Transforming and Caching Entries from Massive, Online Databases using Apache Spark and MongoDB

Mark E. McDermott

Volgenau School of Engineering
George Mason University
Fairfax, VA 22030
Email: mmcderm1@gmu.edu

Jose A. Velasquez-Principe

Volgenau School of Engineering
George Mason University
Fairfax, VA 22030
Email: jvelasq@gmu.edu

***Abstract*—Online databases with user generated content are increasingly common. If the number of users of such a database is large then the amount of content can quickly grow to be very large. This means that it is impractical and expensive to allow users to access the database in its entirety. To work around this many sites provide feature-rich search and query APIs that allow users to download only the portions of the database that they require. After downloading the data the users then processes it using their local computing resources. A very common operation to be performed by users is to perform preprocessing and cleaning of the data since, due the distributed sources of the data, it is likely to be of varying quality. This represents duplication of work that could be captured back into the public database. We present an architecture based on Apache Spark for performing this cleanup near the source of the data. We also present the creation of a cache of cleaned data so that other users can benefit from the work and analysis already performed. We demonstrate its effectiveness using a real-world example: We perform spectral filtering of recordings from the Xeno Canto database of bird-songs and then cache the filtered results.**

I. Introduction

One very interesting development to come from the cloud computing and data science explosion is crowd sourced data bases. These databases can grow very large as the work it takes to curate the databases is distributed around the globe. This distributed nature presents some challenges though.

One issue is that quality of individual entries in a database can vary since no single source or curator was involved in generating the entries. Some data providers, such as Kaggle [1], solve this by allowing users to rate the quality of datasets. However this approach, while it does provide useful information to potential users, does not do anything to improve quality. A dataset uploaded to a service such as Kaggle can only be updated by the user that uploaded it so improving the quality of a dataset after it is created can only be done by the original user. This reduces the benefits of distributed data curation.

Another challenge is that the meta data associated with particular data points can vary greatly based on the individual contributor's capacity to gather the data. This could be solved by careful curation of the submissions to ensure that each data point is properly labeled with metadata but this again reduces the benefits of crowd sourcing data collection.

We found a particular example on Kaggle that highlights this issue and inspired this project. A user of the very large, user generated database of bird call recordings - Xeno Cano [2] - uploaded a carefully curated subset of the Xeno Canto library to Kaggle so that it was ready for data science workflows [3]. This represents significant work that would have to duplicated in the future since the Xeno Canto database continues to grow as user's submit new recordings. The recordings are of many different lengths and have different levels of background noise, thus necessitating the cleanup step prior to use for data science.

In this paper we present an architecture that allows for crowd sourcing the curation efforts by caching cleaned up data points using a database based on flexible schema. This allows user's to perform the necessary data manipulation efforts and save the results for other's to use without requiring updating the source database's schema. We pair this cache with Apache Spark to allow for distributed processing of user's requests.

II. Technology Stack

Figure 1 shows the technologies we integrated for this project, each of which is introduced in this section. In the next section we show in detail how these technologies can be used together to create a cache of the Xeno Canto database.

A. MongoDB & MongoDB Compass

MongoDB is a cross-platform program that allow users to create databases for very large data and is used in an abundance of modern applications. MongoDB differs from traditional SQL databases in that data is stored as JSON documents with a flexible schema. This is crucial in applications where data comes from diverse sources and may be of differing quality and completeness. Because each entry in MongoDB is stored as a JSON document MongoDB pairs naturaly with internet APIs where JSON is the defacto format for transfering

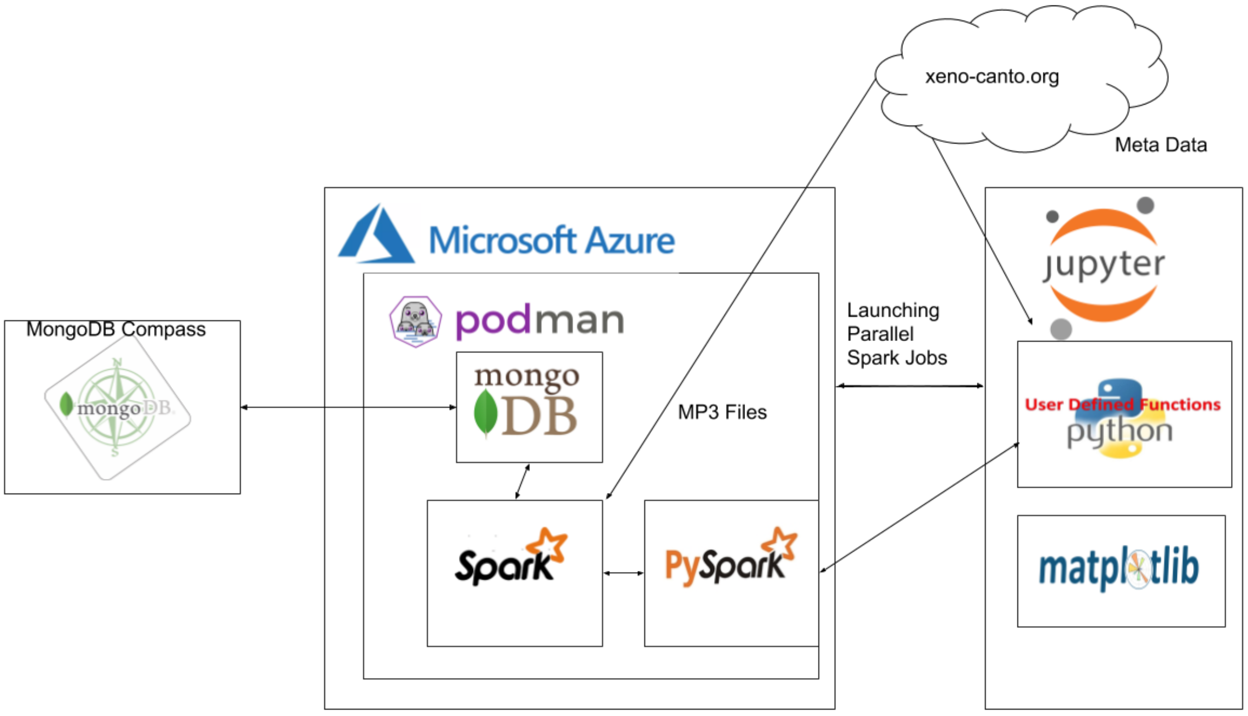


Fig. 1. Technology stack used in this project.

data. MongoDB can handle very large files with millions of records of data with ease [4].

MongoDB Compass is a freely available desktop application for Linux, Mac, and Microsoft Windows that allows viewing and editing of remote MongoDB databases. This tool also allows for engineers to view read/write performance statistics to help them resolve performance issues [5].

B. Apache Spark

Apache Spark is a data processing framework that allows users to perform operations on data sets that are very large. Apache Spark is a key application that allows big data and machine learning to be incorporated in a user-friendly application. Spark provides a framework whereby operations on very large data sets can be parallelized and distributed across a cluster of compute resources. Often compared to the earlier technology - Hadoop MapReduce - Spark is 100 times faster for some workloads.

Spark has interfaces in in many popular programming languages. Of particular interest for us is Python which is widely used in the scientific computing community for signal processing and machine learning. Apache Spark allows for user defined functions which enables the whole range of Python based scientific computing libraries to be parallelized on the Spark cluster [6].

C. Python

Python is one the fastest growing programming language [7]. Though considered to be a general purpose language, it has been adopted by the scientific computing community because of the availablity of libraries such as `numpy`, `scipy`, and `matplotlib`. It has also positioned itself as the leader in machine learning applications through the availabilty of high quality machine learning libraries such as `tensorflow`, and many others.

Python allows for seamless integration of all technologies to be used in this project since each of the technologies we selected have high quality Python APIs.

D. Jupyter Notebooks

Jupyter Notebooks is a user-friendly open source application that operates as a GUI or ”Notebook App” that allows users to write code similar to a text-editor but results of computation are presented inline with the code [8]. The Juypter Notebook allows us to demonstrate step by step how our architecture is to be used and incorporated for a particular problem domain.

E. OCI Containers

The Open Container Initiative (OCI) provides standards and formats for containerized computing technologies to ensure compatility of a wide range of products - both commercial and open source [9]. We used the container runtime system Podman [10] to build and integrate the technologies mentioned above. Containers are a powerful way to automate installation and configuration of software and help unlock automated deployment of cloud services [11].

In this project we use a container definition to control how all of the components are installed and configured to talk together. The container definition allows for managing each configuration change in a source control system so that we can track changes to configuration over time.

The container is also easily transfered to virtual machines running in the cloud so that efforts to configure and integrate the technologies don't have to be duplicated each time a cloud instance is brought online.

F. *Microsoft Azure*

Microsoft Azure is a subscription based application that allows users to create virtual machines (VM). VMs hosted on Microsoft's cloud infrastructure allow running very large and intensive applications that scale horizontally as user's needs increase or decrease. Azure provides tools for managing both large and small clusters of virtual machines across the cloud [12].

III. Architecture

The main purpose of caching previous results is to reduce the amount of repeated work. Similarly, we also want to reduce the amount of wasted network traffic. This is especially important if the source database and or end users are limited in their bandwidth. To do this we present and architecture that keeps the the cached data (using MongoDB) and the computation on that data using (using Spark) as close to each other as possible.

User's of the architecture can manipulate and cache results as necessary but keeping the data on the remote, cloud-based infrastructure until the very final step when they retrieve results for viewing on their local resources. To acomplish this it is necessary that only metadata be transfered to the user's local environment (as can be seen in Figure 1). The data itself which is assumed to be much larger than the metadata is only ever transferred from the source database into the remote cluster. This is beneficial if we assume that the network capacity between the source database and remote cluster is much higher than the network bandwidth available to the user's local environment.

A typical transaction takes place as follows:

1) The user retrieves some metadata from the source database into their Jupyter Notebook according to the their desired parameters. The user has a direct connection to the MongoDB cache which allows them to determine if there are results already available that match the operation they wish to perform and the metadata they retrieved. The dynamic nature of MongoDB schema makes it a perfect technology for performing open ended queries and comparisons such as this.
2) Any results that could not be found in the cache represent new operations that need to be performed by Apache Spark. The user creates a processing request that includes the metadata associated with the data to be processed and the instructions for how to perform the processing. This processing request is sent to Apache Spark using a Spark DataFrame and a user defined function (UDF).
3) For each item of metadata Apache Spark retrieves the actual data from the source database and then runs the requested processing on that data using the UDF provided by the user. Each item is retrieved and processed in parallel.
4) Apache Spark saves the results of these computations into the MongoDB cache.
5) Now all items from the original query are available in the cache so the user may retrieve them into their local environment (an expensive operation) or they may run further operations on them by providing other UDFs to Apache Spark.

It is worth reiterating that step 3 is a huge savings in time since we have assumed that the network bandwidth between the user's environment and the source database is much less than the bandwidth between the Spark cluster and the source database. The ability to apply further processing steps also represents savings since the user does not need to retrieve results from the cluster into their local environment until the final step.

All of the steps above are seamlessly integrated through the use of an OCI Container which bundles preconfigured installations of MongoDB, Apache Spark, Python, Jupyter Notebooks, and all supporting libraries. This container is deployed to a VM allocated using Microsoft Azure so that heavy processing and network bandwidth requirements can be spread across the cloud. Additionally the container may be run on local resources for testing and development purposes thus reducing the chance of configuration error when moving from a development environment to a production environment.

The current implementation of our container does not support multi-node Spark and MongoDB clusters. However as that capability is built into Spark and MongoDB such an extension to this work would be supported.

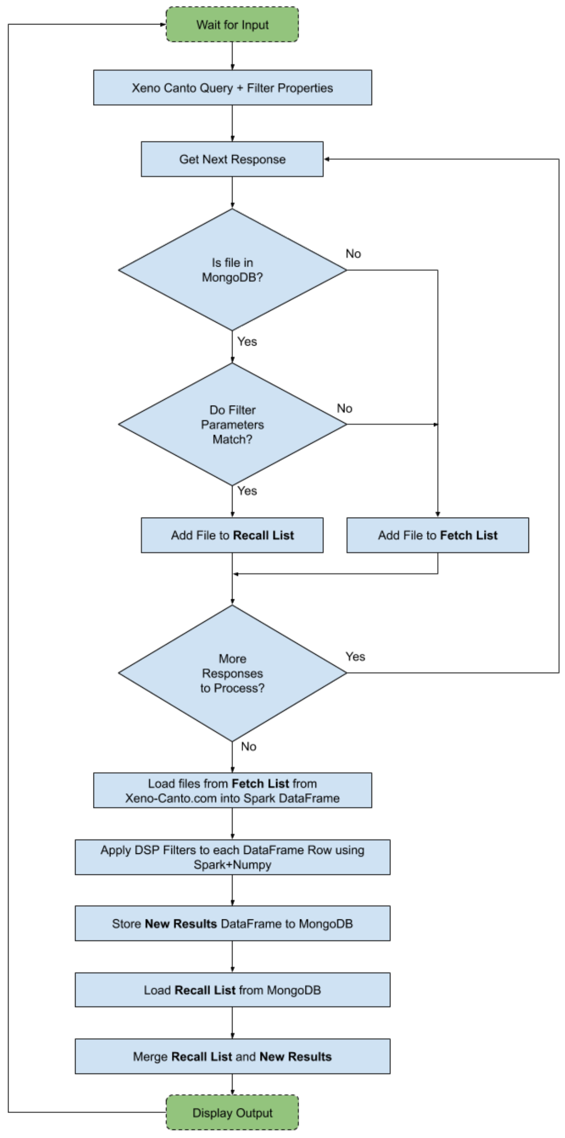
IV. Example Use Case

To demonstrate and prove out the proposed architecture we developed an application[1]to cache and query the massive, crowd sourced database of bird call recordings - *xeno-canto*. Figure 2 shows the logical flow of our test application. We will describe this flow in detail below. We chose bandpass filtering as a demonstration of the kinds of operations that could be performed and cached in the database. The flexible schema of MongoDB allows for any kind of operation to be performed provided that the parameters of the operation are in a format that can be stored to a JSON document.

A. *Caching the Xeno Canto Bird Call Database*

The *xeno-canto* database has over 500,000 recordings from all around the world, generated by more than 6000 contributors, and representing close to 9000 hours of recordings [2]. *Xeno-canto* provides a rich API for querying the database which we used to populate the MongoDB database. These

[1]Our application is available at https://github.com/MarkOfLark/gmu_ece_499_590_project under the MIT License [13] and subject to the George Mason University Honor Code [14].



Wait for Input
Xeno Canto Query + Filter Properties
Get Next Response
Is file in MongoDB?
No
Yes
Do Filter Parameters Match?
No
Yes
Add File to Recall List
Add File to Fetch List
More Responses to Process?
Yes
No
Load files from Fetch List from Xeno-Canto.com into Spark DataFrame
Apply DSP Filters to each DataFrame Row using Spark+Numpy
Store New Results DataFrame to MongoDB
Load Recall List from MongoDB
Merge Recall List and New Results
Display Output

Fig. 2. Flow chart of the process used to merge querries from the cache and from the online database and then perform parallel processing using Apache Spark.

two technologies integrate seamlessly because the *xeno-canto* API returns JSON data which directly forms the basis for documents in the MongoDB collection.

The example query in Listing 1, which asks for all recordings made in the United States that are 10 seconds long ($\pm$ 1%). It returns over 200 results, of which we use a subset for demonstration purposes.

Listing 1
Xeno-canto QUERY

```
query=cnt:"united states"+len:10
```

Each result contains a host of metadata but for purposes of caching we are most interested in the `id` field which is a unique identifier provided by *xeno-canto*.

We loop over each result and query MongoDB if an entry with a matching `id` is already in the file collection. We further query if an entry with a matching `id` has identical filter parameters to the parameters we requested. If it does not then we add this `id` to a list of recordings to be fetched and filtered. MongoDB's flexible schema allows for very straight forward matching of `id` fields and filter parameter fields which may or may not be present in each record, or of the same type and length. This is a desireable property since if a recording is present in the cache but was filtered with different parameters then we would still consider this a cache-miss.

After processing all of the results we have two lists of entries: recordings that were found in the cache and recordings that were not. We pass the list of not-found recordings to Spark through a Spark DataFrame for parallel downloading and processing.

B. User Defined Functions

Through the power of Apache Spark and the PySpark Python module we are able to download and process each recording in parallel. This capability is provided by the UDF (user defined function) module of PySpark. The signature of the UDF we defined for downloading and filtering is given in Listing 2.

Listing 2
PYSPARK USER DEFINED FUNCTIONS

```
@udf(returnType=ArrayType(DoubleType()))
def load_and_filter(
  file_url,
  filter_edges,
  filter_amplitudes):
  ...
```

The annotation `@udf` tells PySpark that this function can be offloaded to the Spark engine and further that it should expect the return type to be an array of double precision values. This return type annotation is crucial since Spark is natively running the Scala programming language on the Java Virtual Machine so types of data that are transfered to the Spark engine must be mapped to Java language types.

To call this function we use the `select` operation on our DataFrame as shown in Listing 3

Listing 3
PYSPARK DATAFRAME SELECT()

```
new_df = df.select(
  '*',
  load_and_filter(
    'file',
    'filter_edges',
    'filter_amplitudes').alias('file_content')
)
```

The `select` operation takes a comma-separated list of DataFrame column names and returns a new DataFrame with those columns. In this example we use `*` for the first parameter to say that we want to select all columns of the source DataFrame. The second parameter represents appending a new column to the DataFrame which will be called `file_content` and will be the actual samples we read from the downloaded MP3 recording from *xeno-canto* and filtered by our DSP filter. The source of this new column comes from three columns already in the DataFrame - `file`, `filter_edges`, and `filter_amplitudes` - that we pass to the UDF shown in Listing 2. The Spark engine will call this UDF on each row of the DataFrame in parallel using as many computing resources as are available in the Spark cluster. This means that downloading the original recording never occurs in the local processing environment.

C. Digital Filtering

After downloading a recording our UDF creates a digital filter using the Parks–McClellan algorithm [15] according to the parameters specified by `filter_edges` and `filter_amplitudes`. This filter is then applied to the recording samples which are returned by the UDF and stored as a new column in the DataFrame called `file_content` (as described in the previous section). It should be noted at this point that all the data movement (with the exception of the *xeno-canto* metadata) has occured on the remote Spark cluster and not in the local Python environment.

D. Merging New Results and Cached Results

Merging new results and cached results is acomplished simply by creating a new Spark DataFrame from the Spark MongoDB connector plugin using a query of all requested `id` fields and not just the fields that required downloading and filtering. Since the last step performed by the UDF in listing 2 is to store the new results DataFrame into MongoDB we can leverage MongoDB's ability to efficiently query documents and create a new Spark DataFrame that contains all the recordings we are interested in.

E. Viewing Results

At this point we have a DataFrame where each row contains the results of filtering the *xeno-canto* recordings. We could directly view these results by calling `collect` on the DataFrame but if we have more operations to perform it is more efficient to keep the data in Spark and offload further processing to the Spark cluster.

As an example of how to do this we created the UDF shown in Listing 4.

Listing 4
TAKING FFTS WITH SPARK

```
@udf(returnType=ArrayType(DoubleType()))
def get_magnitude_spectrum( signal, Nfft ):
    return np.abs(
      np.fft.fft(
        np.array( signal ),
        n=Nfft
      )
    ).tolist()
```

Here we take the FFT of each signal and return that as a new column. To call this we again use the Spark DataFrame function `select` as shown in Listing 5. The difference from Listing 3 is that now we are not selecting all columns but only the columns we need since the result is going to get collected from the Spark cluster into our local python environment.

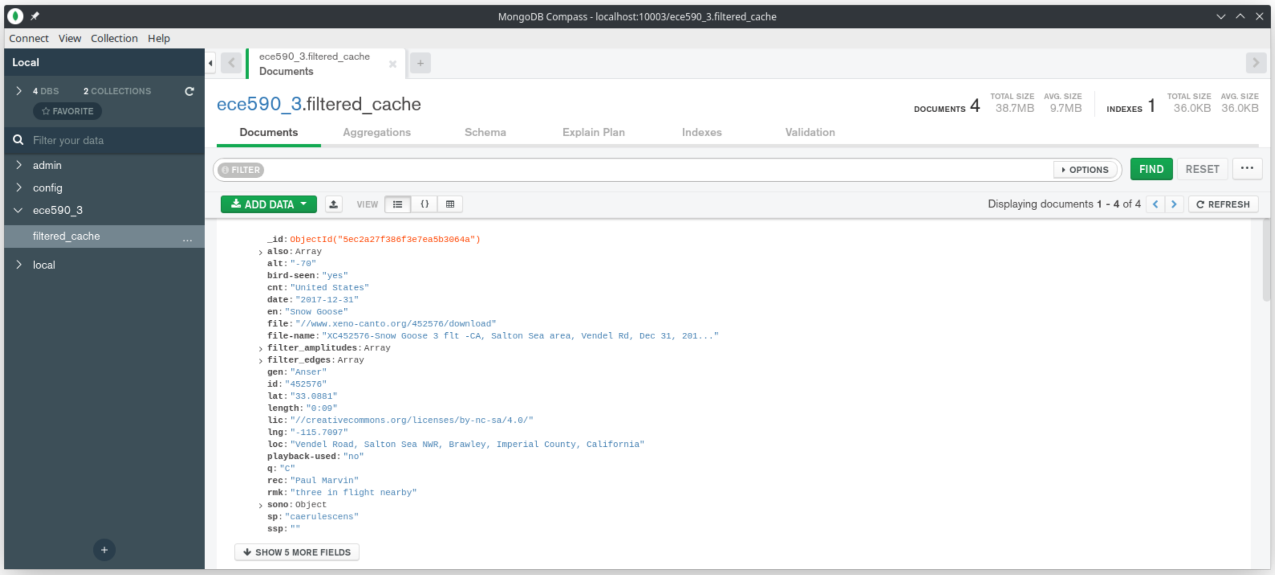
Listing 5
COLLECTING UDF RESULTS FOR LOCAL PROCESSING

```
Nfft = 4096
spec = df.select(
  'id',
  get_magnitude_spectrum(
    'file_content',
    lit(Nfft)
  ).alias('mag')
).collect()
```

The local DataFrame `spec` only contains the columns `id` and `mag`. For each row (there are 4 in this example) contains 4096 floating point values in the `mag` column which represents a huge reduction in memory since the original MP3 files were all approximately 10 seconds long with 48000 samples per second. Figure 4 shows local plots of the spectra for the 4 recordings that we fetched from the cache. Using Spark and UDFs we were able to obtain recordings, bandpass filter them, and obtain their spectra without ever moving data out of the Spark cluster until the last step. In this example we asked for filters that passed the band from 10% to 20% of the sampling frequency. We can see in the plot that we have indeed acheived the desired filtering.

Because the filtered files are cached in MongoDB, if we desire to visualize them in other ways or do further processing there is no need to download the source files again.

In addition to using Python data processing libraries (matplotlib, numpy) to visualize data we can also directly view and control the cache using MongoDB Compass [5] as described before. Figure 3 shows one of the cache entries visualized by that tool.



MongoDB Compass - localhost:10003/ece590_3.filtered_cache
Connect View Collection Help
Local
4 DBS 2 COLLECTIONS
☆ FAVORITE
Filter your data
admin
config
ece590_3
filtered_cache
local
ece590_3.filtered_cache
Documents
ece590_3.filtered_cache
DOCUMENTS 4 TOTAL SIZE 38.7MB AVG. SIZE 9.7MB INDEXES 1 TOTAL SIZE 36.0KB AVG. SIZE 36.0KB
Documents Aggregations Schema Explain Plan Indexes Validation
FILTER
OPTIONS FIND RESET
ADD DATA VIEW
Displaying documents 1 - 4 of 4
REFRESH
SHOW 5 MORE FIELDS

```
  _id: ObjectId("5ec2a27f386f3e7ea5b3064a")
› also: Array
  alt: "-70"
  bird-seen: "yes"
  cnt: "United States"
  date: "2017-12-31"
  en: "Snow Goose"
  file: "//www.xeno-canto.org/452576/download"
  file-name: "XC452576-Snow Goose 3 flt -CA, Salton Sea area, Vendel Rd, Dec 31, 201..."
› filter_amplitudes: Array
› filter_edges: Array
  gen: "Anser"
  id: "452576"
  lat: "33.0881"
  length: "0:09"
  lic: "//creativecommons.org/licenses/by-nc-sa/4.0/"
  lng: "-115.7097"
  loc: "Vendel Road, Salton Sea NWR, Brawley, Imperial County, California"
  playback-used: "no"
  q: "C"
  rec: "Paul Marvin"
  rmk: "three in flight nearby"
› sono: Object
  sp: "caerulescens"
  ssp: ""
```

Fig. 3. MongoDB Compass Application for viewing and managing MongoDB databases.

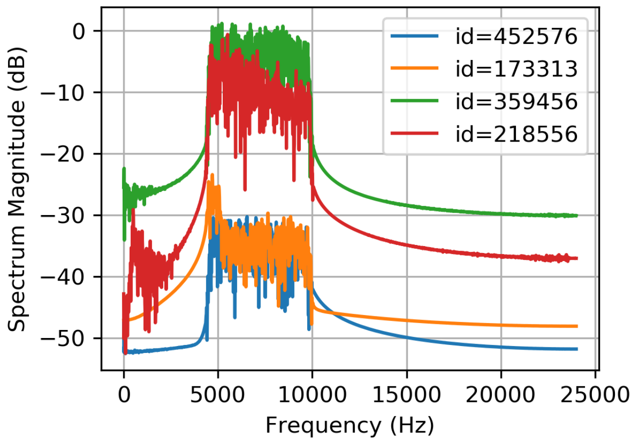


Fig. 4. Results generated by running two separate Apache Spark jobs: one to download and apply a bandpass filter to four different recordings from xeno-canto.org in parallel, and another to compute the magnitude spectra of the recordings.

V. Conclusion

In conclusion based on the projects result we can conclude that in fact the project operated as intended as seen in the data and graphs in the report. The scope of this project was to incorporate the techniques learned in ECE 499 and combine those techniques with Machine Learning and signal processing. Although this was just incorporated in something as simple as bird sounds these practices can be scaled onto data that is more rigorous and/or import. Finally this report hopes to inform the audience of the bridge between Signal Processing and Machine Learning and how they can be applied to uses cases.

References

[1] Kaggle, "Kaggle: Creating a dataset," https://www.kaggle.com/docs/datasets#creating-a-dataset, last visited on 2020/5/17.

[2] XenoCanto, "Kaggle: Sharing bird sounds from around the world," https://www.xeno-canto.org/, last visited on 2020/5/17.

[3] F. de Abreu e Lima, "Bird songs from europe (xeno-canto)," https://www.kaggle.com/monogenea/birdsongs-from-europe, last visited on 2020/5/17.

[4] MongoDB, "What is mongodb?" https://www.mongodb.com/what-is-mongodb, last visited on 2020/5/19.

[5] ——, "Mongodb compass: The easiest way to explore and manipulate your mongodb data," https://www.mongodb.com/products/compass, last visited on 2020/5/19.

[6] Spark, "Apache spark: Lightning-fast unified analytics engine." https://www.apache.org/, last visited on 2020/5/19.

[7] D. Robinson, "The incredible growth of python," https://stackoverflow.blog/2017/09/06/incredible-growth-python/, Stack Overflow, Tech. Rep., 2017.

[8] Jupyter, "The jupyter notebook," https://jupyter.org/, last visited on 2020/5/19.

[9] OCI, "Open container initiative," https://www.opencontainers.org/, last visited on 2/8/2020.

[10] RedHat, "Building, running, and managing containers on red hat enterprise linux 8," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/building_running_and_managing_containers/index, last visited on 2/8/2020.

[11] D. Smith, "Best practices to enable continuous delivery with containers and devops," Gartner Research, Tech. Rep., 2020.

[12] Microsoft, "Microsft azure overview," https://azure.microsoft.com/en-us/overview/, last visited on 2020/5/19.

[13] O. S. Initiative, "The mit license," https://opensource.org/licenses/MIT, last visited on 2020/5/19.

[14] G. M. U. O. of Academic Integrity, "George mason university honor code 2019-2020," https://oai.gmu.edu/wp-content/uploads/2019/08/George-Mason-University-Honor-Code-2019-2020-final.pdf, last visited on 2020/5/19.

[15] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd ed. USA: Prentice Hall Press, 2009.