Homework 7 | NoSQL Design

Updates made to the assignment after its release are highlighted in red.

Objectives: To gain experience with designing schemas in NoSQL systems, and to contrast these systems with relational schema design.

Due date: Thu, May 30th @ 9pm

***Median completion time (23au):* 7 hours**

Introduction

Congratulations! You've been hired at Flightapp, Silicon Valley's hottest [unicorn](#), which sells tickets to *flights that have already departed*. Your employers were deeply impressed with your experience with the July 2015 flights dataset and are excited to have their newhire design a schema that will ✨*WOW!*✨ investors and secure their next billion dollars.

To help them choose a schema, they've asked you to write five "one-pagers" describing how to design Flightapp in the following:

- a relational database (eg, Microsoft SQLServer)
- a key-value database (eg, AWS DynamoDB)
- a document store (eg, Asterix)
- a graph database (eg, Neo4j)
- a wide column database (eg, Google Cloud Bigtable)

Homework Requirements

Describing a Data Model

For each NoSQL family, we have provided prompts/questions on Gradescope to get you pointed in the right direction; **do them first**. These prompts are not required, *not worth points*, and will not yield a fully-designed schema; they are, however, enough to guide you approximately halfway through the design. For the other half, you will provide **a few paragraphs of high-level description** and then answer a few questions (which *are* worth points) that assess your design's functionality.

The phrase "high level description" means that it is not necessary to give table creation statements or even to select a specific database management system. While it is tempting to get into the details of transaction handling or high performance writes, the goal of this assignment is to give a brief taste of NoSQL design. The sample solution is only 5 pages long (including rather large screenshots!) and only describes the rows and columns, plus a short paragraph about the trickier use-cases (eg, reserve). That's it. This spec and its associated documentation is longer than the sample solution!

Some students benefit from being able to create specific key stores/datasets/tables and test queries on them. For those students, we include a "recommended sandbox"; for students who do not find sandboxes helpful, you can safely ignore these recommendations. (note: the sample solution is written without the aid of a sandbox).

Most students find it helpful to see an example problem; to assist, we have released the Payroll/Registry/ParkingTickets example here.

Functional Specification

Flightapp consists of the following logical entities. These entities are *not necessarily database tables*; it is up to you to decide what entities to store persistently.

- **Flights / Carriers / Months / Weekdays**: modeled the same way as HWs 1-3. For this project you should consider these entities to be *read-only*. Also, we introduce the concept of a flight's *reservable capacity*; if a flight's capacity is *n* and there are *k* existing reservations for it (see below), then its *reservable capacity* is *n-k*.

- **Users**: A user has a username, password, and balance in their account. All usernames should be unique in the system. A user can have any number of reservations.

- **Itineraries**: An itinerary is *one or more* flights connecting an origin city to a destination city. Unlike HW3, we are not constrained to 3-hop itineraries – there can be an arbitrary number of intermediate cities – however, we define a **direct itinerary** as one that does not have any intermediate cities (in other words, an itinerary consisting of a single flight).

- **Reservations**: A reservation is similar to an itinerary in that it consists of one or more flights. Reservations are different in a few key aspects, however:

    - A reservation is a sequence of flights *associated with a specific user*
    - The existence of a reservation decrements the *reservable capacity* of each of its component flights. Using the previous example, if flight *f* has a reservable capacity of *n-k*, then only *n-k* future reservations which include *f* can be made. Please note that, unlike the rest of the flight attributes, *reservable capacity* is not read-only.

  Each reservation can be marked as either be *paid* or *unpaid*, and a reservation must have a unique ID associated with it.

In addition to these logical entities, Flightapp must support the following operations:

- **create(username, balance, password)** - creates a new user account with the specified parameters. Although we do not discuss password management in this class, we have released supplemental information about cryptographic hashing and password salting.

- **search(origin, dest, day_of_month, directonly)** - returns itineraries that match a user's requested parameters. If the user specifies **directonly**, only direct itineraries should be returned.

- **reserve($\mathbf{fid_1}$, …, $\mathbf{fid_n}$)** - reserves the specified flights. It should verify:
    - That all the component flights have remaining *reservable capacity*
    - That the user does not have another reservation on the same day

  Once these conditions have been verified, **reserve()** should decrement the flights' reservable capacity and then assign a unique ID for the reservation.

- **pay(reservation_id)** - decrements the associated user's balance and marks the reservation as paid. It is an error if the user attempts to pay for a reservation that they do not have sufficient funds for.

- **list_reservations(user_id)** - lists all reservations, both paid and unpaid, for the specified user.

Problem Set

Expected Query Patterns

Designing a NoSQL schema requires understanding the pattern of reads/writes to its data. You may make the following assumptions:

- Carrier, month, and weekday data will never change.
- Flight information will be updated very rarely (eg, once a week). Also, these updates will happen in bulk; there are no incremental changes to the flight data.
    - eg, we might replace the entire July 2015 dataset with August 2015's data.
- We expect users to be created rarely (eg, a few times a day), and we expect users to login at a moderate frequency (eg, a few times an hour).
- We expect users to reserve, pay, and view reservations at a moderate frequency (eg, a few times an hour)

- You can assume a moderate number of users (eg, N million, where N is a double-digit number) registered in your system.
- We expect users to search for itineraries quite frequently: several times a second when aggregated across our full userbase.

1. Relational Database

Present an ER diagram for Flightapp's logical entities. This diagram must be able to support all of Flightapp's use-cases; notably, if the Flights table is read-only, how might you represent (or dynamically calculate) a flight's reservable capacity?

Recommended Sandbox: You can use SQLite or your existing Azure SQLServer instance.

2. Key-Value Data Store

Describe how you would implement the flights application using a key-value data store. Your description should contain enough detail that we understand the key (or keys if you have multiple "types" of keys), any structure that you introduce into the key, and the structure of a key's value.

Recommended Sandbox: If you prefer to instantiate a specific database to test out your ideas or generate screenshots, we recommend AWS's DynamoDB; you can download their NoSQL Workbench for this purpose.

3. Document Store

Describe how you would implement the flights application using a document store database. Your description should contain enough detail that we understand all the types in your document.

Recommended Sandbox: If you prefer to instantiate a specific database to test out your ideas or generate screenshots, we recommend AsterixDB; download their software and start a single-node instance on your machine.

4. Graph

Describe how you would implement the flights application using a graph database. Your description should contain enough detail that we understand all the types of nodes and all the types of edges, and how they might be used to support Flightapp's operations.

Recommended Sandbox: If you prefer to instantiate a specific database to test out your ideas or generate screenshots, we recommend Neo4j; you can sign up for a free trial that is good for a week.

5. Wide Column Database

Describe how you would implement the flights application using a wide column database. Your description should contain enough detail that we understand the keyspace (eg, do you have different "types" of keys?), the column families and their contained columns, as well as whether you use explicit timestamps.

Recommended Sandbox: If you prefer to instantiate a specific database to test out your ideas or generate screenshots, we recommend Google Cloud Bigtable. Sign up for a free trial for *all* the Google Cloud products here, then create a "Cloud Bigtable" instance (depending on where you are in the UI, Cloud Bigtable is either a "Cloud and Storage" or a "Storage" product); once you have a Cloud Bigtable instance, create tables/colfamilies/rows using the cbt tool.

6. Feedback and Reflection

Lastly, please note that feedback is **required** for this assignment; it's been massively retooled for this quarter, and we appreciate your assistance in assessing the changes.

Submission

Please submit your .pdfs containing the text and all supporting documentation (eg, screenshots, code snippets), as well as your feedback and reflection answers, to Gradescope.

✨🥂✨ *Congratulations!* ✨🥂✨ You've finished the last homework for DATA 514 :)