The NØVID* Project

App Implementation to Support COVID-19 Management

CS-505 Final Project, Spring 2021, University of Kentucky

Prepared for Dr. Cody Bumgardner

By Javid Fathi & Alex King

(Team X)

* Unrelated to the wildly successful mobile app, so don't sue us.
It's not like we're making a profit here.

Introduction:

Over the last 14 months, federal, state, and local governments have struggled to respond to the threat of COVID-19 in a timely manner. As populations are tested for the virus, public health officials need to direct their patients quickly and systematically with next steps while considering their symptoms, location, and other health needs.

In Kentucky, nearly half of the state's 120 counties are rural and struggle to have continuous access to healthcare resources (e.g.: Internet, hospice care, virus testing, etc.). Because of these needs, defending populations against the virus has become an important engineering problem, and this project was developed to prototype a solution.

We propose the deployment of an asynchronous testing model, where healthcare providers document COVID-19 tests and patients using a simple cross-platform application (i.e.: accessible on browsers, MacBook, Windows, iOS, etc.). If the provider is unable to access the Internet, the patient records are temporarily stored to an in-memory process of the app, which will automatically push to a global webserver the next time that the machine connects to the Internet.

In this report, we discuss a prototype of this global webserver, which analyzes records across ZIP codes in Kentucky and automatically assigns patients to hospitals or their homes based on their symptoms and test results.

Please note: We have included our source code files, but this implementation requires the use of a Kafka executable bin, along with temporary log locations for Kafka and Zookeeper.

If you attempt to run this source code on a machine other than the given host (server), please download Kafka and correct the Bash directory variables in reset.sh and kill_bg.sh.

System Design

Design Summary

Our system design designed works off of a document datastore model to handle the given CSV files for hospital data and distances between zip codes. These were loaded to independent collections within a MongoDB Cluster, with databases for patient data, hospital data, and distances between Kentucky zip codes.

From there, the team used a RabbitMQ subscriber to handle the in-stream of patient records and implemented three categories of functions for the application: logical, real-time-reporting, application management.

The RTR functions use a Kafka servers to handle the incoming data, which ports the streams to an asynchronous handler app called Faust.

Lastly, Flask was used to handle REST calls for the HTTP endpoints.

The reasoning behind using these components was that:

1) MongoDB provides simply querying and great data visualizations for debugging.
2) Kafka was used since it provides high throughput for a high number of events, and Faust was chosen since it was explicitly written to handle Kafka streams.

As a challenge for unified development, we chose to use Python as the sole programming language for this prototype, along with bash scripts to restart processes and clean outdated logs.

Design Discussion

For this project, we were tasked by the Kentucky state government to prototype a solution for reporting and managing healthcare in relation to the COVID-19 pandemic. In practice, this meant developing a web application that could handle several threads of logic simultaneously.

First, the application would need to process and analyze data streams of patient records for real-time case growth by ZIP code and provide alerts at the local and state level accordingly.

Second, the application would need to assign patients either to their home or a hospital based on their status (symptoms and test results).

Third, the application would need to handle HTTP queries for real-time data on info related to the state's handling of COVID-19 (e.g.: alerts, patient assignment, hospital bed counts, etc.)

Patient records were represented as JSON dictionaries and record batches were sent in the form of lists. All RMQ streaming data was transmitted with UTF-8 string encoding. Each patient record is given in a standardized form. This makes it possible to process the data into both SQL tables for real-time alerts and NoSQL datastores for location assignment and long-time storage.

This record can be reviewed on the following page:

{

first_name: String representing the patient's first name.

last_name: String representing the patient's last name.

mrn: Unique identifier across the application to identify the patient.
UUID stored as a string.

zip_code: Integer representing the location of the person. Stored as a string.

patient_status_code: A single-digit integer code representing the case status.
Stored as string. Used to determine location_code.

0: no symptoms, untested

1: no symptoms, tested negative

2: no symptoms, tested positive

3: symptoms, untested

4: symptoms, tested negative

5: symptoms, tested positive

6: critical/emergency symptoms, tested positive

}

DBMS Design Choices

For our database solution, we chose to use a MongoDB cluster for its flexibility, split-second response times, and easy integration. Ultimately, all patient, hospital and ZIP code information is processed and stored into the cluster.

Additionally, our team was given CSV files of data for state hospitals and the distances between state ZIP codes. We uploaded both to our MongoDB cluster to be used extensively throughout our implementation.

Upon starting or resetting the application, a local CSV file of hospital data can be uploaded to the cluster as a default collection for hospital data. While we do not expect hospital counts to change often, it is more common for total bed numbers to increase or decrease.

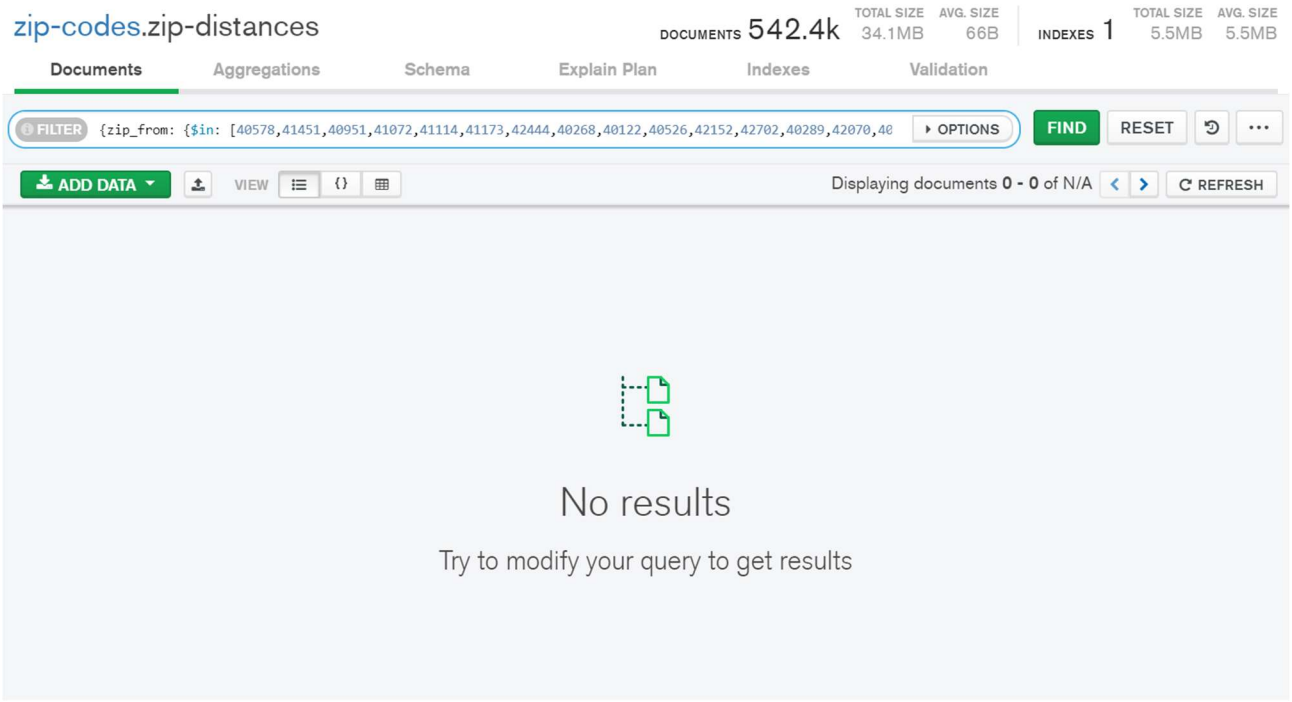
To keep the prototype from unnecessary down-time, we have placed this process under an independent HTTP query:

Normal Reset: http://server.cs.uky.edu:9000/api/reset

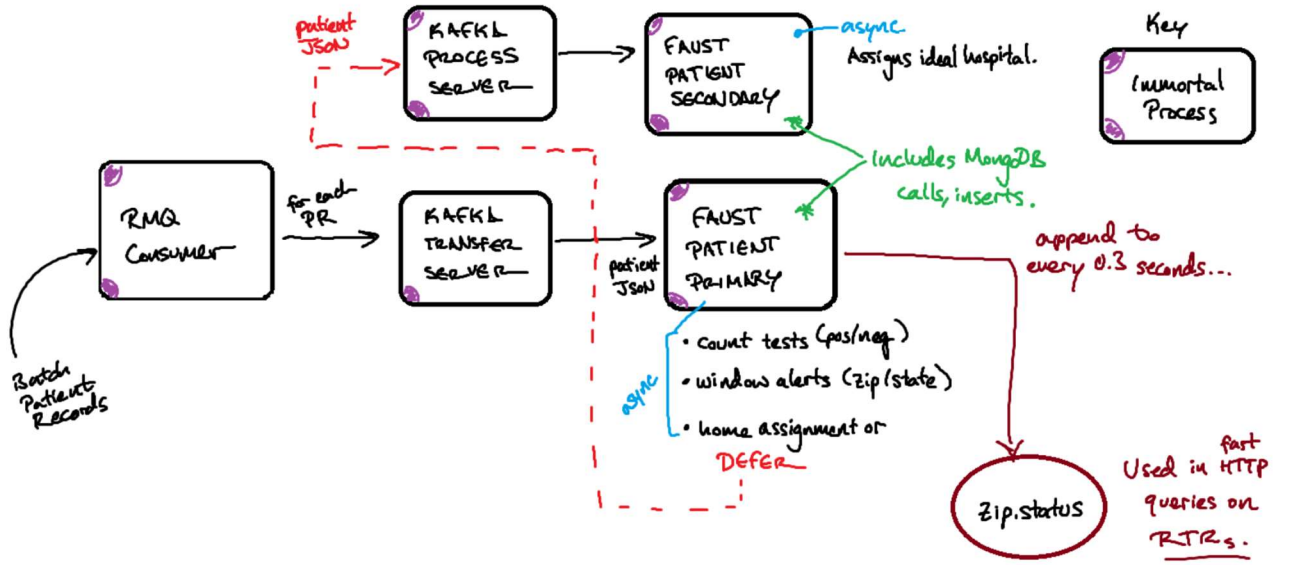
Reset w/Hospital Upload: http://server.cs.uky.edu:9000/api/reset/with-hospitals

Note: No mechanism was designed to regularly upload ZIP code data to the MongoDB cluster. It was assumed that ZIP codes do not change often enough to justify this process.

That being said, we discovered that some Kentucky ZIP codes are unavailable in our existing collection, and the existing data should be updated before deployed to real-world applications.



Real-Time Design Choices



To meet our real-time requirements, we started by partitioning a RabbitMQ Consumer to read in data from a pre-existing Producer. In practice, this Consumer would be replaced with a secured web-server capable of receiving form data and passing it to the necessary microservices services.

We used our advisor's RMQ Subscriber Python code for this step – with one significant change. When the Consumer receives a batch of data – on callback – the data is parsed and transmitted to a local Kafka server record-by-record, which brings us to our next tool.

To transmit data between concurrent processes, we chose to use Kafka servers with Zookeeper initializations. When the application starts or is reset, two Kafka servers are started with their own Kafka topic to handle data transfers and data processing, respectively. Each Kafka topic is initialized with 50 partitions, meaning that – in theory – the Kafka server will be able to handle up to 50 concurrent processes of an identical nature. In practice, that number is likely lower, but the partitioning allows for greater asynchronization.

When our Kafka server for processing receives a patient JSON record, the data is forwarded to a Faust process, which handles most of the logic for our real-time reporting via asynchronous threading:

- One of two global patient counters are incremented by one, depending on the patient's status.
    - If the patient has tested positive for COVID-19, then a "positive patient" counter is incremented.
    - Otherwise, a "negative patient" counter is incremented.

- The patient's status code is also used to determine where they should be assigned.
    - If the patient has tested negative or is asymptomatic (statuses 0, 1, 2, 4), then they are placed on home assignment and their data is pushed into the appropriate MongoDB collection.
    - If the patient has tested positive and is symptomatic or experiencing a critical medical emergency (status 3, 5, 6), then they must be taken to an appropriate hospital.
        - For now, the patient is left unassigned (-1), and their data is pushed into the appropriate MongoDB collection.
        - Then, the hospital decision is deferred to a concurrent Faust process *reserved* for hospital assignments – via the second Kafka server.
            - If the patient is experiencing a medical emergency, then they will be sent to the *best* available treatment facility within their vicinity.

              Decision Criteria: Level 1 > Level 2 > Level 3 > Level 4 > N/A

            - In any other case, the patient is assigned to the nearest available treatment center with an available bed.

              Notes:

              The given CSV file needed to be altered so that UK Hospital would be recognized as a Lvl I treatment facility.
              LEVEL I, LEVEL I PEDIATRIC → LEVEL I

              If a ZIP code is not present in the distances graph/collection, the patient will be placed on home assignment. (Must be resolved before full deployment.)

What remains underdeveloped in our Faust process is the windowing functions.

```
zip_list = []
for r in windowed_covid_cases:
    zip_cases_overall = len(
        windowed_covid_cases
            .relative_to_now().values()
    )
    if zip_cases_overall == 0:
        continue
    zip_cases_new = len(
        windowed_covid_cases
        .relative_to_now()[zip]
        .items().delta(WINDOW_TIME)
    )
    zip_cases_prev = zip_cases_overall - zip_cases_new
    if zip_cases_new >= 2 * zip_cases_prev:
        zip_list.append(r.zip_code)

zip_list.sort()

if len(zip_list) >= STATE_ALERT_MIN:
    state_status = ON_ALERT
else:
    state_status = NO_ALERT
```

Faust is a very new module. It is intended to be a replacement for Apache Flink or Spark.

Although Faust has objects to handle windowing, next to no documentation exists. Javid had to join a Slack channel with the module developers to get assistance, and even then, we're *still* not sure of what to change!

Ultimately, once we have an answer on how to properly window and update our Kafka data in SQL tables, we'll be able to share alerts without hassle.

```
STATUS_LOGS = open(STATUS_FILENAME, "a")
STATUS_LOGS.write(str({
    "ziplist": zip_list,
    "state_status": state_status,
    "positive_test": n_positive_tests,
    "negative_test": n_negative_tests
}) + '\n')
STATUS_LOGS.close()
```

All real-time reporting data required for outside viewers is appended to a running log every 0.3 seconds that the Faust process is alive, making the log virtually synchronous with any updates in patient records. This log – zip.status – is used in all relevant web-server queries to the application. All RTR processes are cleanly tracked with independent logs for internal review.

Ultimately, all RTR processes are killed or started using two bash scripts, which also clear all dependent logs before starting the new background processes.

Web-Server Design Choices

To handle HTTP queries, we chose to use a Python Flask app running on the expected URL:

http://server.cs.uky.edu:9000/

This solution allowed for background logging and sub-second responses on most requests.

See next page for Individual Contributions.

Individual Contributions

Click here to ensure app is running: http://server.cs.uky.edu:9000/

Management Functions

Management functions were a shared team responsibility.

Alex King, MF1: http://server.cs.uky.edu:9000/api/getteam

Javid Fathi, MF2: http://server.cs.uky.edu:9000/api/reset

Javid Fathi, MF2a: http://server.cs.uky.edu:9000/api/reset/with-hospitals

Real-time Reporting Functions

Javid completed all RTR functionality, along with necessary Flask endpoints.

Javid Fathi, RTR1: http://server.cs.uky.edu/api/zipalertlist

Javid Fathi, RTR2: http://server.cs.uky.edu/api/alertlist

Javid Fathi, RTR3: http://server.cs.uky.edu/api/testcount

Logical/Operational Functions

These functions were a team effort, led primarily by Alex.

Alex King & Javid Fathi, OF1: See *background_rtr/patient_mover.py*

Alex King, OF2: http://server.cs.uky.edu/api/getpatient/{mrn}

This URL reports back with the MRN of the patient, along with their location code of patient:

- 0 for at home,
- Hospital ID for the hospital location of the patient,
- -1 for no assignment, or
- "NOT-IN-SYSTEM" for an invalid MRN identifier.

Alex King, OF3: http://server.cs.uky.edu/api/gethospital/{id}

This URL reports the number of available & total beds, and ZIP code of the hospital with the given ID. If the queried ID is invalid, then the returned ZIP code will be -1, along with 0 beds (both in total & available).