SWDV 691 Database Design Revision

Bryan Goggin

Overview

For my project, I will be using a non-SQL database. To make installation and integration simple with the rest of my AWS framework, I have chosen DynamoDB. The no-SQL style storage is the most natural choice for the data I will be storing due to the fact that it is hard to define a consistent schema for recipe data. Recipes tend to have varying numbers of ingredients, which makes this data a poor choice for a relational style database. By choosing DynamoDB, I am also eliminating the need to design complicated foreign key relationships and map out how my tables will be JOINed to give the users the information they seek.

My website will have 3 main tables: Recipes, Reviews and Users. The data structures will be no-SQL (sometimes referred to as JSON-style) and schema-less. I will use a composite key, which in DynamoDB terminology is called the **Partition Key** and the **Sort Key**, for my tables. These keys will refer to items in other tables in order to pull the necessary data to my website. Below I give examples of my data structures.

Table Design - Recipe

This table will store all of the recipes and be the major information store for my site. Here is the structure for a single entry. This table will have many entries

```
{
    RecipeName: str,
    RecipeID: str,
    Ingredients: {
        Ingredient1: str,
        Ingredient2: str,
        …
        IngredientN: str
    },
    Instructions: str,
    UploadedBy: str,
```

```
        UploadDate: Date,
        ImageURL: str
    }
```

The `Instructions` key will be a freeform text field where users submit the instructions for their recipe. The document type storage of DynamoDB is a more appropriate choice for text-based data that a relational database.

Table Design – Reviews

The Reviews table will store text data comprising the user reviews for a given recipe. This again makes the no-SQL document store an excellent choice. Below is an example.

```
{
        RecipeName: str,
        ReviewID: str,
        ReviewText: str,
        ReviewScore: float,
        ReviewedBy: str,
        ReviewDate: date
}
```

With this design, RecipeName will match the RecipeName in the Recipe table, but ReviewID will act as the Sort Key, which makes these entries unique. This allows the system to quickly find the reviews associated with a given recipe.

Table Design – Users

The Users table will store user information which the Recipe and Review tables will both reference. This table will be used to keep track of users with respect to who is uploading and leaving reviews.

```
    {
```

```
    UserName: "TacoTim9001",
    UserID: "serh4364",
    Recipes:["recipe22", "recipe32"],
    Reviews: []
}
```

In this table, Recipes and Reviews will be an array of RecipeIDs and ReviewIDs from the other tables that will allow the system to pull up this information quickly for each user.

Feedback – Support for Search

In version 1, search will be supported largely in the service layer and be very basic. In later iterations (stretch features) I plan to add "search by" options. To support this, the structure of the data in the Recipe table will change (new items in red):

```
{
    RecipeName: "Country Fried Steak",
    RecipeID: "abcd123",
    Ingredients: {
        "Steak": "1 lb.",
        "flour": "1/2 cup",
        …
        "pepper": "2 pinches"
    },
    Instructions: "Mix dry ingredients in a bowl, then in
    a separate bowl mix egg and milk, dip the steak in the
    egg batter, then in the dry ingredients, …"
    UploadedBy: "SteakMan55",
    UploadDate: "3MAR2021,
    ImageURL: "s3://images/country_fried_steak.png",
```

```
    Cuisine: ["traditional", "southern", "American"],
    Meal: ["lunch", "dinner"]

}
```

These new elements in the table will enable users to make specific searches. Note also the use of arrays in these new features. This structure will allow recipes to appear in multiple search options, which is an accurate model for my use case. The example of Country Fried Steak is an acceptable meal for both lunch and dinner.

Feedback – noSQL?

Absolutely. Note the `Instructions` key and its free form text value. Here the user will be able to explain how to make their recipe. This data structure is ill suited for a relational database design. Further, the design thus far has future features in the works. A "schema-less" approach will allow for easy upgrading, as there will be no strict requirement that data all have a consistent shape and structure. Again, this matches the reality of storing recipes as a document.

Feedback – Scalability?

A common (and valid) concern when using noSQL is the long term scalability of the storage system. This is important when it comes to search. With this in mind, the fundamental design of the system has the Recipe as a central component, meaning each recipe is indexed in the main table for quick retrieval. To enhance performance, an "indexer table" can be added in the future to take advantage of the indexable keys. Tools like Solr and Elasticsearch exist to support this, but the idea is simple. As a stretch feature, a table may be built in the follow way:

(Ingredient Index Table)

```
{
    IngredientName: "Cayenne Pepper",
    IngredientID: "s2La52bCP",
    UsedIn: ["abch123", "qwer4321", "35ab83ag25"]
}
```

Here, the `UsedIn` key contains all of the recipes which use the ingredient, in this case Cayenne Pepper. If a user searches for recipes containing Cayenne Pepper, that query can hit the Ingredient Index table, which will return an array of recipes which use that ingredient. A second step will pull those recipes from the Recipe table and display to the user. When the tables are large, this process will be much faster than scanning each item in the recipes table and looking for matches.

Feedback – Additional Tables and Stretch Features

A major advantage of noSQL is that tables are less dependent on each other, meaning updating the design requires less overall work. One stretch feature to consider may be a user group feature. Users may want to create a separate server for themselves so that their recipes are only shared to trusted members. A table to support this feature would look as follows:

(User Group Table)

```
{
      UserGroupName: "Very Average Bakers",
      UserGroupID: "aqqF42tHKe",
      Members: ["user1", "user2", "user10"],
      SharedRecipes: [
            "recipe1", "recipe35", "reciped2451235"
      ],
      GroupSettings: {
            Privacy: "InviteOnly",
            Notifications: "On"
      }
}
```

This table will integrate with the existing tables by matching group users to userIds in the User table, and shared recipes to recipeIds in the Recipes table.

Conclusion

After further reflection and analysis, I believe the database design is strong. The no-SQL technology will allow flexibility in the future as new features are added and the document style storage is a natural fit for this data structure. Using indexer tables will help mitigate performance concerns as the app grows and the use of recipe and users IDs to link the tables together makes up for the lack of traditional SQL JOIN operations. This revised design address feedback concerns and illustrates how new tables can be added for stretch features, as well as how new tables can be added to maintain app performance.