FlowDB

Monjoy Narayan Choudhury
IMT2020502
IIIT Bangalore
Monjoy.Choudhury@iiitb.ac.in

Hardik Khandelwal
IMT2020509
IIIT Bangalore
Hardik.Khandelwal@iiitb.ac.in

Tejas Sharma
IMT2020548
IIIT Bangalore
Tejas.Sharma@iiitb.ac.in

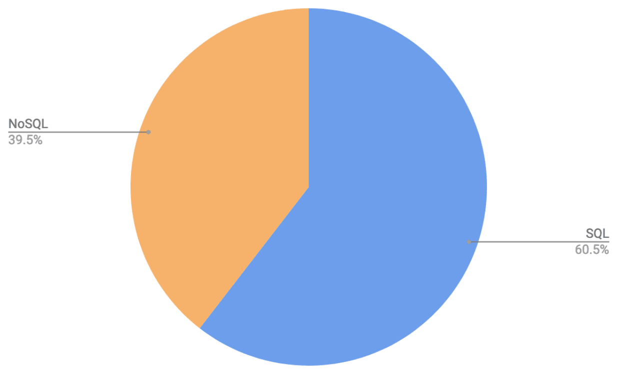


Figure 1. Courtesy of [4]: Market Share of database types

ABSTRACT

In this report, we introduce to you "FlowDB" a full-stack application that bridges the gap between SQL and NoSQL. We provide a data ingestion pipeline along with a simple query tool via an interactive UI. The application also allows a user to do data cleaning with various options available and define their own schema for the data provided in the form of a CSV. We present the work and idea behind this application and do a performance analysis using two benchmark tests on ingestion and query processing. We believe using this application users can leverage the technical advantages which NoSQL brings but couldn't do so due to their lack of underlying concepts.

PROBLEM DEFINITION

For a long time, NoSQL-based database solutions have existed in the market as well as well researched in academia. However, in a study shown by ScaleGrid in 2019 [4] a majority of the user base is still using classical SQL-based databases with MySQL being majorly used. It is also noticed that out of the 39.5 % share which NoSQL has, about 63% of it is filled with MongoDB users. This presents an interesting angle of users switching to MongoDB in the advent of the MERN Stack. However, we can still see that most users are accustomed to the "SQL" based working especially in the business analytics side where data entry technicians and analysts are much more comfortable working with SQL queries and avoid bothering about the advantage of distributed systems. FlowDB tries to bridge the gap between this. FlowDB aims to provide a familiar interactive GUI-based SQL approach with MongoDB running under the hood giving the advantage of both SQL and NoSQL so that it can cater to users who want to leverage the technical advantages which NoSQL brings but couldn't do so due to the lack of underlying concepts. In the themes given to us, this lies well under the category of Data Ingestion and Structured Data processing and Analytics. We try to provide a GUI-based data ingestion pipeline where the user can submit their default dataset in form of a csv and can define schema based on the fields present on the csv. They can perform data cleaning steps like null value handling and enforce datatypes on various fields. After this, we also provide an interactive way to perform simple queries (Select and Aggregate queries) which abstracts the MongoDB syntax away from users. This allows users with some amount of SQL knowledge to perform basic queries without knowing anything about MongoDB.

From a NoSQL standpoint, the challenge comes in obtaining the user inputs (in the form of interactive forms) and converting them into NoSQL-based commands. We designed multiple response structures which allow us to communicate between the MongoDB database and React frontend. In the ingestion process, the interesting part comes in defining a schema and using data to fill that schema due to the nature of MongoDB and SQL. SQL is a table-oriented data store while MongoDB stores it in a document format. Unifying them efficiently and in an explainable way was one of the main targets of our project.

Overall, we want to create a system that allows anyone without the knowledge of MongoDB to perform data ingestion along with data cleaning and gain analytics on the schema they created. Along with this we also want to create a backend that efficiently bridges this semantic gap between the 2 paradigms of SQL and NoSQL.

APPROACH

We discuss our approach in the following flow, we start with the selection of the tech stack and the reason for doing so. We follow this with a discussion of the features present in our program. We then deep dive into the features discussing how we did it and what were the challenges we faced.

Tech Stack

Our application is built on the MERN stack comprising MongoDB as our datastore, React as the frontend, and Express[1]

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

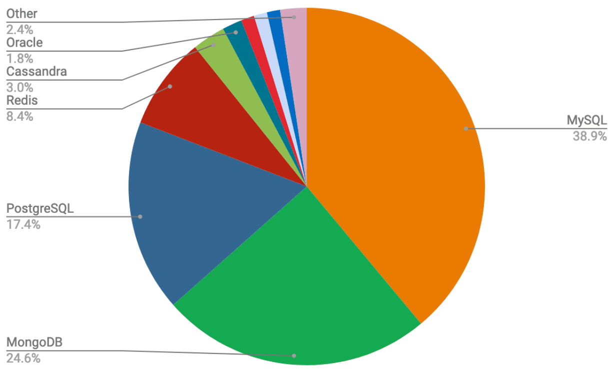


Figure 2. Courtesy of [4]: Distribution based on database

and Node[2] as the backend to handle server response and query the database.

Using React [5] as the frontend enabled us to structure our pages in components, which helped in reusability. While the reason for MongoDB is due to the following points:

1. Schema flexibility: MongoDB does not enforce a rigid schema. This makes it easy to store data in a flexible, JSON-like format. This allows us to also define our own schema once the data is loaded and not before the data is even loaded.

2. Scalability: MongoDB is horizontally scalable, which means it can handle large amounts of data and traffic by distributing it across multiple servers. This makes it a good choice for applications that need to scale quickly.

3. Integration with Node.js: MongoDB is the default database for Node.js and is supported by many popular Node.js modules. This makes it easy to integrate MongoDB into a Node.js application.

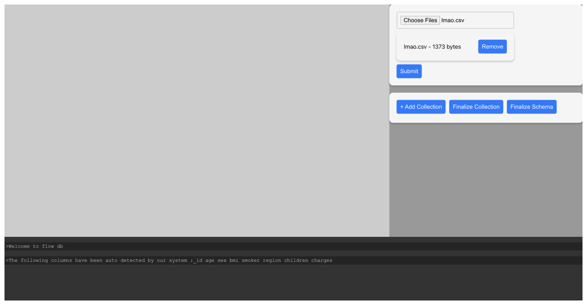
4. JSON format: MongoDB uses a JSON-like format to store data, which is easy to read and write. This makes it easy to work with data in a MERN stack application. Also the processing from frontend and sending requests from it becomes very simple as JSON is the standard format used for requests and responses.

Flow of our Application

The features of our application are as follows (presented in a sequential manner):

1. Once the application is first loaded. The user has the option to upload a csv file. Once the upload is complicated the terminal below will present the users with columns auto-found so that the users know that their upload was successful and some preview of the csv file uploaded.

2. Now a user will click on +Add collection and can define the schema of a collection. The user can define the features like null check: where they can state if they want to retain it, drop it, or replace it with some value with mean as the option also given.

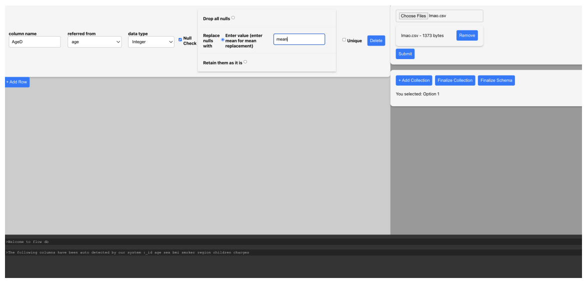


Imao.csv

Imao.csv - 1373 bytes

```
>Welcome to flow db
>The following columns have been auto detected by our system :_id age sex bmi smoker region children charges
```

Figure 3. FlowDB: Starting Screen



column name: AgeD

referred from: age

data type: Integer

Drop all nulls

Replace nulls with: Enter value (enter mean for mean replacement) mean

Retain them as it is

Imao.csv

Imao.csv - 1373 bytes

You selected: Option 1

```
>Welcome to flow db
>The following columns have been auto detected by our system :_id age sex bmi smoker region children charges
```

Figure 4. FlowDB: Data ingestion setup

3. Once it's done the user needs to finalize the collection. One can again add more collections and define multiple schemas which refer to columns of the original csv.

4. Once all schemas are made, one needs to hit finalize schema to send it to the server for processing. Upon successful processing, a user will be redirected to the query window.

5. Now in the query window, we have 2 options to add: either a select query or an aggregate query. Based on query options user has an option to fill out the choices provided to them. Upon hitting Submit, the server will respond once the query is processed. The user needs to refresh the terminal to see the output on it.

Implementational Aspects

Handling Data ingestion

The flow for data ingestion is as follows:

1. We ask the user to upload a dataset of his/her choice. The file is then sent to the backend where we create a database "testDB" and a collection "temporary" to store the dataset. The dataset is created by using a insertMany query where we pass the csv file contents.

2. Now that the baseline dataset is stored in mongoDB, we allow the user to specify the schema that he wants to create from the file that he has uploaded. This is done in the front-end. From the data that is received, for every schema that is specified, we create a new collection and we select those columns specified in the schema by using a find query. We rename the columns as specified by using an updateMany query.
The datatype of the fields are changed by using an aggregation query which uses the convert statement. This ensures

that the datatype of fields of the new collection follow the schema specified by the user.

Handling Null Replacement

We also provide the user options to deal with NULL values and all duplicates. The operation to be performed is sent as a request while creating the schema itself.

1. Once the data is inserted into the new collection, we call a *runConstraints* function to implement NULL checks and duplicate checks if specified by the user. Each operation is associated with a flag in the request sent from the front-end.

2. For every field in the collection, we check if there are flags for running constraint checks. We check for operations to remove NULL values, remove duplicates and replace NULL values with the mean of a field.

3. To remove NULL values, we perform a simple deleteMany opearti to delete those documents which have a NULL value corresponding to that field.

4. To replace NULL values with mean, we first run an aggregate query to find out the mean value of that field. Then we use an updateMany operation to update those documents with a null value in it with the average value found.

5. To remove all duplicates, we first perform an aggregate query to find the count of all values in the field and then store those documents with duplicates in a variable. Then we perform deletion using the delete query.

6. Upon successful schema creation, the schema along with the columns associated with it is passed back to the frontend where the user can specify queries that he wants to execute.

Upon finalizing the schema, the user is directed to a query window where he can perform selection or aggregation queries. The request from the front-end is checked to direct the request to the appropriate function to handle the query.

Handling Select Queries

Select queries are handled by *selectQuery* function. It uses a switch statement on the condition sent in the request. Based on the condition, we perform the following queries -

1. The = query: This simply runs a find operation to find those documents in the collection whose field value equals to the one that we are sending in the request. Conditions like $\geqq, \leqq, \leq, \geq$ *and* $\neq$ all follow the similar structure of the query execution.

2. To find > 75 percentile: This first finds the number of docs in the collection. Then sorts based on the column values in ascending order. Then it uses a find operation along with a skip option to skip the first 75 percent of the documents to give the result. The less than 25 percentile works with a similar logic.

Handling Aggregate Queries

Aggregate queries are handled by *aggregateQuery* function. It uses a switch statement on the condition sent in the request. Based on the condition, we perform the following queries -

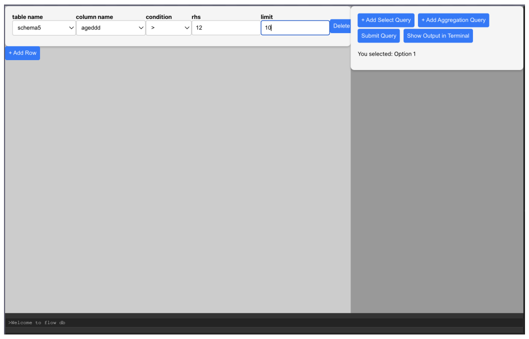
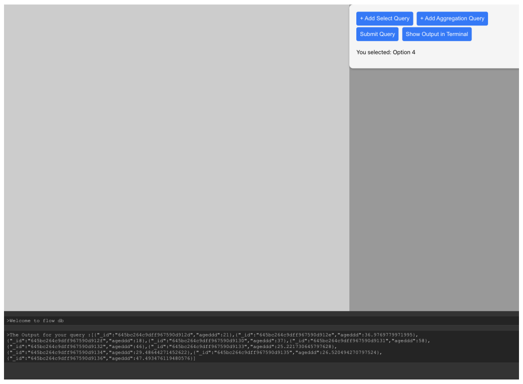


table name	column name	condition	rhs	limit
schema5	ageddd	>	12	10

You selected: Option 1

```
>Welcome to flow db
```

Figure 5. FlowDB: Data query setup



You selected: Option 4

```
>Welcome to flow db

>The Output for your query :[{"_id":"645bc264c9dff967590d912d","ageddd":21},{"_id":"645bc264c9dff967590d912e","ageddd":36.9769779971995},
{"_id":"645bc264c9dff967590d912f","ageddd":18},{"_id":"645bc264c9dff967590d9130","ageddd":37},{"_id":"645bc264c9dff967590d9131","ageddd":58},
{"_id":"645bc264c9dff967590d9132","ageddd":46},{"_id":"645bc264c9dff967590d9133","ageddd":25.221730645797628},
{"_id":"645bc264c9dff967590d9134","ageddd":29.48644271452622},{"_id":"645bc264c9dff967590d9135","ageddd":26.520494270797524},
{"_id":"645bc264c9dff967590d9136","ageddd":47.493476119480576}]
```

Figure 6. FlowDB: Data query output in terminal

1. Min condition - In this case, we simply run a find query over all documents and sort it in ascending order. We apply a limit of 1 to find the minimum value. The working of max is similar where we sort in descending order.

2. count condition - In this case, we run a count query which returns the count of all documents in the collection.

3. range condition - Here we run 2 find queries - one to find the min and one to find the max value in the collection. The result is pushed to an array which is returned.

4. average condition - We an aggregate query similar to the one we do to replace null with average value.

5. The result of each condition is stored in a variable and sent back as the response to the user.

Challenges Faced

1. The data that is loaded into mongoDB from the csv file is of string format. We had 2 options as a solution to this problem. First, we could ask the user to specify the csv format at the time of uploading. Then enforce the schema provided by the user at the time of schema specific collection creation. The second choice was to allow the user to not know much about the datatypes of the csv and upload the dataset into mongoDB and then when creating the collection, we would store values as per the user specified schema.
We choose to go ahead with the second choice as this allowed the user to not worry about the csv types and focus on how the data was being stored in the collection. It gives

the user more flexibility in terms of how he wants the data to be stored.

2. Another challenge that we faced was while designing how to give the user the choice to make a query. We handle multiple ways to do this. We decided to keep the range of choice limited to have a better handling of the queries in the backend. Whenever a schema was finalized, the backend sends a list of key value pairs of table name and column names. This is used to allow the user to make a query. We restricted ourselves to make only aggregation and some simple select query based on a column value. This can be extended in the future to handle complex nested queries.

EVALUATION

We evaluated our system made on the following dataset: Paris House Pricing. It consists of about 1,00,000 rows. To benchmark we sample multiple sizes and present the results here in this section. Our sample sizes consist of small (100 rows), medium (1000 rows), large(10,000 rows) and full which consist of the entire dataset. To introduce null values we randomly take 3 rows and 3 columns and replace its original value with null.

Ingestion Pipeline

For this setup, we run an ingestion query for 5 self-made columns in our own schema using the columns present in the dataset for various sizes of it as discussed above. Along with that, we try the different null replacement strategies i.e. do nothing, drop the row, replace with mean, and computed the time. The time is computed in milliseconds from when the submit button is clicked and a request is sent till the moment the response from the server with the status is received. The results are compiled in Table 1. We notice that a linear scale-up of time taken based on the size of the dataset which was an expected output.

Select and Aggregate Queries

Here we perform queries on a single column for various sizes of sample datasets as performed above. The queries consist of the following

1. Finding houses with >=100 square meters area
2. Finding the minimum-sized house
3. Finding the average of the sizes.
4. Finding all houses which have size > than the 75th percentile.

Do note to prevent flooding of the output panel we limited each of the outputs to finding only 50 such eligible entries. The time taken in milliseconds for these queries to execute can be found in Table 2. Here, one thing we noticed that the more time you spent querying on the same table, the faster it gets. Upon further research[3], we noticed that MongoDB cache frequently accessed data in memory, including the results of aggregate queries. The in-memory storage engine called the 'Wired Tiger Cache' is designed to provide faster access to frequently accessed data by keeping it in memory rather than reading it from disk. So to prevent this we had to flush our databases again and again per query.

LINKS

Our repository can be found here on Github

REFERENCES

[1] The OpenJS Foundation. 2023a. Express Documentation. (15 May 2023). from `https://expressjs.com/`.

[2] The OpenJS Foundation. 2023b. NodeJS Documentation. (15 May 2023). from `https://nodejs.org/en/docs`.

[3] MongoDB. 2023. Wired Tiger Cache. (15 May 2023). from `https://www.mongodb.com/docs/manual/core/wiredtiger/`.

[4] Scalegrid. 2019. "2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database Use". (4 March 2019). from `https://shorturl.at/cizMT`.

[5] Meta Open Source. 2023. React Documentation. (15 May 2023). from `https://react.dev/`.

Dataset Size	Normal ingestion	Null removed	Null replace with mean
Small (100)	111.2	156.2	176.2
Medium (1000)	253	255	259
Large (10,000)	748	750	776
Complete (1,00,000)	1908.8	1908.4	1914.8

Table 1. Ingestion time in milliseconds calculated from when the request is sent from the frontend to the time the frontend receives the response from the back-end

Dataset Size	Select Query with >=	MIN	AVERAGE	>75%
Small (100)	44	40.70000002	55	51.40000001
Medium (1000)	62.5	89.299	97	215.2
Large (10,000)	69.90000001	89.59999999	97.11	101.2
Complete (1,00,000)	71.222	93.73	98	140

Table 2. Query time in milliseconds calculated from when the request is sent from the frontend to the time the frontend receives the response from the back-end