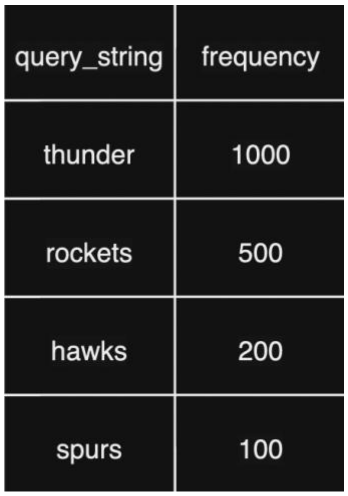
Autocomplete Feature Design

Tianye Wang

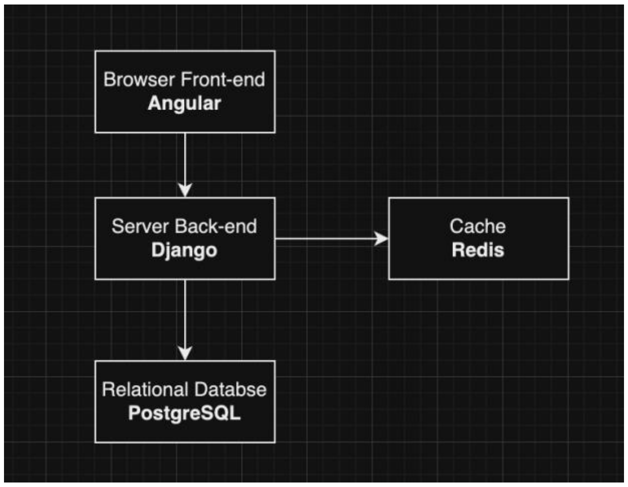
So at first glance, there are several features my autocomplete design could have. To display the relevant suggestions and show them in sequence, we could have a Record of common and popular search results stored in the database(such as PostgreSQL), each result string could have an "frequency" attribute that determines the display sequence.



query_string	frequency
thunder	1000
rockets	500
hawks	200
spurs	100

Minor considerations like first we could discard capitalization differences to reduce redundancies, thus could add relevant helper functions to convert/compare the result strings in lowercases. For updating the stored search results and their frequencies I don't think we need to update the data constantly or in real-time like social apps, we could use batch processing like most search engines, updating the database like once a week.

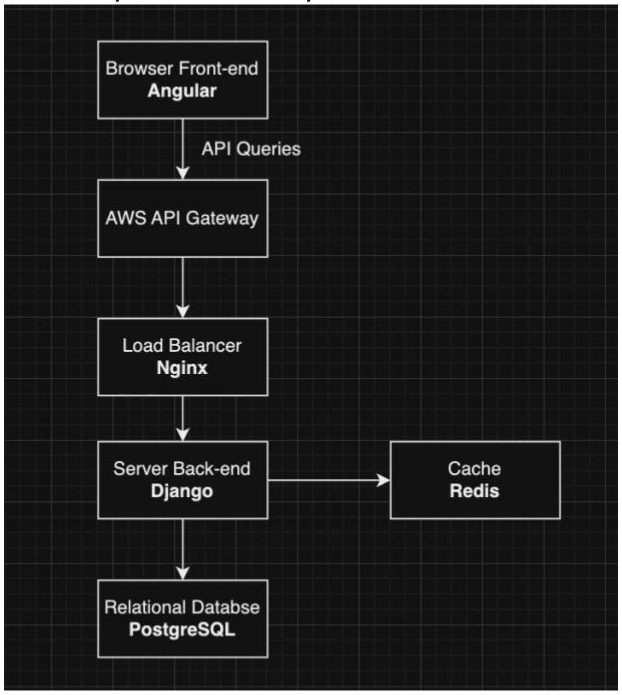
Another feature is to ensure quick responsiveness, suggestions should appear almost in real-time as the user inputs, with low latency. (I tested out myself searching google, and feel like ideally it should be below 200ms) We could use Caching here, In DB design class I just learned there is a NoSQL database called Redis that stores value to value relations and could provide almost real-time searching efficiency. We could use this as the cache layer.



So applying the above solutions, a complete search process would start at the design's client side:
At Client-Side:
- Wait for user input 2-4 characters or 50ms before querying the server (however, connection need be immediate after open the engine).
- Use API endpoints and send, for example, a GET request like: **/suggestions?q=<prefix>**

At Server-Side:
- First check the Redis cache for matching suggestions to the request.
- If no matches found in the cache, query the PostgreSQL database and update Redis with the new results for future requests. An example SQL statement could be:
SELECT * FROM search_history_table WHERE query_string ILIKE '<prefix>%'; (using ILIKE to perform case-insensitive searches).
- Sort the suggestions using the "frequency" attribute to ensure that common and popular search results are displayed first.

The above design is suitable for small scale of dataset and low traffic requests. ("<= 100 users simultaneously.") However, if the company wants to further develop and add more features, I researched and found an efficient data structure called the Trie Data Structure. Trie is a tree-like structure that can be generated from the current search results. Each node contains a letter, and by traversing the Trie, we can obtain all the suggestions efficiently. We could store the data in this structure and store it in the database. Besides that, to deal with growing requests and traffic we could deploy a Load-Balancer Layer such as Nginx, to distribute traffics. We could also apply an API Gateway Service such as AWS API Gateway to manage a larger number of incoming API requests. So to conclude we could add these new features to the autocomplete system in order to feature greater efficiency and scalability.


Browser Front-end
Angular
API Queries
AWS API Gateway
Load Balancer
Nginx
Server Back-end
Django
Cache
Redis
Relational Databse
PostgreSQL