

Mushroom Classification
DSCI551 Spring 2024

By:

Allison Chan (6905922727)
Brandyn K. Lee (2422176714)
Eric Crouse (9271913867)

Table of Contents

Introduction	1
Project Details	1
Planned Implementation	1 - 3
Architecture Design	
Data Flow Diagram	3
System Design Flow Diagram	4
Implementation	
Tech Stack	4
Functionalities-Back End	5-8
Screenshots-Back End	5-8
Functionalities-Front End	8-9
Screenshots-Front End	8-9
Learning Outcomes	
Challenges	10
Individual Contribution	10-11
Conclusion	11
Future Scope	11-12

Introduction

The central aim of this project was to create a distributed database system that would be efficient for storing, retrieving, and updating large volumes of data, complete with a user-friendly front-end interface, and easy-to-use back end database management. To build this system we used eight Google Firebase Realtime Databases, each of which hosts a subset of our data, partitioned on three key attributes. A web-based application for end users was created using Streamlit which allows for a high level of user interactivity.

The purpose of our application is to allow a user to identify whether a mushroom is poisonous or edible based on its visible characteristics, habitat, etc. They can do so via a series of button clicks that efficiently query our data and display the results. The first three options presented to the user represent the attributes on which our data is partitioned. After that, the corresponding database is queried and intermediate results are displayed and updated with each new click. As soon as the mushroom can be identified with certainty (based on our data, of course), the final results are displayed and the user is given the option to start over. Our application also features a number of other interactive visualizations, such as poisonous/edible distributions among certain attributes.

Project Details

Title	Mushroom Classification
About	Mushroom Edibility Identifier
Github Repo	https://github.com/chanalli/mushroom
Data Source	https://archive.ics.uci.edu/dataset/73/mushroom

Planned Implementation

Originally, our team considered using either Flask + JS or Streamlit to create our user interface, and we opted for Streamlit due to its simplicity of deployment and interactivity. Our intent was to allow users to describe characteristics of a mushroom and have the application determine whether the given mushroom is poisonous or edible, which we accomplished via a series of button clicks, each representing a different attribute. We also intended to also have a real-time prediction indicator on if the mushroom is edible or poisonous based on the permutations of current features from user input, which we accomplished via the horizontal progress bar graphic, which displays the percent likelihood that the mushroom is

poisonous/edible after each attribute is selected. We had originally hoped to host a gamified version of this, wherein users would guess whether a mushroom was poisonous or edible based on characteristics, but unfortunately the time allotted for this semester-project did not allow for us to complete that stretch goal.

For our back end database system, we originally had debated between Firebase and MongoDB, and in the end we chose Firebase for its flexibility, ease of collaborative use, and real time capabilities. We had planned to partition our data first by poisonous/edible, then after some thought we decided to partition based on the attribute 'odor'. When odor gave an uneven spread among databases, we were forced to make some changes in order to ensure an even distribution of our data. Ultimately, we partitioned our data into eight databases based on the three attributes having the most evenly distributed values, ensuring the most even hashing possible. A hashing function was created which encodes each record by its values for each of those three attributes and assigns it to one of our eight databases (0-7) accordingly. Each record holds unique values for each attribute, including whether it is poisonous or edible, which allows for filtering and querying based on the search criteria specified by the user.

Team Responsibilities

- [Allison] Collect and/or find data
- [Eric + Brandyn] Analyze the data + what we're going to use
- [Eric] Preprocess/Clean Data
- [Brandyn + Eric] Research + choose appropriate database to use
- [All] DB schema design
 - How many tables, relation (if any), determine relevant info, etc
- [Allison + Brandyn + Eric] Populate databases (use appropriate hashing function)
 - Determine partitioning method
- [Allison + Brandyn] Hash out queries + filters
- [Brandyn + Eric] Data visualizations with Plotly and Streamlit
- [Allison] Design + set up front end UI
 - [Allison] Connect database to front end UI

Timeline

✓ = done ⏳ = in progress ✗ = blocked

Date	Goal	Notes	Status
Fri 2/2	project proposal due		✓
	finalize data sources	Analyze, Preprocess, Brainstorm front end features/use cases	✓
Thur, 2/15	Finalize data format	Clean data	✓

	finalize data manipulation requirements/use cases		✓
Fri 3/1	midterm progress report	Populate DB with data	✓
	ingest/process data through DB manager perspective (command line base)		✓
	extract/query data to be used in front end		✓
Wed 4/17	in class demo	Query DB	✓
Mon 4/29	complete front end		✓
Fri 5/3	final report due		✓
	finalize demo		✓

Architecture Design

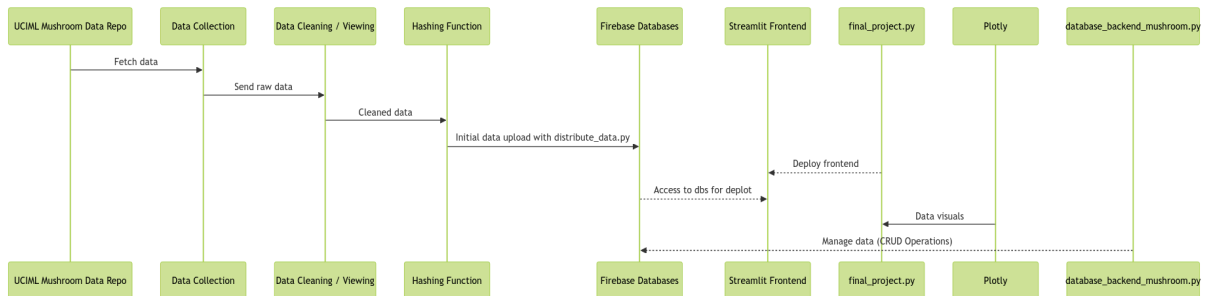


Figure 1. Data flow diagram

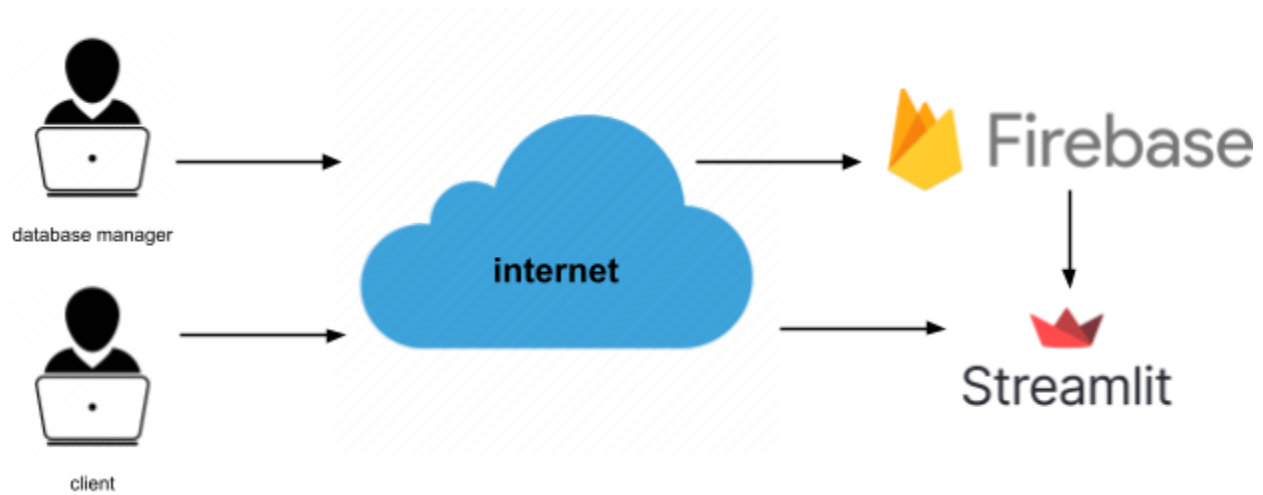


Figure 2. System design diagram

Implementation

Name	Usage
Python	- Self explanatory.
Firestore	<ul style="list-style-type: none"> - REST API: access to real-time database via REST API allowing interaction with Firestore's data store with HTTP methods - requests library: python library for simple integration of Firestore CRUD operations (or any REST API) for data management
Plotly	<ul style="list-style-type: none"> - Advanced Data Visualizations: offers multitude of plotting capabilities, ideal for interactive graphs for end users to explore the data - Customizability: provides extensive options for customization of graphs such as tooltips, zooming, labeling, layouts - Integration with Streamlit: intuitive integration with Streamlit using the Plotly python wrapper to easily deploy data visuals in Streamlit
Streamlit	- Python Library: allows developers to create interactive web applications directly from Python scripts

Github	<ul style="list-style-type: none"> - Version Control - Collaboration Tools
---------------	--

Functionalities-Back End

To optimize the data distribution across our databases and prevent bottlenecks, we crafted a hashing function that keys on the attributes 'gill-color,' 'cap-color,' and 'habitat.' These attributes were chosen due to their high cardinality, meaning each has a wide range of distinct values and, importantly, because they exhibit less skew in their values compared to other attributes. This approach minimizes the risk of data clustering too heavily on any single database and is vital for maintaining a balanced load across our system.

The hashing algorithm begins by merging the attribute values from the stated attributes from each data point into a single string, which is converted into bytes. Such bytes are then hashed using the SHA-256 hash function. This robust algorithm is defined by its three fundamental features: collision resistance, preimage resistance, and second preimage resistance [1, p. 7]. The algorithm's collision resistance is 128 bits [1,p. 8], allowing for high data integrity and security. Such, in conjunction with its other two properties, makes it a suitable hashing algorithm for our needs as it securely and uniformly spreads out our data while still being able to handle new data entries efficiently.

After hashing, we convert the resulting hash value into a hexadecimal string, which we then turn into an integer using base 16. This integer is crucial for the next step: mapping the data entry to one of our databases. Using a simple modulo operation with the number of databases, we can easily assign each entry to a specific database, thus ensuring a smooth and even data distribution across our network. This method supports our current data load and is scalable to accommodate future entries without overwhelming any single database.

Using "database_backend_mushroom.py" will allow backend users to perform CRUD operations on the data through a simple command line interface. Given that the README.md has been followed for project setup, all of the new rules (Figure 1) have been updated properly for each Firebase database as outlined in the README.md. If not, please do so before reading on.

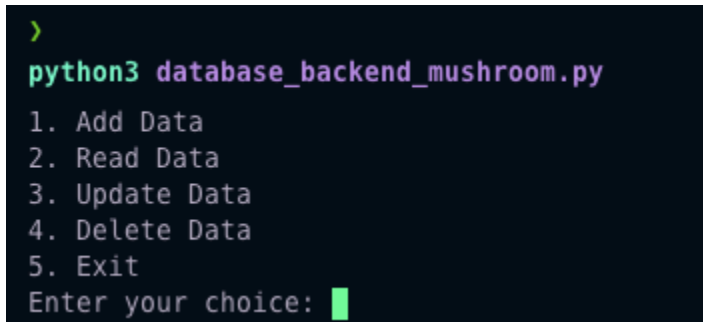
```
{
  "rules": {
    ".read": true,
    ".write": true,
    ".indexOn": ["bruises", "cap-color", "cap-shape", "cap-surface", "gill-attachment",
    "gill-color", "gill-size", "gill-spacing", "habitat", "odor", "poisonous", "population",
    "ring-number", "ring-type", "spore-print-color", "stalk-color-above-ring",
    "stalk-color-below-ring", "stalk-root", "stalk-shape", "stalk-surface-above-ring",
    "stalk-surface-below-ring", "veil-color", "veil-type"]
  }
}
```

Figure 3. Example of Rules in Firebase

With this understanding of the setup of our databases, "database_back_end_mushroom.py" can be launched from the root folder to perform CRUD operations on our data using the requests library with the Firebase REST API. The most crucial part of our code lies in its robustness to updates of attribute values associated with the hashing function. Whenever the "update_data()" function is called, it will save the "db_id" of the indexed data prior to the update; if one of the hashing attribute values is modified and the newly generated "db_id" (via gen_hash()) is not equal to the original "db_id" currently saved in memory, a "requests.put()" response is invoked to move the JSON data from its current location to its new database. Lastly, it will remove the old entry using "requests.delete()." This methodology allows us to properly maintain the initially assigned index (i.e., key) value. The "add_data()" function also index assigning convention as it utilizes "get_next_index()" to determine the subsequent unassigned number to be assigned to the data entry.

Screenshots-Back End

From the root folder containing the cloned repository, start a virtual environment with the appropriate requirements and run "python3 database_backend_mushroom.py" to initialize database management through the command line (Image 1).



```
>
python3 database_backend_mushroom.py
1. Add Data
2. Read Data
3. Update Data
4. Delete Data
5. Exit
Enter your choice: █
```

Image 1. Start up screen database manager python script

To read data, enter "2" and hit "Enter" to generate the next prompt; input index number of data to read. It will return the JSON data associated with the inputted index (Image 2). Data of index 59 has a "cap-color" of "n" representing brown. Updating this value can affect the output of the hashing function, making it necessary to migrate the data and all of its attributes to the new database and remove it from the old. For example, data index 59 currently exists in DB 2. When I choose option "3" to update the data, changing the cap-color to "y" forces the script to move the data with its key maintained to DB 0 as shown (Image 3). This functionality utilizes Firebase's real time databases capabilities to dynamically and securely read and update our data and its storage.


```

Enter your choice: 2
Enter the index of the data to retrieve: 59
Data found in DB 2: {
  "bruises": "t",
  "cap-color": "n",
  "cap-shape": "x",
  "cap-surface": "y",
  "gill-attachment": "f",
  "gill-color": "p",
  "gill-size": "b",
  "gill-spacing": "c",
  "habitat": "p",
  "odor": "a",
  "poisonous": "e",
  "population": "y",
  "ring-number": "o",
  "ring-type": "p",
  "spore-print-color": "k",
  "stalk-color-above-ring": "w",
  "stalk-color-below-ring": "w",
  "stalk-root": "r",
  "stalk-shape": "e",
  "stalk-surface-above-ring": "s",
  "stalk-surface-below-ring": "y",
  "veil-color": "w",
  "veil-type": "p"
}

```

Image 2. Read data implementation for database manager

```

1. Add Data
2. Read Data
3. Update Data
4. Delete Data
5. Exit
Enter your choice: 3
Enter the index of the data to update: 59
Enter the attribute to update: cap-color
Enter cap-color (brown: n, buff: b, cinnamon: c, gray: g, green: r, pink: p, purple: u, red: e, white: w, yellow: y):
y
Data moved and updated to DB 0 from DB 2, maintained index: 59

```

Image 3. Update data implementation with case of new `gen_hash()` value

The “`add_data()`” function (option 1) allows managers to add data by manually adding the on character attribute values for each respective attribute. The hash function will then appropriately determine which database to store the new data and response requests are used to write the new data. The index value (serving as our unique key/identifier for all of our data) is assigned as stated in the Functionalities section. Lastly, option 4 of “Delete Data” requires the user to input the “`db_id`” and “`index`” values; this serves as an extra step to ensure that the manager would like to delete the inputted index and its associated values (Image 4).

```
Data added to DB 6, index: 8125
1. Add Data
2. Read Data
3. Update Data
4. Delete Data
5. Exit
Enter your choice: 4
Enter the database ID: 6
Enter the index of the data to delete: 8125
Data successfully deleted.
1. Add Data
2. Read Data
3. Update Data
4. Delete Data
5. Exit
Enter your choice: 2
Enter the index of the data to retrieve: 8125
Data not found in databases at the specified index.
```

Image 4. Delete data implementation and confirmation of deletion by using read data

Functionalities-Front End

The front end is hosted on Streamlit, implemented with Python with embedded HTML and used the Plotly library. The website is designed to be user interactive where users are able to play around with the data in the backend in a gamified way. There is a section to select a single trait for each of the 23 mushroom characteristics, and for each trait there is a progress bar indicating the likelihood that the mushroom is edible or poisonous given the current selection (Image 5). The process of iterating through each characteristic continues until our data is able to determine that with the current selected subset of traits, the mushroom is 100% poisonous or 100% edible. If there is no match for such a permutation in our database, we are also able to determine that in real time and present an option to reset the process.

The first 3 selections of the characteristics will determine which database the data of our interest is hashed into as mentioned in the backend functionality. Such a database is queried for the entirety of its contents to be populated in a dataframe. Then with each additional selection of a characteristic, the data is filtered through the dataframe. Another option considered was to query the database through REST API over HTTP by modifying the filter in the query param. Both resulted in similar time differences, so we went with the first approach of reducing the number of requests and kept our initially queried data locally since it was relatively small.

Additionally, there are 3 interactive charts for visualization (shown below in Image 6) that are provided for support to help understand the data, especially when playing around with selecting traits. The first chart, pie chart, tells us the divide between poisonous and edible mushrooms holistically in our database. The second chart is a collection of bar charts for each characteristic and for each bar chart, it shows the number of poisonous or edible mushrooms for each trait. Lastly, there is the sunburst diagram (Image 6), which provides a visual representation

of the structure of our data based on our hashed attributes (ie, gill-color, cap-color, habitat), color coded by their poisonous attribute value. Hovering over any sector will provide a value for “poisonous” as either “p,” “e,” or “(?)” indicating poisonous, edible, or a mix of both respectively. The most true “red” colored sectors are that of “p” while the most true “light green” colors are that of “e;” all other hues of ed and light green sectors are that of “(?)” Clicking on any of the sectors in the inner two rings (ie, gill-color and cap-color from inside to out) filters the data on that attribute value exclusively and provides a more condensed view of our data. Hovering over any of the sectors also provides information such as “count” which is also represented visually by the size of the sector.

Screenshots-Front End



Image 5: Interactive section showcasing percentage edible versus percentage poisonous after 3 selections were made.



Image 6: Interactive data visualizations

Learning Outcomes

Challenges

One of the challenges of the project was determining what to do with this data. The data we are working with is already plentiful and structured, but figuring out how to leverage this aspect in an interesting and engaging way was a hurdle.

Additionally, while cleaning the data to be stored in our desired variety, JSON, we were required to clean the inconsistencies regarding value types that JSON supported. More specifically, we had to convert many NaN values to the character '?', which indicates a missing value according to the dictionary from UCI ML Repository. This was not due to a fault in the data but rather the inconsistent nature of real-life data, as some specimens examined were literally missing a certain attribute. These values were originally recorded as NaN, but this was not consistent with the data dictionary being used for this data/project.

Also, we were originally going to partition based on poisonous/edible, until we realized that feature-based partitioning (rather than target variable based) would allow for more efficient querying. The challenge then became how to select attributes for partitioning and how to hash them in a way that would evenly balance the load across all databases. We opted for the three attributes whose values had the most even distribution among records in our dataset to ensure a variety of hash values which could then be directed into one of our eight databases.

Lastly, learning new tools like Streamlit was a steep learning curve. Although the library was well documented and contained a lot of useful examples, utilizing all of its content fully required dedicated effort and experimentation to a degree.

Individual Contribution

Eric

- Along with Brandyn, analyzed and cleaned/preprocessed data, selected an appropriate database system, and developed code for database population. Created and hosted a key for mapping the single-character values our dataset to their corresponding English words (e.g. 'n'→'brown'); this can be found on Firebase as well as a json file named "mushroom-default-rtdb-import.json" on our GitHub repository. Also designed and implemented the percentage graphic which efficiently calculates intermediate probabilities of a mushroom being poisonous/edible after each user-click and displays the final result once edibility is definitively determined.

Allison

- Worked on obtaining the initial data to be used for this project. Helped with discussing and determining the database to be used, along with strategies on distributing the data. Then helped create and set up the database to be used for when the data is loaded. Set up

the front end visualization and interactive components of the streamlit website. Additionally helped debug data cleaning, data loading, and visuals.

Brandyn

- Wrote the hashing function to distribute data and populate to Firebase databases. Wrote the backend database management python script. Created data visualizations in Plotly. Helped write code for `load_df()` in front end to more effectively pull initially filtered data (ie, pull only requested data not the entire database).

Conclusion

In conclusion, our project successfully achieved its goal of developing a distributed database system capable of efficiently storing, retrieving and updating large datasets, complemented by a user-friendly front-end interface with a streamlined backend database management process. Utilizing 8 Firebase Realtime Databases, each tailored to host specific subsets of our data partitioned across 3 attributes used for hashing them into respective databases, we eventually constructed a robust foundation for our application.

Implementing with Streamlit, our web-based interface offers users a highly interactive interface, allowing for users to identify mushroom toxicity based on observable and selected characteristics. Through a series of intuitive button clicks, users are able to navigate attribute options, triggering efficient queries that dynamically update and pull results. Once a conclusive result is reached on how poisonous or edible a mushroom is, users are presented with final outcomes.

Beyond its primary function, our application offers additional interactive visualizations, enhancing the user's engagement and providing insights into the distribution of poisonous and edible mushrooms across various attributes. Overall, our project showcases the seamless integration of database management, front end development, and user interaction to address a practical and informative task.

Future Scope

Although we reached the goals of this project, there remain exciting opportunities for future enhancement and expansion. Machine learning integration with trained models on extensive datasets could improve the accuracy in prediction in which we would not only rely on the present data in the database. Additionally, if we expand the range and interactive components of data visualization, we will be able to offer deeper insights into mushroom characters and toxicity patterns. Advanced visualization techniques will also enrich the user experience, enabling a deeper understanding of the data presented. Lastly, since the dataset we worked with in the project contains a very small subset of a singular mushroom family, collaborating with other datasets to access additional mushroom data will broaden the scope and accuracy of our application.

By integrating with machine learning models, enhancing data visualization capabilities, and collaborating with even more external sources to access additional datasets, we further increase not only the accuracy, but the usability and relevance of our application.

Citations

[1]: Q. Dang, R. Blank, and P. Gallagher, “NIST Special Publication 800-107 Revision 1 Recommendation for Applications Using Approved Hash Algorithms,” 2012. Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf>