Design of Yelp

Learn to fulfill the design requirements of the Yelp system.

We'll cover the following

We identified the requirements and calculated the estimations for our Yelp system in the previous lesson. In this lesson, we discuss the API design, go through the storage schema, and then dive into the details of the system's building blocks and additional components.

API design

Let's discuss the API design for Yelp.

Search

We need to implement the `search` function. The API call for searching based on categories like "cafes" will be:

```
search(category, user_location, radius)
```

Parameter	Description
`category`	This is the type of search the user makes—for example, restaurants, cinemas, cafes, and so on.
`user_location`	This contains the location of the user who's searching
`radius`	This is the specified radius where the user is trying to find category.

This process returns a JSON object that contains a list of all the possible items in the specified category that also fall within the specified radius. Each entry has a place name, address, category, rating, and thumbnail.

The API call for searching based on the name of a place like "Burger Hut" will be:

```
search(name_of_place, user_location, radius)
```

Parameter	Description
`name_of_place`	This contains the name of the place that the user wants t

This process returns a JSON object that contains information of the specified place.

Add a place

The API call for adding a place is below:

```
add_place(name_of_place, description_of_place, category, latitude, longitude, phot
o}
```

Parameter	Description
`name_of_place`	This contains the name of the place, for example, "Bu
`description_of_place`	This contains a description of the place. For example, "Burg yummiest burgers".
`category`	This specifies the category of the place—for exampl
`latitude`	This tells us the latitude of the place.
`longitude`	This tells us the longitude of the place.
`photo`	This contains photos of the place. There can be a single or

This process returns a response saying that a place has been added, or an appropriate error if it fails to add a place.

Add a review

The API call for adding a place is below:

```
add_review(place_ID, user_ID, review_description, rating)
```

Parameter	Description
`place_ID`	This contains the ID of the place whose review is a
`user_ID`	This contains the ID of the user who adds the
`review_description`	This contains the review of the place—for example, "the were superb".

`rating`	This contains the rating of the place—for example, 4

This process returns a response that a review has been added, or an appropriate error if it fails to add a review.

Storage schema

Let's define the storage schema for our system. A few of the tables we might need are "Places," "Photos," "Reviews," and "Users."

Let's define the columns of the "Place" table:

- **Place_ID**: We use the sequencer to generate an 8 Bytes (64 bits) unique ID for a place.

> **Note:** We generate IDs using the unique ID generator.

- **Name_of_Place**: This is a string that contains the name of the place. We use 256 Bytes for it.
- **Description_of_Place**: This holds a description of the place. We use 1,000 Bytes for it.
- **Category**: This specifies the type of place like restaurants, cinemas, bookshops, and so on (8 Bytes).
- **Latitude**: This stores the latitude of the location (8 Bytes).
- **Longitude**: This stores the longitude of the location (8 Bytes).
- **Photos**: This contains the foreign key (8 Bytes) of another table "Photos," which contains all the photos related to a particular place.
- **Rating**: This stores the rating of the place. It shows how many stars a place gets out of five. The rating is calculated based on the reviews it gets from th

users.

The columns mentioned above are the most important ones in the table. We can add more columns like “menu,” “address,” “opening and closing hours,” and so on. Therefore, keeping in mind the essential columns, the size of one row of our table will be:

```
Size = 8 + 256 + 1000 + 8 + 8 + 8 + 1 = 1289 bytes
```

Now let’s define the “Photos” table:

- **Photo_ID**: We use the sequencer to generate a unique ID for a photo (8 Bytes or 64 bits).
- **Place_ID**: We use the foreign key (8 Bytes) from the “Place” table to identify which photo belongs to which place.
- **Photo_path**: We store the photos in blob storage and save the photo’s path (256 Bytes) in this column.

```
Size = 8 + 8 + 8 + 256 = 280 bytes
```

We need another table called “Reviews” to store the reviews, ratings, and photos of a place.

- **Review_ID**: We use the sequencer to generate a unique ID of 8 Bytes (64 bits) for a review.
- **Place_ID**: The foreign key (8 Bytes) from the “Place” table to determine which place the rating belongs to.
- **User_ID**: The foreign key (8 Bytes) from the “Users” table to identify which review belongs to which user.
- **Review_description**: This holds a description of the review. We use 512 Bytes for it.
- **Rating**: This stores how many stars a place gets out of five (1 Byte).

```
Size = 8 + 8 + 8 + 512 + 1 = 537 bytes
```

We use the “Users” table to store user information.

- **User_ID**: We use the sequencer to generate a unique ID for a user (8 Bytes).
- **User_name**: This is a string that contains the user’s name. We use 256 Bytes for it.

```
Size = 8 + 256 = 264 bytes
```

> **Note:** The `INT` in the following schema contains an 8-Byte ID that we generate using the unique ID generator.

Storage schema

Design

Now we’ll discuss the individual building blocks and components used in the design of Yelp and how they work together to complete various functional requirements.

Components

These are the components of our system:

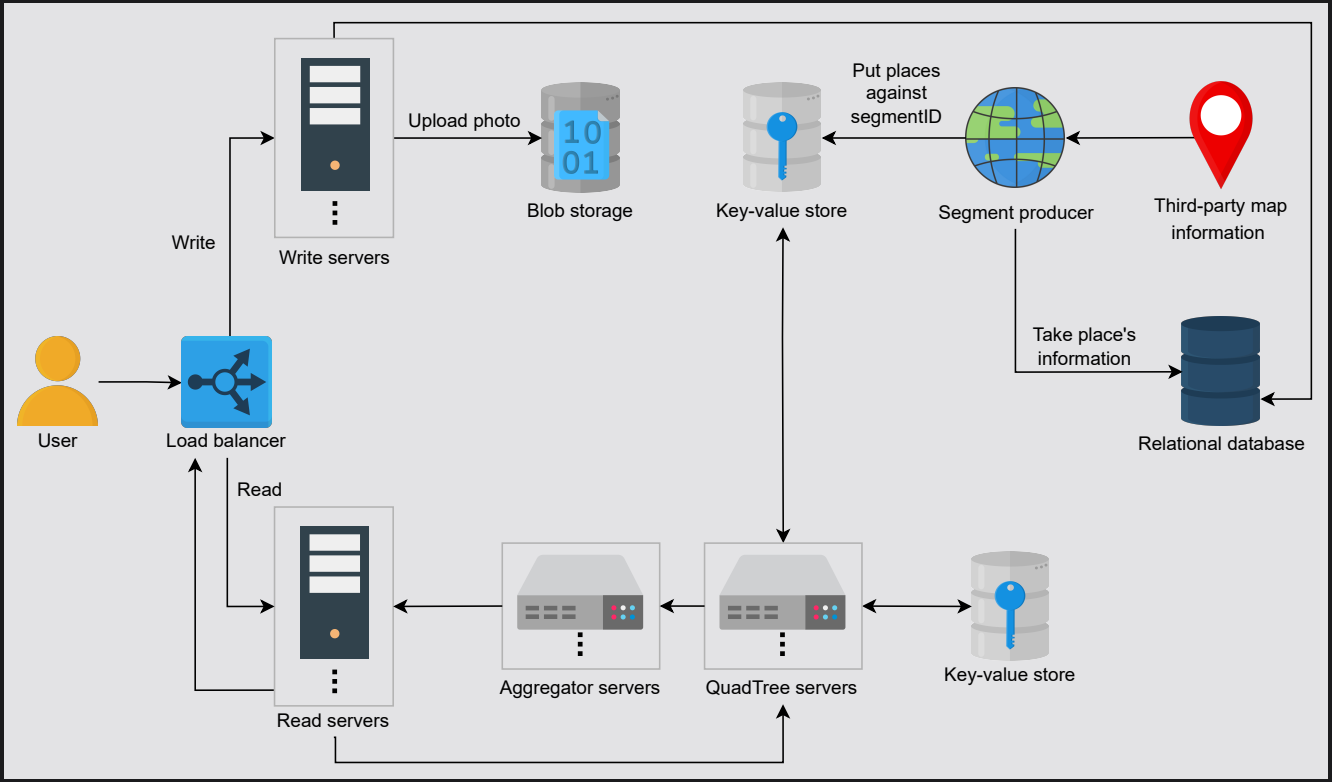
- **Segments producer**: This component is responsible for communicating with the third-party world map data services (for example, Google Maps). It takes up that data and divides the world into smaller regions called segments. The segment producer helps us narrow down the number of places to be searched.
- **QuadTree servers**: These are a set of servers that have trees that contain the places in the segments. A QuadTree server finds a list of places based on the given radius and the user's provided location and returns that list to the user. This component mainly aids the search functionality.
- **Aggregators**: The QuadTrees accumulate all the places and send them to the aggregators. Then, the aggregators aggregate the results and return the search result to the user.
- **Read servers**: We use a set of read servers that we use to handle all the read requests. Since we have more read requests, it's efficient to separate these requests from the write requests. Each read server directs the search requests to the QuadTrees' servers and returns the results to the user.

storage accordingly. Examples for write requests include adding a place, writing a comment, rating a place, and so on.

- **Storage**: We'll use two types of storage to fulfill our diverse needs.
    - **SQL database**: Our system will have different tables like "Users," "Place," "Reviews," "Photos," and others as described below. The data in these tables is inherently relational and structured. We need to perform queries like places a user visited, reviews they added, or view all the reviews of a specific place. It's easy to perform such queries in a SQL-based database. We also want all users to have a consistent view of th data, and SQL-based databases are better suited for such use cases. We'll use reliable and scalable databases, as is discussed in the Database building block.

- **Key-value stores**: We'll need to fetch the places in a segment efficiently. For that, we store the list of places against a segment ID in a key-value store to minimize searching time. We also save the QuadTree information in the key-value store, by storing the QuadTree data against a unique ID.

- **Load balancer**: A load balancer distributes users' incoming requests to all the servers uniformly.



Yelp design

Workflow

The user puts in a search request. We find all the relevant places in the given radius, while considering the user's location (latitude, longitude).

We explain the detailed workflow of our system in terms of the required functionalities below:

Searching a place: The load balancers route read requests to the read servers upon receiving them. The read servers direct them to the QuadTree servers to find

all the places that fall within the given radius. The QuadTree servers then send the results to the aggregators to refine them and send them to the user.

Adding a place or feedback: The load balancers route the write requests to the write servers upon receiving them. Depending on the provided content, meaning the place information or review, the write servers add an entry in the relational database and put all the related images in the blob storage.

Making segments: The segment's producer splits the world map taken from the third-party map service into smaller segments. The places inside each segment are stored in a key-value store. Even though this is a one-time job, this process is repeated periodically for newer segments and places. Since the probability of new places being added is low, we update our segments every month.

We've discussed the design of Yelp, its API design, and the relevant storage schema. In the next lesson, we'll talk about the design considerations.

Back

Requirements of Yelp's Design

Completed

Next

Design Considerations of Yelp