Desnormalización en Apache Cassandra

Objetivo

Comprender el concepto de tablas con denormalización en Apache Cassandra, su uso práctico, ventajas, desventajas y cómo diseñar esquemas optimizados. Utilizaremos el caso de gestión de películas de Netflix como ejemplo para estructurar y denormalizar datos.

Introducción a la Denormalización

1. **¿Qué es la Denormalización?**
   - En Cassandra, se modelan los datos para las **consultas específicas** que se desean realizar, en lugar de un diseño normalizado como en bases relacionales.
   - Los datos se replican en varias tablas para mejorar la velocidad de consulta, sacrificando almacenamiento.
2. **Ventajas**:
   - Consultas rápidas y eficientes.
   - Reducción de JOINs (no soportados en Cassandra).
   - Diseño óptimo para patrones de acceso.
3. **Desventajas**:
   - Mayor uso de almacenamiento.
   - Riesgo de inconsistencia si los datos no se actualizan correctamente.

Caso de Uso: Gestión de Películas en Netflix

En este caso, queremos gestionar información sobre las películas, géneros, actores y reseñas en Netflix. Diseñaremos varias tablas denormalizadas para responder a diferentes tipos de consultas.

Tabla 1: Información de películas por género

1. **¿Qué consulta queremos responder?**
   - *Obtener todas las películas de un género específico.*
2. **Diseño de la tabla**:

```
CREATE TABLE movies_by_genre (
    genre TEXT,
    movie_id UUID,
    title TEXT,
    release_year INT,
    PRIMARY KEY (genre, movie_id)
) WITH CLUSTERING ORDER BY (movie_id ASC);
```

3. **Ejercicio 1: Insertar películas en la tabla**:

```
INSERT INTO movies_by_genre (genre, movie_id, title,
release_year)
VALUES ('Action', uuid(), 'Mad Max: Fury Road', 2015);

INSERT INTO movies_by_genre (genre, movie_id, title,
release_year)
VALUES ('Drama', uuid(), 'The Godfather', 1972);
```

4. **Ejercicio 2: Consultar todas las películas de un género**:

```
SELECT * FROM movies_by_genre WHERE genre = 'Action';
```

Tabla 2: Actores principales por película

1. **¿Qué consulta queremos responder?**
   - *Obtener los actores principales de una película específica.*
2. **Diseño de la tabla**:

```
CREATE TABLE actors_by_movie (
    movie_id UUID,
    actor_name TEXT,
    role TEXT,
    PRIMARY KEY (movie_id, actor_name)
);
```

3. **Ejercicio 3: Insertar actores en la tabla**:

```
INSERT INTO actors_by_movie (movie_id, actor_name, role)
VALUES (uuid(), 'Tom Hardy', 'Max Rockatansky');

INSERT INTO actors_by_movie (movie_id, actor_name, role)
VALUES (uuid(), 'Charlize Theron', 'Imperator Furiosa');
```

4. **Ejercicio 4: Consultar actores de una película**:

```
SELECT * FROM actors_by_movie WHERE movie_id = <movie_id>;
```

Tabla 3: Películas por actor

1. **¿Qué consulta queremos responder?**
   - *Obtener todas las películas en las que ha trabajado un actor.*
2. **Diseño de la tabla**:

```
CREATE TABLE movies_by_actor (
    actor_name TEXT,
    movie_id UUID,
    title TEXT,
    release_year INT,
    PRIMARY KEY (actor_name, movie_id)
) WITH CLUSTERING ORDER BY (movie_id ASC);
```

3. **Ejercicio 5: Insertar películas asociadas a un actor**:

```
INSERT INTO movies_by_actor (actor_name, movie_id, title,
release_year)
VALUES ('Tom Hardy', uuid(), 'Mad Max: Fury Road', 2015);

INSERT INTO movies_by_actor (actor_name, movie_id, title,
release_year)
VALUES ('Tom Hardy', uuid(), 'Inception', 2010);
```

4. **Ejercicio 6: Consultar películas de un actor**:

```
SELECT * FROM movies_by_actor WHERE actor_name = 'Tom Hardy';
```

Tabla 4: Reseñas por película

1. **¿Qué consulta queremos responder?**
   - *Obtener todas las reseñas asociadas a una película.*
2. **Diseño de la tabla**:

```
CREATE TABLE reviews_by_movie (
    movie_id UUID,
    review_id UUID,
    user_id UUID,
    review_text TEXT,
    rating INT,
    PRIMARY KEY (movie_id, review_id)
) WITH CLUSTERING ORDER BY (review_id ASC);
```

3. **Ejercicio 7: Insertar reseñas en la tabla**:

```
INSERT INTO reviews_by_movie (movie_id, review_id, user_id,
review_text, rating)
VALUES (uuid(), uuid(), uuid(), 'Amazing movie!', 5);

INSERT INTO reviews_by_movie (movie_id, review_id, user_id,
review_text, rating)
VALUES (uuid(), uuid(), uuid(), 'Great visuals but weak story',
3);
```

4. **Ejercicio 8: Consultar reseñas de una película**:

```
SELECT * FROM reviews_by_movie WHERE movie_id = <movie_id>;
```

Tabla 5: Películas populares por año

1. **¿Qué consulta queremos responder?**
   - *Obtener las películas más populares de un año específico.*
2. **Diseño de la tabla**:

```
CREATE TABLE popular_movies_by_year (
    release_year INT,
    movie_id UUID,
    title TEXT,
    rating_avg FLOAT,
    PRIMARY KEY (release_year, movie_id)
) WITH CLUSTERING ORDER BY (rating_avg DESC);
```

3. **Ejercicio 9: Insertar películas populares**:

```
INSERT INTO popular_movies_by_year (release_year, movie_id,
title, rating_avg)
VALUES (2015, uuid(), 'Mad Max: Fury Road', 4.7);

INSERT INTO popular_movies_by_year (release_year, movie_id,
title, rating_avg)
VALUES (1972, uuid(), 'The Godfather', 4.9);
```

4. **Ejercicio 10: Consultar películas populares de un año**:

```
SELECT * FROM popular_movies_by_year WHERE release_year = 2015;
```

En los siguientes esquemas,

```
CREATE KEYSPACE hotel WITH replication =
  {'class': 'SimpleStrategy', 'replication_factor' : 3};

CREATE TYPE hotel.address (
  street text,
  city text,
  state_or_province text,
  postal_code text,
  country text );

CREATE TABLE hotel.hotels_by_poi (
  poi_name text,
  hotel_id text,
  name text,
  phone text,
  address frozen<address>,
  PRIMARY KEY ((poi_name), hotel_id) )
  WITH comment = 'Q1. Find hotels near given poi'
  AND CLUSTERING ORDER BY (hotel_id ASC);

CREATE TABLE hotel.hotels (
  id text PRIMARY KEY,
  name text,
  phone text,
  address frozen<address>,
  pois set<text> )
  WITH comment = 'Q2. Find information about a hotel';

CREATE TABLE hotel.pois_by_hotel (
  poi_name text,
  hotel_id text,
  description text,
  PRIMARY KEY ((hotel_id), poi_name) )
  WITH comment = 'Q3. Find pois near a hotel';

CREATE TABLE hotel.available_rooms_by_hotel_date (
  hotel_id text,
  date date,
  room_number smallint,
  is_available boolean,
  PRIMARY KEY ((hotel_id), date, room_number) )
  WITH comment = 'Q4. Find available rooms by hotel date';

CREATE TABLE hotel.amenities_by_room (
  hotel_id text,
  room_number smallint,
  amenity_name text,
  description text,
  PRIMARY KEY ((hotel_id, room_number), amenity_name) )
  WITH comment = 'Q5. Find amenities for a room';
```

```
CREATE KEYSPACE reservation WITH replication = {'class':
  'SimpleStrategy', 'replication_factor' : 3};

CREATE TYPE reservation.address (
  street text,
  city text,
  state_or_province text,
  postal_code text,
  country text );

CREATE TABLE reservation.reservations_by_confirmation (
  confirm_number text,
  hotel_id text,
  start_date date,
  end_date date,
  room_number smallint,
  guest_id uuid,
  PRIMARY KEY (confirm_number) )
  WITH comment = 'Q6. Find reservations by confirmation number';

CREATE TABLE reservation.reservations_by_hotel_date (
  hotel_id text,
  start_date date,
  end_date date,
  room_number smallint,
  confirm_number text,
  guest_id uuid,
  PRIMARY KEY ((hotel_id, start_date), room_number) )
  WITH comment = 'Q7. Find reservations by hotel and date';

CREATE TABLE reservation.reservations_by_guest (
  guest_last_name text,
  hotel_id text,
  start_date date,
  end_date date,
  room_number smallint,
  confirm_number text,
  guest_id uuid,
  PRIMARY KEY ((guest_last_name), hotel_id) )
  WITH comment = 'Q8. Find reservations by guest name';

CREATE TABLE reservation.guests (
  guest_id uuid PRIMARY KEY,
  first_name text,
  last_name text,
  title text,
  emails set,
  phone_numbers list,
  addresses map<text,
  frozen<address>,
  confirm_number text )
  WITH comment = 'Q9. Find guest by ID';
```