of data from the original data corpus. Sifter handles this filtering process by generalizing it into two components — a *primary index filtering process* and one or more *secondary index filtering processes* (Figure 4-2). To support generalizability of what data corpuses Sifter can support, users who integrate new data into Sifter can define the primary and secondary indices to target whatever data fields they want.

Primary Index

The primary index for a dataset represents the lowest granularity level of filtering, and is meant to create an initial rough filtering for the new data corpus creation process. Original data corpuses are stored as flat files sitting in a file system, and are grouped into files by their primary index. These files are named after the primary index, and users who integrate new data sources into Sifter must adhere to this rule for correct filtering of a dataset.

This naming scheme allows for Sifter to perform primary index filtering extremely quickly and trivially. Sifter is given a primary index filtering function for a particular dataset, and it processes all files in the original dataset by passing each file's filename to the primary index filtering function. If the function allows the filename to be passed through the filter, Sifter copies the file over to the destination folder for the next stage of processing.

For the Twitter dataset, the primary index is the time the tweet was created. Thus, the Twitter data is stored in files at the seconds granularity level — each file stores all tweets corresponding to a particular second datetime. The primary index filter function is given a target start and end datetime, and it allows all tweets within the time range to be filtered through. It does this by simply comparing the file name (the datetime for the tweets) to see if it is greater than the start datetime and smaller than the end datetime (Figure 4-3). If a file passes the primary index filter, it is copied to the destination folder. We copy files because the core dataset will likely be reused

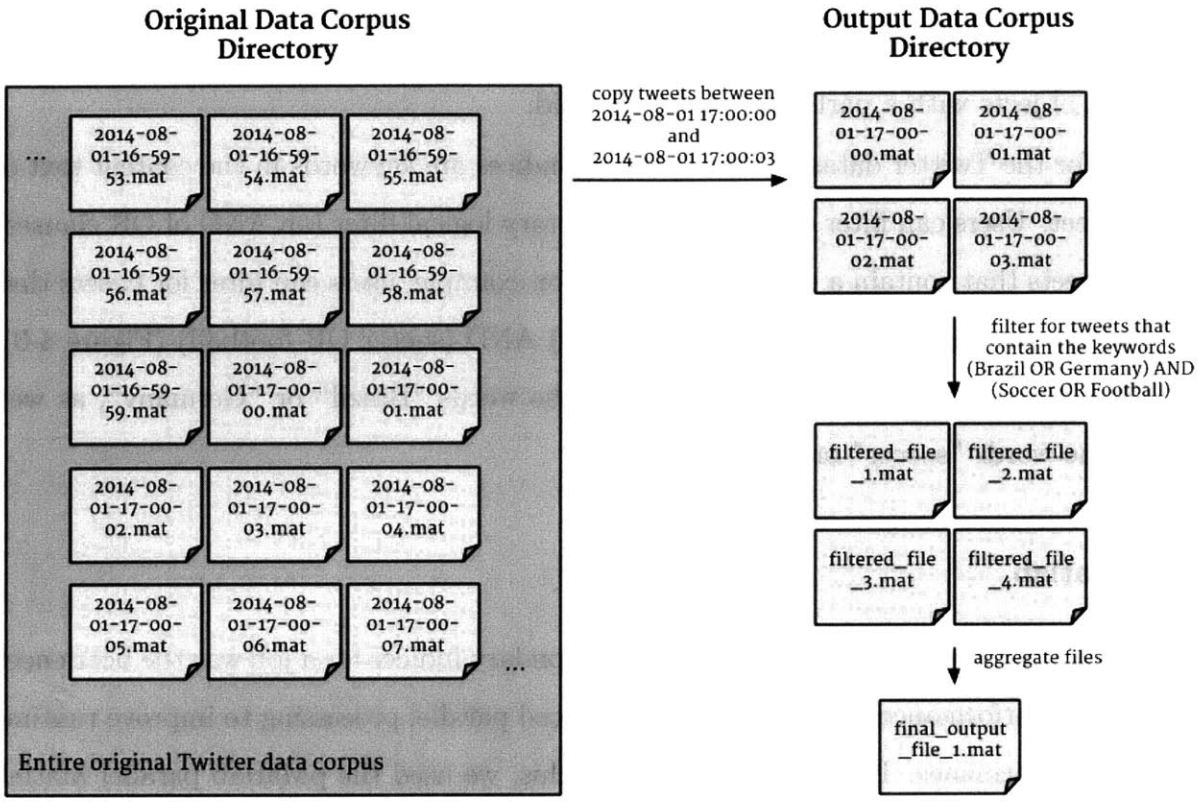


Figure 4-3: Twitter Data Filtering Example

in the future.

Secondary Index

The secondary index for a dataset represents the highest granularity level of filtering, and whittles down the dataset into the more precise final dataset we desire.

Secondary index filtering allows for filtering on any field in a dataset. Sifter is given a secondary index filtering function specific to a particular dataset, and applies this function to data passed into it to complete secondary index filtering. All output from the primary index filtering step (that was copied over into the destination folder) is fed into this secondary index function, and output of this function is written back into the destination folder.

Secondary index filtering involves loading datafiles into Matlab and processing them, and so takes considerably longer than the primary index filtering step. Sec-

ondary index functions typically load a datafile into memory as a D4M associative array, and then takes advantage of the features of the D4M data format to query for data objects with a particular matching field.

For the Twitter dataset, the secondary indices are keywords in the content text of a tweet. Users can filter data given an arbitrary logical filter (an AND of OR clauses) of tweets that contain a certain keyword. For example, users can filter for tweets that contain the keywords (Brazil OR Germany) AND (soccer OR football) (Figure 4-3). This looks for tweets that contain either the words "Brazil" or "Germany", as well as the words "soccer" or "football".

pMatlab

We quickly realized that filtering on the secondary indices for a job was the bottleneck for the performance of Sifter, so we introduced parallel processing to improve runtime and performance. In order to accomplish this, we used the pMatlab parallel Matlab Toolbox [6].

The bulk of the computational work of secondary index filtering involved serially processing the files output from the primary index filtering stage. Therefore, we used pMatlab to parallelize the processing of these files. With pMatlab, we partitioned the input files to the secondary index filtering stage between a predefined number of processes (p = 8 processes for the current deployment). Each process, in parallel, filters the data through the secondary filters, and then outputs the files. A final process then sweeps through the secondary index filter output files and aggregates the data into files of a predefined size (n = 10,000 data objects for the current deployment) (Figure 4-3). Chapter 7 further details the performance increases associated with increasing the number of processes in pMatlab.

4.3.2 Data Estimation System

One of Sifter's primary purposes is to allow users to create precise and high-quality datasets from a large and low-quality dataset. In order to help them do that, Sifter

shows users real-time data estimates as they add and remove filters from the web interface. This capability is powered by Sifter's Data Estimation System.

Data Metadata Generation

In order for users to still use the responsive web application, we needed data estimation to have an extremely low latency. To achieve this high performance, we offset the bulk of the estimation workload to a preprocessing stage — we preprocess the data files and write data estimation statistics to small metadata files. Sifter generates one metadata file for each original corpus data file.

The metadata files are JSON files which can contain any number of fields. For the purpose of this thesis project, the metadata files currently only contain a count of the number of data objects in a data file. This allows for data estimation on the size and number of objects a corpus creation job will make. Sifter runs a nightly cron job which recalculates the metadata files for each dataset; this allows for metadata to be used for dynamic streaming datasets.

Obtaining Data Estimates

The client layer issues three different data estimate requests to the server layer: entire dataset estimates, primary filter estimates, and secondary filter estimates. The client either wants a data estimate for an entire dataset, for a particular primary filter, or for a chain of one or more secondary filters. For entire dataset estimates, Sifter reads all metadata files of an entire dataset and aggregates values to return to the client layer. For size estimates, it simply adds up the data object counts from all metadata files. Primary filter estimates perform identically, aggregating values for data files that pass the currently queried primary filter.

However, secondary filter estimates are different since they involve more complex processing than simple aggregations. Because a low latency for these estimates was crucial for the purpose of Sifter, we perform estimates on *samples* of data for secondary filter estimates. We can think of the sample as a sample of a binary variable (whether or not a data object passes the secondary index filter), and so we perform statistical

analysis to generate confidence intervals on the sample data estimate.

For the sample estimate, Sifter randomly chooses a data file which passes the primary index filter as the sample. It loads it into memory, and runs the exact secondary index filter on this sample to get v_{sample}, the number of data objects in the sample that pass the secondary index filter. Sifter also knows n_{sample}, the size of the entire sample data it just loaded into memory, and so can calculate the proportion of data objects in the sample that pass the secondary index filter:

$$p_{sample} = v_{sample}/n_{sample}$$

This also gives us the standard error of the sample as

$$se_{sample} = \sqrt{\frac{p_{sample} * (1 - p_{sample})}{n_{sample}}}$$

Therefore, in order to get a reasonable 95% confidence interval, we use the following range for the proportion of data points that will pass the secondary index filter:

$$p_{sample} \pm (1.962 * se_{sample})$$

Sifter, from the primary index filter estimate mentioned above, knows the size n_{all} of the intermediate data corpus that comes out of the primary index filter. Thus, the final estimate that we show the user on the client layer is:

$$(n_{all} * p_{sample}) \pm (n_{all} * 1.962 * se_{sample})$$

4.3.3 New Data Corpus Server

The server layer also contains a server for processing and serving data after a job has been submitted. After any job is submitted, Sifter creates a custom directory for the output files and also tracks the processing progress of the job.

Job Status Tracking

Job status tracking is implemented by writing progress of a job to a JSON file whose contents are returned when the progress of a job is requested. The progress file is created the moment a job has started being processed and the empty output directory has been made. At each successive stage of processing, the server layer updates the progress file so that the user can read the most up to date information.

Finished Job Data Viewing

Once a job is finished, users can view the files for the new data corpus they created and download the files they created. The server layer simply serves the data in an output folder specific to the job the user is requesting data for.

4.4 Data Storage Hosting Layer

The data storage hosting layer contains and stores all the data behind Sifter. Because Sifter is deployed on the E1 supercomputing cluster, the data storage hosting layer sits on top of a Lustre File System. The file system abstracts away the distributed nature of the system, and looks like a normal Unix file system to systems which use it such as Sifter. This allows for the storage of the flat files that Sifter uses.

Sifter divides each dataset into three distinct data groups: the original data corpus, corpus metadata, and new data corpuses. The original data corpus contains the core data files from which jobs filter data from. The corpus metadata contains all the metadata that is used for data estimates. Finally, the new data corpuses store the output files from jobs that users have submitted. The new data corpuses are deleted when they are a week old, as users are expected to have downloaded their files by then.

Chapter 5

Integrating New Data into Sifter

Sifter was designed to be a generalized data corpus creator. Therefore, it is designed to make integration of new datasets easy and painless. In this section, we will describe the steps involved in integrating a new dataset by describing how we integrated the MIMIC II medical dataset into Sifter. We will detail the three steps involved in integrating data into Sifter: data preparation, modifications to the client layer, and modifications to the server layer — using the MIMIC dataset as an example.

5.1 Data preparation

In order for data to be used by Sifter, it must be in the D4M data format. Therefore, the entire data preparation process is just an effort to convert the data into the D4M data format. This is where the bulk of the integration work lies, as the modifications to the actual Sifter application are minor.

The MIMIC II dataset, or simply the MIMIC dataset, is a public and freely available data corpus that spans a diverse and extremely large cohort of intensive care unit (ICU) patients [8]. It is a highly diverse dataset with many different forms of data, including high temporal resolution data, electronic documentation, and monitoring waveforms — as well as more "traditional" data such as medical records and patient reports. The data can support a wide range of analytics, but we were only interested in the medical records data.

For the specific MIMIC dataset, its contents were stored in a MySQL database, fragmented over a large number of tables. In order to prepare the dataset that we wanted, we first performed a SQL join to create a temporary table with all the data that we wanted. Then, we would run a script to convert the contents of the MySQL table into D4M.

The data that we wanted from the MIMIC dataset was a comprehensive dataset of medications given to patients in emergency care over a period of time — we were interested in the medication name/dose, patient demographic information, and emergency care time information. The SQL join we ran aggregated all the data we wanted and put it into a temporary table (Figure 5-1).

```
INSERT INTO temp_table (medication, procedure_type, doses_per_24hrs,
   admit_dt, disch_dt, sex, dob, dod)
SELECT poe_order.medication, poe_order.procedure_type,
   poe_order.doses_per_24hrs, admissions.admit_dt, admissions.disch_dt,
   d_patients.sex, d_patients.dob, d_patients.dod
FROM poe_order
INNER JOIN admissions ON admissions.hadm_id = poe_order.hadm_id
INNER JOIN d_patients ON d_patients.subject_id = poe_order.subject_id
WHERE poe_order.medication IS NOT NULL;
```

Figure 5-1: SQL join code for preparation of MIMIC data

Our final step was to convert this temporary SQL table into a large D4M associative array. Because this seemed like a fairly general task, we wrote a Matlab script that could take any SQL table and — given a primary key — generate D4M data files that were partitioned on the primary key.

The script essentially just takes each data object in the SQL table (each row), and "explodes" each column attribute into the D4M associative array (Table 5.1). Each table was stored in a D4M file that were indexed by the ICU admission date for the visit pertaining to each medication treatment.

After the D4M files had been created, we just dropped them into Sifter's data storage layer and started integration on the client and server layers.

medication	procedure_type	sex	
Phenytoin	IV Piggyback	F	...
Metoprolol	Unit Dose	F	

(a) Abbreviated MIMIC MySQL Temporary Data Table

id	medication|Phenytoin	medication|Metoprolol	procedure_type|IV Piggyback	procedure_type|Unit Dose	sex|F
1	1		1		1
2		1		1	1

(b) Abbreviated D4M Associative Array Data Table

Table 5.1: MIMIC data MySQL to D4M Associative Array format

5.2 Client layer modifications

Modifications to the client layer only involve adding support to the job submission form for the new dataset. This involves adding the option to select the new dataset on the submission form and add form elements specific to the new dataset's respective primary and secondary index filters. In addition, the rendering template logic must be modified to selectively show the new dataset's form options, as well as update the data preview panel appropriately.

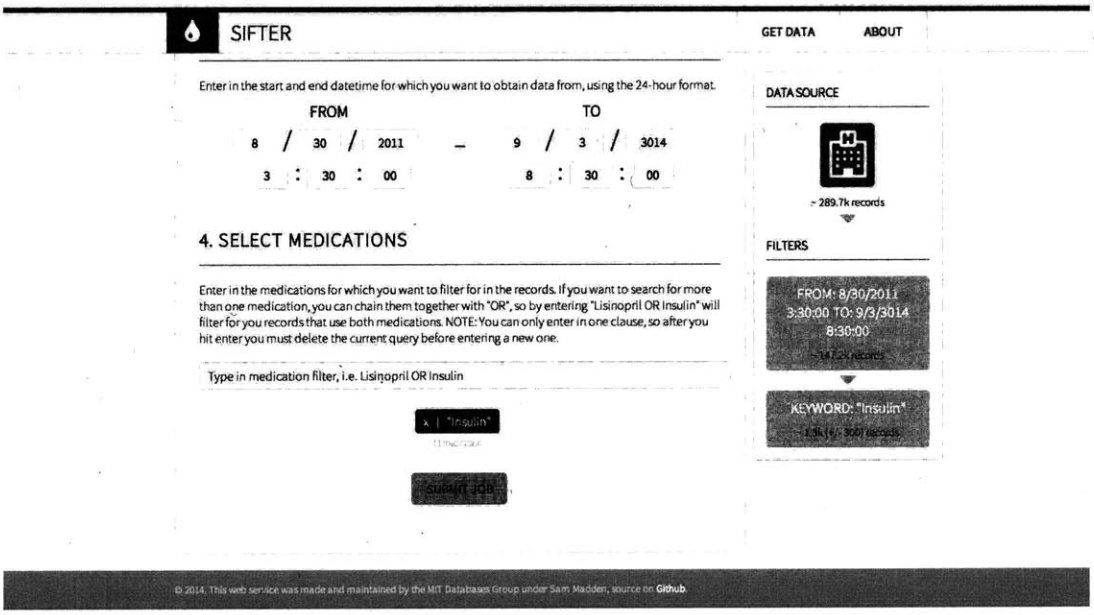


Figure 5-2: Screenshot of the client layer supporting MIMIC data.

Therefore, for integrating the MIMIC dataset into Sifter, we added the option to select the MIMIC dataset in the submission form, and modified the logic for rendering

the form elements and data preview panel (Figure 5-2). Because of the similarities to the Twitter data, the form elements were very similar to that of the Twitter data.

5.3 Server layer modifications

```
function b = twitter_primary_index(fileName, primaryIndex)
        startTweetTime = primaryIndex{1};
        endTweetTime = primaryIndex{2};
        b = strLT(startTweetTime, fileName) && strLT(fileName, endTweetTime);
end
```

(a) Twitter primary data filter

```
function b = mimic_primary_index(fileName, primaryIndex)
        startAdmitTime = primaryIndex{1};
        endAdmitTime = primaryIndex{2};
        b = strLT(startAdmitTime, fileName) && strLT(fileName, endAdmitTime);
end
```

(b) MIMIC primary data filter

Figure 5-3: Twitter vs. Mimic Primary Data Filter Functions

Modifications to the server layer involve changing the primary and secondary index filters in the job processing server for the new data set. The job processing server architecture was implemented in such a way that this would be simple to do. Filters are essentially "plug-and-play", as they can be swapped in and out with ease for each data processing job. Each filter is implemented as a Matlab function, and so to support a new datatype, a user just has to implement new primary and secondary index filter functions, and then add logic to support using the right filter when appropriate.

The primary index filter is implemented as a Matlab function which takes in the name of a file from the original data corpus and processes it. It returns a boolean deciding whether or not the file should pass through the primary index filter and be copied to the destination folder. For the MIMIC dataset, the primary index

filter incidentally happened to be the exact same as the Twitter dataset (Figure 5-3). However, it could be modified to perform any arbitrary logic specific to the new integrated dataset.

The secondary index filter is similarly implemented as a Matlab function. It takes it the actual D4M associative array object of an object to filter, and performs arbitrary filters on the data object, returning a filtered and smaller data object. For the Twitter dataset, it filtered by searching for matching keywords, while for the MIMIC dataset it filtered on the medication name (Figure 5-4).

```
function filtered = twitter_keyword_secondary_index(A, secondaryIndex)
  % iterate through each AND clause
  for k=1:length(secondaryIndex)
    % separate each keyword within the OR clauses
    orClause = secondaryIndex{k};
    results = Assoc({});

    % union all the results of the OR clauses
    for result = cellfun(@(keyword) A(:,['word|' keyword ','']), orClause,
        'UniformOutput', false);
      results = results + result{1};
    end

    A = A(Row(results),:);
  end

  filtered = A;
end
```

(a) Twitter secondary data filter

```
function filtered = mimic_medication_secondary_index(A, secondaryIndex)
  % iterate through each AND clause
  for k=1:length(secondaryIndex)
    % separate each keyword within the OR clauses
    orClause = secondaryIndex{k};
    results = Assoc({});

    % union all the results of the OR clauses
    for result = cellfun(@(keyword) A(:,['medication|' keyword ','']),
        orClause, 'UniformOutput', false);
      results = results + result{1};
    end

    A = A(Row(results),:);
  end

  filtered = A;
end
```

(b) MIMIC secondary data filter

Figure 5-4: Twitter vs. Mimic Secondary Data Filter Functions

Chapter 6

Performance Measurements

In this section, we present benchmarking and performance data for Sifter under a variety of conditions. We aim to show how different factors affect the performance of Sifter, but how even under a variety of conditions, the performance of Sifter is still reasonable and desirable.

Job Processing Server Bottlenecks

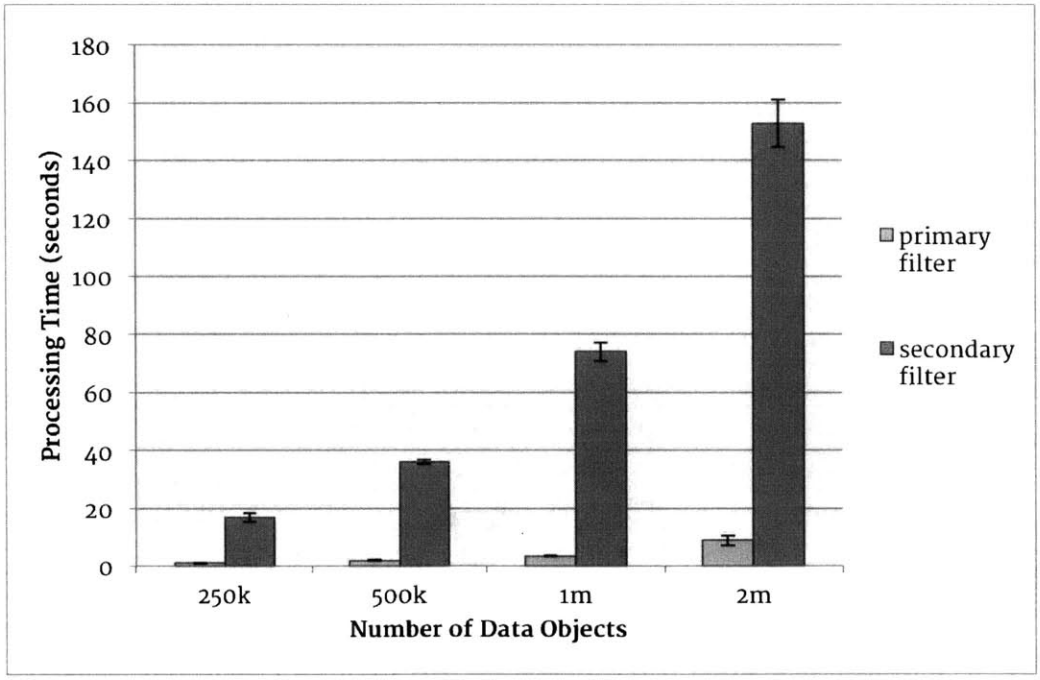


Figure 6-1: Performance bottleneck analysis of the primary and secondary filters.

We first wanted to carefully measure the processing times of each step that the job processing server goes through when filtering a job. This would allow us to identify bottlenecks in the system which we could subsequently alleviate. For these benchmarks, we used the Twitter data set and only used a single process (no parallelization). We varied the size of the dataset that would come out the primary index filter, and fixed the secondary index filter to only look for one keyword.

Figure 6-1 shows how very evident it was that the secondary index filter was the bottleneck in processing. The time it took for Sifter to finish the secondary index filter was over 10-fold that of the primary index filter. This was to be expected, since the core action in the primary index filter was just to copy files, while the secondary index filter had to load files into Matlab and filter them down.

These results led us to parallelizing the code involved in the secondary index filter to help alleviate the bottleneck there. The benchmarks performed in the rest of this chapter utilize parallelization of the secondary index filter with pMatlab.

Number of Processes in pMatlab

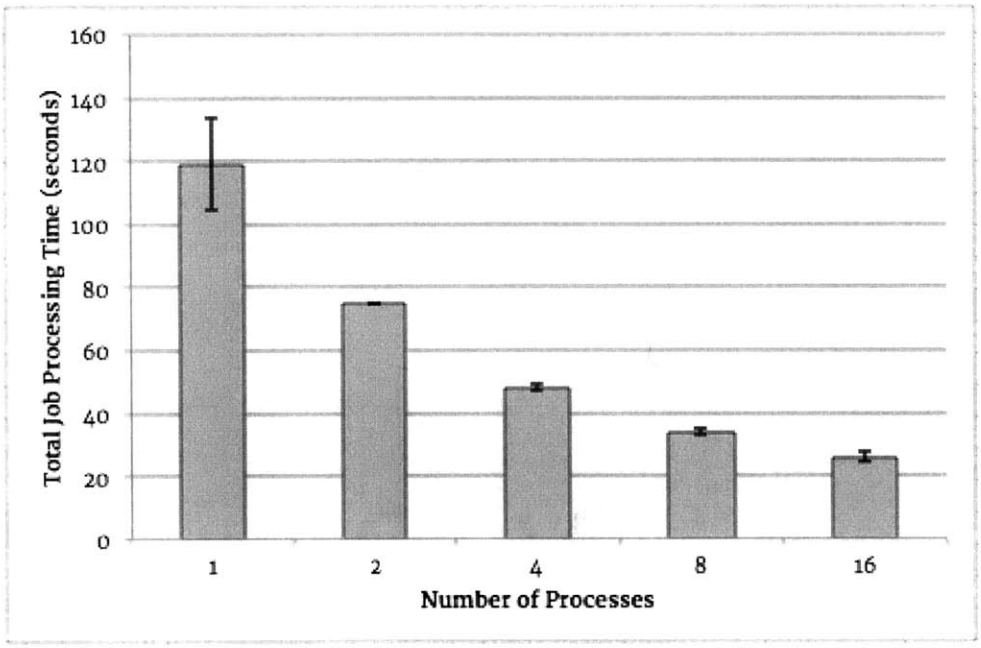


Figure 6-2: Performance gains from increasing the number of parallel processes for secondary index filtering

Because of the bottleneck that the secondary index filter causes when processing jobs and filtering out new custom data corpuses, Sifter uses pMatlab to parallelize this part of the code. By using many processes to filter data, we are able to help achieve reasonable performance while scaling up with the amount of data being processed.

Figure 6-2 demonstrates how parallelizing the secondary index filter processing can give an expected speedup as you increase the number of parallel processes that are handling the data. We benchmarked processing time for a corpus of 2 million data objects where the primary index filter allows them all to pass through. This means the secondary index filter had to process all 2 million data points. The query had only one term, and we increased the number of parallel processes that were used when doing secondary index filtering.

Because we only parallelized the secondary index filter processing, there was still some code that was not parallelized. The non-parallelized code had the same performance regardless of the number of processes we used. This explains why the runtimes speed up, but not exactly linearly, with the number of processes that we used.

Data Corpus Size

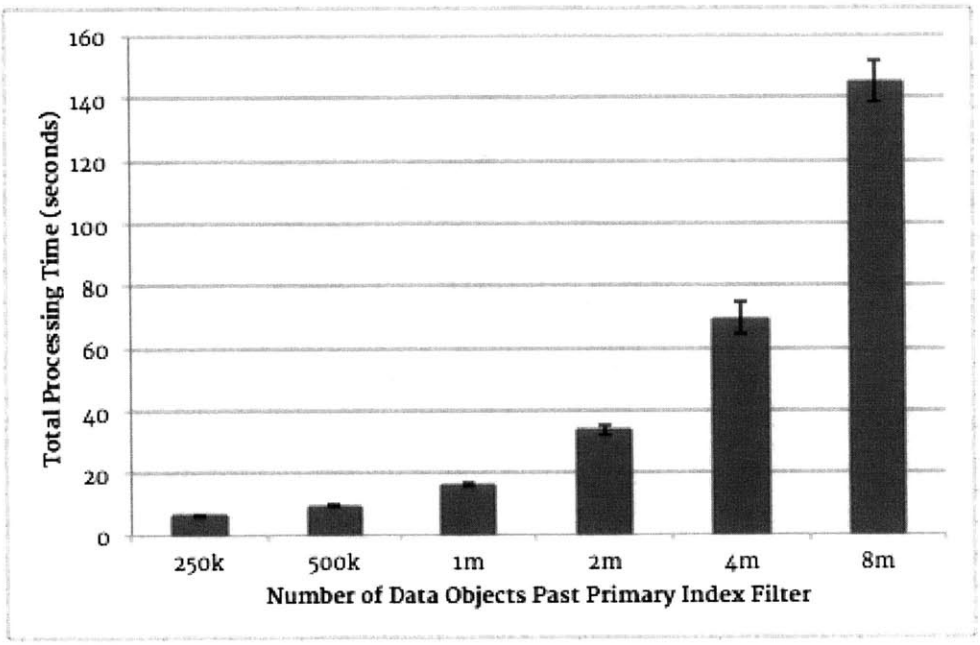


Figure 6-3: Performance with increasing corpus size

Clearly, the size of a data corpus that Sifter must process will affect the processing time for Sifter. We benchmarked the performance of Sifter on cuts of the Twitter dataset while using 8 processes and with a filter query with only one term, changing the number of tweets that pass the primary index filter each time. This made sure that the secondary index had to process that number of files for the job.

Figure 6-3 displays how the processing time goes up as the data corpus size increases. As expected, the data corpus size linearly affects the processing time — as the corpus size doubles, processing time approximately doubles as well. Note that even with 8 million data elements, processing still only takes $\sim$ 145 seconds, which we believe is still reasonable given that the data corpus generation process is asynchronous.

Filter Query Complexity

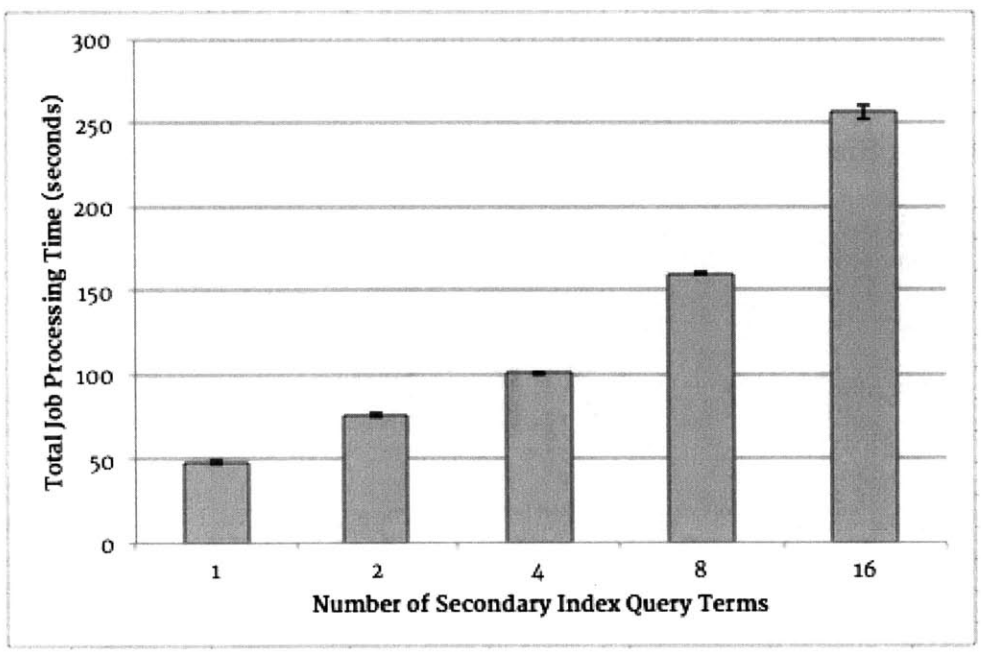


Figure 6-4: Performance with secondary index queries of increasing complexity

The complexity of a filter query for secondary index filters will also affect the performance of Sifter. Intuitively, the more complex a secondary index filter is, the more performance will suffer. We benchmarked the processing time it took to process

a corpus of 2 million tweets where all tweets pass through the primary filter, using 8 processes for the secondary index filter.

Figure 6-4 shows the increase in processing time when the "complexity" of a query increases. In the graph, we use the number of *terms* in a query to represent the "complexity" of the query. For the Twitter dataset, each *term* is just an element in either an AND or OR clause on the keywords. For example, the query ("Germany" OR "Brazil) AND ("Soccer") would have three terms — "Germany", "Brazil", and "Soccer". The figure shows how the processing time increases linearly with the number of terms, but is still reasonable even with 16 terms ($\sim$ 260 seconds), especially since corpus creation is asynchronous.

We varied the number of terms in the secondary index filter by adding on terms to one large OR clause (i.e. the four term experiment had a secondary index filter query of four terms OR'ed together — A OR B OR C OR D). We did this because the resulting runtime would be an upper bound on for all processing of that number of terms. If any of the terms in the OR clause were added as an AND (i.e. (A OR B OR C) AND (D)), this would be strictly faster to process. This is because the secondary index filter would interpret this first as a 3 term filter (A OR B OR C), and it would pass the result to the second 1 term filter (D). The filter that looks for D is processing a smaller subset of the data that had passed through the first secondary index filter, so the runtime would be faster.

Data Estimation Latency Times

The data estimation feature for Sifter is meant to give the user realtime feedback on the data corpus they are trying to create. Using the architecture detailed in Chapter 4, we were able to achieve very fast latency times for data estimations. As mentioned before, data estimation is dependent on the number of files for a dataset, and not actually on the number of data objects.

Figure 6.1 shows the latencies when performing data estimation on an entire data corpus. This estimation is shown when a user selects a data source, and aggregates counts from metadata files for every data file in a data corpus. Data estimations

for primary filters will look at strictly fewer metadata files, and so these latencies represent upper bounds for data estimation.

	Number of Data Files	**Estimation Latency**
Twitter	202	149ms
MIMIC	4065	2258ms

Table 6.1: Data Estimation Latency Times on an Entire Data Corpus

Chapter 7

Future Work

This section explores the possible future directions of Sifter. While we believe the Sifter system is very useful as it stands, there are many possible future areas of work which we hope to accomplish with Sifter.

One future direction for Sifter is to integrate it with a live, streaming dataset. Currently, the Twitter and MIMIC datasets are static — they have a fixed size and Sifter filters data from the static data corpus. A streaming, constantly increasing dataset would bring additional challenges in scalability and maintainability for the Sifter system, but would greatly increase the capabilities of Sifter. Many big data corpuses nowadays are actually streams of data that are constantly being collected and processed, so being able to support them would be a great next step for Sifter. A good candidate for a streaming dataset would be the original Twitter dataset, as research has even been done already on streaming Twitter datasets [9]. We are already in the process of creating a live Twitter pipeline that can feed data from the official Twitter firehose, and do not expect support of the Twitter firehose pipeline with Sifter to take much work.

We could also work on making the data integration portion of Sifter more streamlined for the administrative user. We could generalize the client layer web interface some more, to the point where a single interface could be dropped on top of a variety of datasets. Currently, there needs to be some modifications to the interface to support each additional dataset. Further, while we currently can import SQL data

from a single table that had been prepared, we could also modify Sifter to support importing data from an entire SQL database. This way, we could integrate arbitrary datasets into Sifter with even more ease.

Another addition to the Sifter system could be to just integrate new datasets to the system. As detailed in Chapter 5, the integration process for new datasets is fairly simple and straightforward. Adding more datasets to Sifter would increase how useful it is, as well as test the generalizability of the system.

Further work can also be done on helping users better understand and analyze the dataset they are trying to create from Sifter. One such example would be to provide analytics on the dataset as users fill out the job submission form. This could be an extension of the data estimates feature which currently gives users an estimate on the size of the corpus they are creating. Potential candidates that we have explored for analytics are dimensional data analysis, common-method variance of data, or other analytics on the potential dataset so users can glimpse into some metrics on the corpus they are about to create [2].

Chapter 8

Conclusion

This thesis set out to ultimately help allow scientists to use any big data corpus without being limited by the computer systems they use. We wanted to do this by improving the ingest stage of a general big data pipeline. In doing so, we accomplished four things:

1. Architected, designed, and implemented the Sifter system. (Chapter 4)

2. Deployed the Sifter system on the E1 supercomputing cluster, on top of a Twitter data corpus. (Chapter 4)

3. Extended the Sifter system by incorporating the MIMIC dataset as a secondary dataset. (Chapter 5)

4. Benchmarked the Sifter system to demonstrate its scalability and performance. (Chapter 6)

We described the three-layer architecture of Sifter, as well as provided insight into our key architectural design decisions. We showed how a client user would interact with Sifter, as well as how an administrative user would be able to easily integrate his or her own dataset into Sifter for use. We also showed benchmarking metrics to demonstrate that Sifter can scale up and handle very large datasets while maintaining reasonable performance.

Using the Sifter system, researchers can create their own high quality data corpus subset from an original larger dataset. They can also integrate their own custom datasets for their own use or for others. With Sifter, we believe that researchers can be empowered to process and analyze larger and larger datasets, pushing big data research forward.

Bibliography

[1] J. Kepner, W. Arcand, B. Bergeron, *et al.*, "D4M 2.0 schema: A general purpose high performance schema for the Accumulo database," *IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[2] V. Gadepally and J. Kepner, "Big Data Dimensional Analysis," *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[3] H. River and T. Flanagan, "Superior performance at breakthrough size, weight and power: Avionic and radar systems soar with TI's multicore DSPs." `http://www.ti.com/lit/wp/spry196/spry196.pdf`, 2012.

[4] A. Reuther, W. Arcand, C. Byun, *et al.*, "DR&E LLGrid Portal: Interactive Supercomputing for DoD," *IEEE High Performance Extreme Computing Conference (HPEC)*, 2010.

[5] J. Kepner, C. Anderson, W. Arcand, *et al.*, "Dynamic Distributed Dimensional Data Model (D4M) Database and Computation System," *ICASSP (International Conference on Accoustics, Speech, and Signal Processing)*, 2012.

[6] H. Kim, J. Mullen, and J. Kepner, "Introduction to Parallel Programming and pMatlab v2.0." `http://www.ll.mit.edu/mission/cybersec/softwaretools/pmatlab/pMatlab_intro.pdf`, 2008.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI*, 2004.

[8] G. Clifford, D. Scott, and M. Villarroel, "User Guide and Documentation for the MIMIC II Database." `https://mimic.physionet.org/UserGuide/UserGuide.pdf`, 2002.

[9] N. Sundaram, A. Turmukhametova, N. Satish, *et al.*, "Streaming Similarity Search over one Billion Tweets using Parallel Locality Sensitive Hashing," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, 2013.