Homework 7 - NoSQL Design - Jonathan Jacobs

Question 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.

ER Diagram Design for Relational Database (Flightapp)

Logical Entities and Relationships

1. **Flights**: Stores information about individual flights.
2. **Carriers**: Stores information about flight carriers.
3. **Months**: Stores information about months.
4. **Weekdays**: Stores information about weekdays.
5. **Users**: Stores user information including username, password, and account balance.
6. **Itineraries**: Represents one or more flights connecting an origin city to a destination city.
7. **Reservations**: Represents a user's reservation, consisting of one or more flights.

Operations Supported

- `create(username, balance, password)`
- `search(origin, dest, day_of_month, directonly)`
- `reserve(fid1, …, fidn)`
- `pay(reservation_id)`
- `list_reservations(user_id)`

Tables and Columns

- **Flights**
  - `flight_id` (PK)
  - `carrier_id` (FK)
  - `origin`
  - `destination`
  - `departure_time`
  - `arrival_time`
  - `capacity`
  - `reservable_capacity` (calculated dynamically based on reservations)
- **Carriers**
  - `carrier_id` (PK)
  - `carrier_name`
- **Months**
  - `month_id` (PK)
  - `month_name`

- **Weekdays**
  - `weekday_id` (PK)
  - `weekday_name`
- **Users**
  - `user_id` (PK)
  - `username` (unique)
  - `password`
  - `balance`
- **Itineraries**
  - `itinerary_id` (PK)
  - `origin`
  - `destination`
  - `is_direct` (boolean)
- **Itinerary_Flights** (link table for many-to-many relationship between Itineraries and Flights)
  - `itinerary_id` (FK)
  - `flight_id` (FK)
- **Reservations**
  - `reservation_id` (PK)
  - `user_id` (FK)
  - `is_paid` (boolean)
  - `reservation_date`
- **Reservation_Flights** (link table for many-to-many relationship between Reservations and Flights)
  - `reservation_id` (FK)
  - `flight_id` (FK)

Dynamic Calculation of Reservable Capacity

- Reservable capacity is calculated as `capacity - COUNT(reservation_id)` where `flight_id` is the same in the `Reservation_Flights` table.

Question 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.

Key-Value Data Store Design for Flightapp

Overview

In a key-value store like AWS DynamoDB, data is stored as a collection of key-value pairs. Each key is unique and maps to a value, which can be a complex data structure. The schema for Flightapp will be designed to efficiently support the expected query patterns and data access frequencies.

Keys and Values

We will define multiple types of keys to handle different entities and their relationships. The key structure and the corresponding value for each type of entity are as follows:

1. **Flights**
   - **Key**: `FLIGHT#<flight_id>`
   - **Value**:

     ```
     {
       "carrier_id": "<carrier_id>",
       "origin": "<origin>",
       "destination": "<destination>",
       "departure_time": "<departure_time>",
       "arrival_time": "<arrival_time>",
       "capacity": "<capacity>",
       "reservable_capacity": "<reservable_capacity>"
     }
     ```

2. **Carriers**
   - **Key**: `CARRIER#<carrier_id>`
   - **Value**:

     ```
     {
       "carrier_name": "<carrier_name>"
     }
     ```

3. **Months**
   - **Key**: `MONTH#<month_id>`
   - **Value**:

     ```
     {
       "month_name": "<month_name>"
     }
     ```

4. **Weekdays**
   - **Key**: `WEEKDAY#<weekday_id>`
   - **Value**:

     ```
     {
       "weekday_name": "<weekday_name>"
     }
     ```

5. **Users**
   - **Key**: `USER#<user_id>`

   - **Value**:

     ```
     {
       "username": "<username>",
       "password": "<password>",
       "balance": "<balance>"
     }
     ```

6. **Itineraries**
   - **Key**: `ITINERARY#<itinerary_id>`
   - **Value**:

     ```
     {
       "origin": "<origin>",
       "destination": "<destination>",
       "is_direct": "<is_direct>",
       "flights": [
         {
           "flight_id": "<flight_id>",
           "departure_time": "<departure_time>",
           "arrival_time": "<arrival_time>"
         },
         ...
       ]
     }
     ```

7. **Reservations**
   - **Key**: `RESERVATION#<reservation_id>`
   - **Value**:

     ```
     {
       "user_id": "<user_id>",
       "is_paid": "<is_paid>",
       "reservation_date": "<reservation_date>",
       "flights": [
         {
           "flight_id": "<flight_id>"
         },
         ...
       ]
     }
     ```

Expected Query Patterns

1. **Create User**
   - **Operation**: PutItem

   - **Key**: `USER#<user_id>`
   - **Value**: User details
2. **Search Itineraries**
   - **Operation**: Query
   - **Key Pattern**: `ITINERARY#<origin>#<destination>#<date>`
   - **Value**: List of itineraries for the given origin, destination, and date.
3. **Reserve Flights**
   - **Operation**: Transaction (PutItem and UpdateItem)
   - **Key**: `RESERVATION#<reservation_id>`
   - **Value**: Reservation details
   - **Additional Operation**: Update `reservable_capacity` in the corresponding flights
4. **Pay for Reservation**
   - **Operation**: UpdateItem
   - **Key**: `RESERVATION#<reservation_id>`
   - **Update**: Set `is_paid` to true and decrement user balance
5. **List Reservations**
   - **Operation**: Query
   - **Key Pattern**: `USER#<user_id>#RESERVATION#*`
   - **Value**: List of reservations for the user

Implementation Notes

- **Flight reservable capacity**: Dynamically updated by decrementing the `reservable_capacity` in the flight's value each time a reservation is made.
- **User balance management**: Ensured by updating the user balance during the `pay` operation.
- **Indexing**: Appropriate secondary indexes can be created for frequent query patterns, such as searching itineraries. This design leverages the flexibility and scalability of key-value stores to handle the varied and frequent queries expected in Flightapp's operations.

Question 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.

Document Store Design for Flightapp

Overview

In a document store like AsterixDB, data is stored as JSON-like documents, allowing for nested structures and complex data types. Each document is a self-contained unit of data that can include nested arrays and objects, making it suitable for the flexible and hierarchical data model required by Flightapp.

Document Types

1. **Flight Document**
   - **Key**: `flight_id`

- **Structure**:

  ```
  {
    "flight_id": "<flight_id>",
    "carrier": {
      "carrier_id": "<carrier_id>",
      "carrier_name": "<carrier_name>"
    },
    "origin": "<origin>",
    "destination": "<destination>",
    "departure_time": "<departure_time>",
    "arrival_time": "<arrival_time>",
    "capacity": "<capacity>",
    "reservable_capacity": "<reservable_capacity>"
  }
  ```

2. **User Document**
   - **Key**: `user_id`
   - **Structure**:

     ```
     {
       "user_id": "<user_id>",
       "username": "<username>",
       "password": "<password>",
       "balance": "<balance>",
       "reservations": [
         {
           "reservation_id": "<reservation_id>",
           "is_paid": "<is_paid>",
           "reservation_date": "<reservation_date>",
           "flights": [
             {
               "flight_id": "<flight_id>",
               "origin": "<origin>",
               "destination": "<destination>",
               "departure_time": "<departure_time>",
               "arrival_time": "<arrival_time>"
             },
             ...
           ]
         }
         ...
       ]
     }
     ```

3. **Itinerary Document**
   - **Key**: `itinerary_id`
   - **Structure**:

```
{
  "itinerary_id": "<itinerary_id>",
  "origin": "<origin>",
  "destination": "<destination>",
  "is_direct": "<is_direct>",
  "flights": [
    {
      "flight_id": "<flight_id>",
      "departure_time": "<departure_time>",
      "arrival_time": "<arrival_time>"
    },
    ...
  ]
}
```

Expected Query Patterns

1. **Create User**
   - **Operation**: Insert
   - **Document**: User document with user details
2. **Search Itineraries**
   - **Operation**: Query
   - **Filter**: Documents with matching `origin`, `destination`, and optionally `is_direct`
   - **Output**: List of itinerary documents
3. **Reserve Flights**
   - **Operation**: Update (Transaction)
   - **Document**: Update user document to include new reservation
   - **Additional Operation**: Update `reservable_capacity` in flight documents
4. **Pay for Reservation**
   - **Operation**: Update
   - **Document**: Update reservation in user document to mark `is_paid` and decrement user balance
5. **List Reservations**
   - **Operation**: Query
   - **Filter**: User document by `user_id`
   - **Output**: List of reservations from the user's document

Example Documents

Flight Document Example

```
{
  "flight_id": "FL123",
  "carrier": {
    "carrier_id": "C1",
    "carrier_name": "Airline A"
  },
  "origin": "City A",
  "destination": "City B",
  "departure_time": "2024-05-25T08:00:00Z",
  "arrival_time": "2024-05-25T10:00:00Z",
  "capacity": 100,
  "reservable_capacity": 90
}
```

User Document Example

```
  {
    "user_id": "U123",
    "username": "john_doe",
    "password": "hashed_password",
    "balance": 200.00,
    "reservations": [
      {
        "reservation_id": "R123",
        "is_paid": false,
        "reservation_date": "2024-05-25T09:00:00Z",
        "flights": [
          {
            "flight_id": "FL123",
            "origin": "City A",
            "destination": "City B",
            "departure_time": "2024-05-25T08:00:00Z",
            "arrival_time": "2024-05-25T10:00:00Z"
          }
        ]
      }
    ]
  }
```

Itinerary Document Example

```
  {
    "itinerary_id": "I123",
    "origin": "City A",
    "destination": "City C",
    "is_direct": false,
    "flights": [
      {
        "flight_id": "FL123",
        "departure_time": "2024-05-25T08:00:00Z",
        "arrival_time": "2024-05-25T10:00:00Z"
      },
      {
        "flight_id": "FL456",
        "departure_time": "2024-05-25T11:00:00Z",
        "arrival_time": "2024-05-25T13:00:00Z"
      }
    ]
  }
```

Implementation Notes

- **Reservable Capacity Management**: When a reservation is made, the corresponding flight documents need to be updated to decrement the `reservable_capacity`.
- **User Reservations**: User documents embed reservations, which include detailed flight information to minimize the need for additional queries.
- **Indexing**: Utilize indexes on common query fields such as `origin`, `destination`, and `username` to optimize search and access patterns. This document-oriented approach leverages the flexibility of document stores to manage hierarchical data structures and efficiently handle the diverse query patterns required by Flightapp.

Question 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.

Graph Database Design for Flightapp

Overview

In a graph database like Neo4j, data is represented as nodes (entities) and edges (relationships) that connect them. This model is well-suited for representing complex relationships and querying interconnected data efficiently.

Node and Edge Types

1. **Nodes**
   - **User**
     - Properties: `user_id`, `username`, `password`, `balance`
   - **Flight**
     - Properties: `flight_id`, `carrier_id`, `origin`, `destination`, `departure_time`, `arrival_time`, `capacity`, `reservable_capacity`
   - **Carrier**
     - Properties: `carrier_id`, `carrier_name`
   - **Itinerary**
     - Properties: `itinerary_id`, `origin`, `destination`, `is_direct`
   - **Reservation**
     - Properties: `reservation_id`, `is_paid`, `reservation_date`
2. **Edges**

  - **User RESERVES Reservation**
  - **Reservation CONTAINS Flight**
  - **Itinerary INCLUDES Flight**
  - **Flight OPERATED_BY Carrier**

Operations and Queries

1. **Create User**
   - **Operation**: Create a `User` node
   - **Data Type**:

     ```
     {
       "user_id": "U123",
       "username": "john_doe",
       "password": "hashed_password",
       "balance": 200.00
     }
     ```

2. **Search Itineraries**
   - **Operation**: Query for `Itinerary` nodes based on origin, destination, and direct flag
   - **Data Type**:

     ```
     {
       "origin": "City A",
       "destination": "City B",
       "is_direct": true
     }
     ```

3. **Reserve Flights**
   - **Operation**: Create a `Reservation` node, link it to `User` and `Flight` nodes
   - **Data Type**:

     ```
     {
       "reservation_id": "R123",
       "is_paid": false,
       "reservation_date": "2024-05-25T09:00:00Z",
       "flights": [
         {
           "flight_id": "FL123",
           "origin": "City A",
           "destination": "City B",
           "departure_time": "2024-05-25T08:00:00Z",
           "arrival_time": "2024-05-25T10:00:00Z"
         }
       ]
     }
     ```

4. **Pay for Reservation**
    - **Operation**: Update `Reservation` node to mark it as paid, decrement user balance
    - **Data Type**:

    ```json
    {
      "reservation_id": "R123",
      "is_paid": true,
      "user_balance": 180.00
    }
    ```

5. **List Reservations**
    - **Operation**: Query for `Reservation` nodes linked to a `User`
    - **Data Type**:

    ```json
    {
      "user_id": "U123"
    }
    ```

Example JSON Data for Nodes and Edges

User Node Example

```json
{
  "user_id": "U123",
  "username": "john_doe",
  "password": "hashed_password",
  "balance": 200.00
}
```

Flight Node Example

```json
{
  "flight_id": "FL123",
  "carrier_id": "C1",
  "origin": "City A",
  "destination": "City B",
  "departure_time": "2024-05-25T08:00:00Z",
  "arrival_time": "2024-05-25T10:00:00Z",
  "capacity": 100,
  "reservable_capacity": 90
}
```

Carrier Node Example

```
{
  "carrier_id": "C1",
  "carrier_name": "Airline A"
}
```

Reservation Node Example

```
{
  "reservation_id": "R123",
  "is_paid": false,
  "reservation_date": "2024-05-25T09:00:00Z",
  "flights": [
    {
      "flight_id": "FL123",
      "origin": "City A",
      "destination": "City B",
      "departure_time": "2024-05-25T08:00:00Z",
      "arrival_time": "2024-05-25T10:00:00Z"
    }
  ]
}
```

Itinerary Node Example

```
{
  "itinerary_id": "I123",
  "origin": "City A",
  "destination": "City C",
  "is_direct": false,
  "flights": [
    {
      "flight_id": "FL123",
      "departure_time": "2024-05-25T08:00:00Z",
      "arrival_time": "2024-05-25T10:00:00Z"
    },
    {
      "flight_id": "FL456",
      "departure_time": "2024-05-25T11:00:00Z",
      "arrival_time": "2024-05-25T13:00:00Z"
    }
  ]
}
```

Implementation Notes

- **Reservable Capacity Management:** Managed through update operations within transactions to ensure atomicity.

- **User and Reservations**: Direct relationships between users and reservations make it easy to list and manage reservations.
- **Indexing and Performance**: Utilize graph database indexing features for common query fields to optimize performance. This graph-based approach allows for efficient querying of interconnected data, which is ideal for the complex relationships and frequent searches expected in Flightapp's operations.

Question 5 - Wide Column Database

Wide Column Database Design for Flightapp

Overview

A wide column database like Google Cloud Bigtable stores data in tables with rows and dynamic columns grouped into column families. This model allows for efficient read and write operations on large datasets, making it suitable for handling the high frequency of search queries and updates expected in Flightapp.

Keyspace and Column Families

1. **Keyspace**
   - Primary key structure is crucial for performance. We will use composite keys to uniquely identify rows.
2. **Column Families**
   - **Flights**: Stores flight information.
   - **Users**: Stores user information.
   - **Itineraries**: Stores itinerary information.
   - **Reservations**: Stores reservation details.

Table and Column Family Structure

1. **Flights Table**
   - **Row Key**: `flight_id`
   - **Column Family**: `details`
     - Columns: `carrier_id`, `origin`, `destination`, `departure_time`, `arrival_time`, `capacity`, `reservable_capacity` Example:

   ```
   Row Key: FL123
   Column Family: details
   Columns:
     carrier_id: C1
     origin: City A
     destination: City B
     departure_time: 2024-05-25T08:00:00Z
     arrival_time: 2024-05-25T10:00:00Z
     capacity: 100
     reservable_capacity: 90
   ```

2. **Users Table**

- **Row Key**: `user_id`
- **Column Family**: `profile`
  - Columns: `username`, `password`, `balance` Example:

```
Row Key: U123
Column Family: profile
Columns:
  username: john_doe
  password: hashed_password
  balance: 200.00
```

3. **Itineraries Table**
   - **Row Key**: `itinerary_id`
   - **Column Family**: `details`
     - Columns: `origin`, `destination`, `is_direct`, `flights` (serialized array of flight ids) Example:

```
Row Key: I123
Column Family: details
Columns:
  origin: City A
  destination: City C
  is_direct: false
  flights: [FL123, FL456]
```

4. **Reservations Table**
   - **Row Key**: `reservation_id`
   - **Column Family**: `details`
     - Columns: `user_id`, `is_paid`, `reservation_date`, `flights` (serialized array of flight ids) Example:

```
Row Key: R123
Column Family: details
Columns:
  user_id: U123
  is_paid: false
  reservation_date: 2024-05-25T09:00:00Z
  flights: [FL123]
```

Operations and Queries

1. **Create User**
   - **Operation**: Insert into `Users` table
   - **Command**:

```
cbt set user_id:U123 profile:username=john_doe
profile:password=hashed_password profile:balance=200.00
```

2. **Search Itineraries**
   - **Operation**: Query `Itineraries` table
   - **Command**:

   ```
   cbt read Itineraries prefix=CityA#CityB
   ```

3. **Reserve Flights**
   - **Operation**: Insert into `Reservations` table, update `Flights` table to decrement `reservable_capacity`
   - **Command**:

   ```
   cbt set reservation_id:R123 details:user_id=U123
   details:is_paid=false details:reservation_date=2024-05-
   25T09:00:00Z details:flights=[FL123]
   cbt set flight_id:FL123 details:reservable_capacity=89
   ```

4. **Pay for Reservation**
   - **Operation**: Update `Reservations` table to mark as paid, update `Users` table to decrement balance
   - **Command**:

   ```
   cbt set reservation_id:R123 details:is_paid=true
   cbt set user_id:U123 profile:balance=180.00
   ```

5. **List Reservations**
   - **Operation**: Query `Reservations` table by `user_id`
   - **Command**:

   ```
   cbt read Reservations prefix=U123
   ```

Explicit Timestamps

- Explicit timestamps are typically used for versioning data. For Flightapp, we assume the default timestamp behavior provided by Bigtable is sufficient, unless specific version control is required for operations like auditing or historical data analysis.

Implementation Notes

- **Reservable Capacity Management**: Updated directly in the `Flights` table during reservation operations.
- **Serialized Arrays**: Flights in itineraries and reservations are stored as serialized arrays to simplify data retrieval and updates.
- **Indexing**: Use composite row keys to support efficient queries, such as `origin#destination` for itineraries. This wide column database design leverages the flexibility and scalability of Google Cloud Bigtable to handle the diverse and high-frequency data operations required by Flightapp.

Question 6 - Feedback and Reflection

While I'm not sure what the feedback and reflection sections were like in past years, I believe that incorporating quantitative questions such as time spent, perceived value, and collaboration is significantly more informative in assessing students' learning experiences compared to instructors' initial expectations. Qualitative feedback is crucial for adding nuance, but it remains influenced by the reader's perception. Metrics like time provide a clear, objective comparison, allowing instructors to better understand the actual dynamics and whether their goals are being met.

That said, I found the qualitative questions to be very thoughtful and conducive to introspection about the homework itself.