University of North Dakota

SERVER SIDE PROJECT IMPLEMENTATION

Choice of Database management system - MySQL

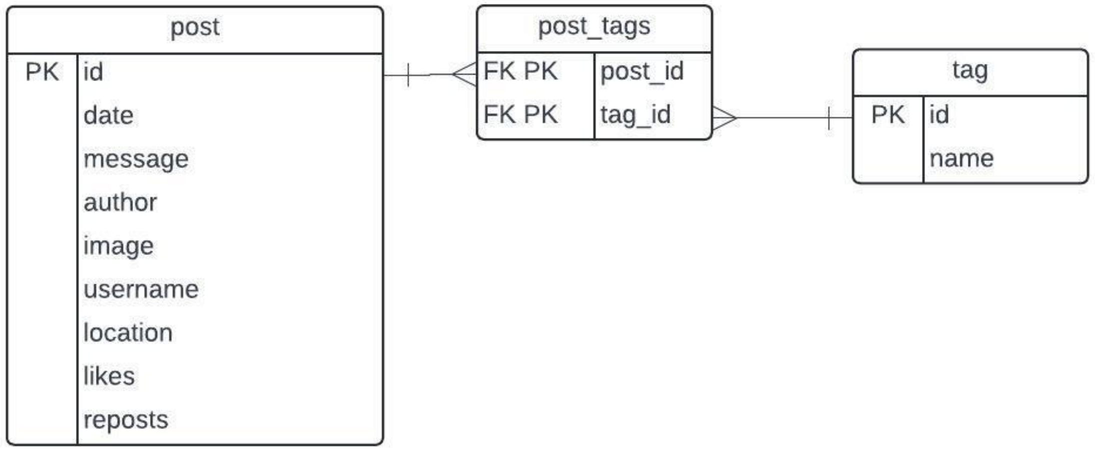
- While NoSQL databases can be a great choice for Blog applications, MySQL can be a great choice in RDBMS domain

  Why ?

- Easy to use
- Great community support
- Scalable
- Open source and free
- Great for servers

Database Entities

- `post`
- `tags`
- `post_tags`



INGESTING DATA PROCESS

- I used python to first create a database and make required tables and connect them with PK and FK relations
- Then I wrote ingestion script to fill the tables one by one in rounds of three (since only the data 3 days can be requested from the source API).
- In all I am dividing the arbitrarily large date range into chunks of 3 days and using source APIs and SQL queries to populate these tables and sending the appropriate data to its place in the DB
- I also made optimized ingestion command which we shall discuss in enhancements section

USING FLASK TO EXPOSE DATA THROUGH APIs

- Upon having my data ready in the database I used flask to process and expose data through REST API endpoints as instructed
- The reason I chose flask is to keep application easy and simple to implement and leverage python. The support for flask is also great along with great extensibility using third party libraries

Understanding the top post SQL query

I wanted MySQL server to compute whatever it can because they are fast in iterating compared to python[1]. For top_post function after researching on various possible SQL powers, I figured that ROW_NUMBER() OVER PARTITION BY can help me compute rankings of posts based on score for each month. Let us look at the query.

Query - Lets focus on inner query first

```
SELECT * FROM (SELECT *,

                    (reposts / 10 + likes / 15) AS score,

                        ROW_NUMBER() OVER (PARTITION BY DATE_FORMAT(date,
'%Y-%m') ORDER BY (reposts / 10 + likes / 15) DESC, date ASC) AS r

                    FROM post

                    WHERE YEAR(date) * 100 + MONTH(date) BETWEEN YEAR(%s) *
100 + MONTH(%s) AND YEAR(%s) * 100 + MONTH(%s)

                ) AS ranked_posts

                WHERE r <= 3

                ORDER BY DATE_FORMAT(date, '%Y-%m'), score DESC
```

Over clause with Partition by ?

It is used to combine non aggregated columns with aggregated columns because in group by we can not see non aggregated columns. In our query - ROW_NUMBER OVER PARTITION BY Works like this [2]. Row number clause seeks to apply within a range of rows defined by partition by and if it is not defined then it applied to all rows considering everything as a single big partition. Since we are partitioning by a month it ranks only within a month. For each month we order posts in descending order of scores(since more score means less row number) and if scores are same then earlier date gets better rank. Once we get the results from inner query we filter the results with appropriate rank and sorts them with month and then display them

Why did we not just add another case to where statement in the inner query to get the results?

```
SELECT *, (reposts / 10 + likes / 15) AS score,

ROW_NUMBER() OVER (PARTITION BY DATE_FORMAT(date, '%Y-%m') ORDER BY (reposts / 10 + likes / 15) DESC, date
ASC) AS r

FROM post

WHERE YEAR(date) * 100 + MONTH(date) BETWEEN YEAR('2023-01-01') * 100 + MONTH('2023-01-01') AND
YEAR('2023-02-05') * 100 + MONTH('2023-02-05') AND r<=3
```

Placing the condition r <= 3 directly in the inner query won't work because r is a computed column generated by the ROW_NUMBER() window function. In SQL, the WHERE clause filters rows based on the data present in each row. However, window functions like ROW_NUMBER() are computed after the WHERE clause is applied. This means that the column r does not yet exist at the time the WHERE clause is evaluated, so we cannot reference it directly in the WHERE clause of the same query level.[3]

Enhancements

- Optimized future ingestion commands
- JWT tokens
- Rate limiter
- SQL Injection by using parameterized queries and SQLALchemy
- Sorting results
- Listing end points on home url
- Validation for ingestion and API date range

Optimized Future INgestion

What would happen if someone tries to ingest data for the dates for which we already have data present?

Should we do repeat work?

Unless we want to refresh records, I do not think so.

```
Example: Old data    ____________

         New data --------------------
```

The highlighted section is the repeated work that our classic script would do. Why not just add new data

Why not protect API using temporary tokens

It would be a good idea to protect API endpoints using a randomly generated tokens for each username and password authenticated to use our API.

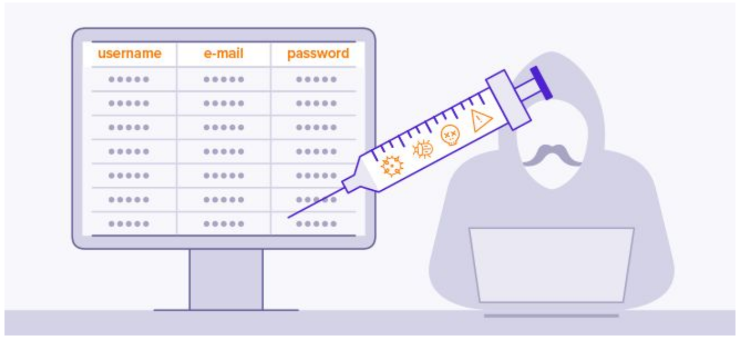
That would prevent unauthorized people to get access of our data. These easy to use tokens are called json web tokens(JWT). They get generated for a particular username by encoding using a complex algorithm, and then decoded and verified for that user for allowing access for a session.

Rate Limiter

Through rate limiter we can prevent someone accessing hitting our API relentlessly over a very short period of time. While someone may do it unknowingly there are huge chances of any script hitting end points just to increase costs and increasing the load of server. Rate limiting is very crucial especially while using cloud services like AWS, because you pay for what is being used.

SQL Injections

For endpoints requesting data from client before giving them results it is crucial to make sure that input is not a malicious SQL query which can break havoc in the database. We can either manually check for each keyword to make sure it not malicious (not including names like “SELECT FROM”, “DROP”, “--0 or 1”), or we can use third party libraries like SQLAlchemy.



Thank you & Questions

References

1. https://streamsets.com/blog/python-vs-sq
2. https://www.simplilearn.com/tutorials/sql-tutorial/row-number-funtion-in-sql
3. https://www.sisense.com/blog/sql-query-order-of-operations/
4.