Documentacion: Implementacion del Modelo Estrella

Fabian Fernandez, Diego Castro, Andres Corrales

October 14, 2024

1 Introduccion

Este documento proporciona una descripcion detallada de la implementacion de un esquema en estrella para el analisis de los datos de alquiler de peliculas utilizando PostgreSQL. El objetivo es crear un modelo dimensional que permita realizar consultas eficientes sobre las transacciones de alquiler para la generacion de informes y analisis. Este modelo incluye medidas como el numero de alquileres y el monto total recaudado, asi como dimensiones como peliculas, fechas de alquiler, direcciones y tiendas.

Las siguientes secciones describen cada paso del proceso, desde la creacion del esquema hasta la carga de datos y su validacion.

2 Diseno del Esquema

El esquema en estrella consiste en una tabla de hechos que almacena las metricas clave (numero de alquileres y total de ingresos) y varias tablas de dimensiones que contienen informacion descriptiva (peliculas, fechas de alquiler, direcciones y tiendas).

2.1 Tabla de Hechos: `fact_rentals`

La tabla `fact_rentals` almacena las metricas principales como el numero de alquileres y los ingresos totales. Se relaciona con las tablas de dimensiones mediante claves foraneas.

```
CREATE TABLE fact_rentals (
    rental_id SERIAL PRIMARY KEY,
    film_id INT REFERENCES dim_film(film_id),
    address_id INT REFERENCES dim_address(address_id),
    store_id INT REFERENCES dim_store(store_id),
    rental_date DATE REFERENCES dim_rental_date(rental_date),
    rental_count INT,               -- Medida: Numero de alquileres
    total_amount NUMERIC(10, 2)     -- Medida: Ingresos totales por alquiler
);
```

2.2 Tablas de Dimensiones

Las tablas de dimensiones describen los atributos relacionados con cada medida en la tabla de hechos.

2.2.1 Tabla: `dim_film`

Esta tabla almacena informacion sobre las peliculas, incluidos sus titulos, categorias y actores.

```
CREATE TABLE dim_film (
    film_id INT PRIMARY KEY,
    title VARCHAR(255),
    category VARCHAR(100),
    actor_name VARCHAR(255)
);
```

2.2.2 Tabla: `dim_address`

La tabla `dim_address` almacena informacion sobre la jerarquia de ubicaciones, incluyendo pais y ciudad.

```
CREATE TABLE dim_address (
    address_id INT PRIMARY KEY,
    country VARCHAR(100),
    city VARCHAR(100)
);
```

2.2.3 Tabla: `dim_rental_date`

Esta tabla representa la jerarquia de fechas de alquiler, con los campos ano, mes y dia.

```
CREATE TABLE dim_rental_date (
    rental_date DATE PRIMARY KEY,
    year INT,
    month INT,
    day INT
);
```

2.2.4 Tabla: `dim_store`

La tabla `dim_store` almacena informacion sobre las tiendas donde se realizan los alquileres.

```
CREATE TABLE dim_store (
    store_id INT PRIMARY KEY,
    store_name VARCHAR(100)
);
```

3 Procedimientos Almacenados

Se utilizan procedimientos almacenados para cargar datos en las tablas de hechos y dimensiones desde la base de datos transaccional. Tambien implementamos estrategias para evitar la duplicacion de datos.

3.1 Procedimiento: `load_fact_rentals`

Este procedimiento agrega datos de alquiler de las tablas transaccionales y los inserta en la tabla `fact_rentals`. Antes de insertar, se eliminan los registros existentes para asegurar la consistencia de los datos.

```
CREATE OR REPLACE FUNCTION load_fact_rentals()
RETURNS VOID AS $$
BEGIN
    -- Eliminar registros existentes para evitar duplicacion
    DELETE FROM fact_rentals;

    -- Insertar nuevos datos de alquiler agregados
    INSERT INTO fact_rentals (film_id, address_id, store_id, rental_date,
        rental_count, total_amount)
    SELECT
        i.film_id,                    -- ID de la pelicula desde la tabla '
            inventory'
        a.address_id,                 -- ID de la direccion desde la tabla '
            address'
        s.store_id,                   -- ID de la tienda desde la tabla 'store'
        r.rental_date,                -- Fecha de alquiler desde la tabla 'rental'
        COUNT(r.rental_id) AS rental_count,    -- Conteo de alquileres
        SUM(p.amount) AS total_amount         -- Suma del total de ingresos
    FROM rental r
    JOIN payment p ON r.rental_id = p.rental_id        -- Unir con 'payment' para
        obtener el monto total
    JOIN customer c ON r.customer_id = c.customer_id   -- Unir con 'customer' para
         obtener la direccion
    JOIN address a ON c.address_id = a.address_id      -- Unir con 'address' para
        obtener la ubicacion
    JOIN inventory i ON r.inventory_id = i.inventory_id -- Unir con 'inventory'
        para obtener el film_id
    JOIN store s ON i.store_id = s.store_id            -- Unir con 'store' para
        obtener los detalles de la tienda
    GROUP BY i.film_id, a.address_id, s.store_id, r.rental_date;  -- Agrupar por
        dimensiones
END;
$$ LANGUAGE plpgsql;
```

3.2 Procedimiento: `load_dim_film`

Este procedimiento carga la tabla **dim_film** con datos de peliculas, incluidos el titulo, la categoria y los actores. La clausula `ON CONFLICT DO NOTHING` asegura que no se inserten duplicados.

```
CREATE OR REPLACE FUNCTION load_dim_film()
RETURNS VOID AS $$
BEGIN
    -- Insertar peliculas unicas junto con sus categorias y actores
    INSERT INTO dim_film (film_id, title, category, actor_name)
    SELECT
        f.film_id,                                  -- ID de la pelicula desde la
            tabla 'film'
        f.title,                                    -- Titulo desde la tabla 'film'
        COALESCE(c.name, 'Unknown') AS category,    -- Categoria desde 'category', o
             'Unknown' si falta
        COALESCE(STRING_AGG(a.first_name || ' ' || a.last_name, ', '), 'No actors
            listed') AS actor_name -- Actores o por defecto
    FROM film f
    LEFT JOIN film_category fc ON f.film_id = fc.film_id     -- Unir con
        film_category
    LEFT JOIN category c ON fc.category_id = c.category_id   -- Unir con category
    LEFT JOIN film_actor fa ON f.film_id = fa.film_id        -- Unir con
        film_actor
    LEFT JOIN actor a ON fa.actor_id = a.actor_id            -- Unir con actor
    GROUP BY f.film_id, f.title, c.name
    ON CONFLICT (film_id) DO NOTHING;  -- Evitar duplicados
END;
$$ LANGUAGE plpgsql;
```

3.3 Procedimiento: `load_dim_address`

Este procedimiento llena la tabla **dim_address** con datos de ubicacion, incluidos ciudad y pais. La clausula `ON CONFLICT` evita la insercion de entradas duplicadas.

```
CREATE OR REPLACE FUNCTION load_dim_address()
RETURNS VOID AS $$
BEGIN
    -- Insertar direcciones unicas con su ciudad y pais
    INSERT INTO dim_address (address_id, city, country)
    SELECT
        a.address_id,               -- ID de direccion desde la tabla 'address'
        ci.city,                    -- Ciudad desde la tabla 'city'
        co.country                  -- Pais desde la tabla 'country'
    FROM address a
    JOIN city ci ON a.city_id = ci.city_id          -- Unir con city
    JOIN country co ON ci.country_id = co.country_id -- Unir con country
    ON CONFLICT (address_id) DO NOTHING;  -- Evitar duplicados
END;
$$ LANGUAGE plpgsql;
```

3.4 Procedimiento: `load_dim_rental_date`

Este procedimiento carga la tabla `dim_rental_date` con informacion de fechas de alquiler. Los duplicados se evitan utilizando `ON CONFLICT`.

```
CREATE OR REPLACE FUNCTION load_dim_rental_date()
RETURNS VOID AS $$
BEGIN
    -- Insertar fechas de alquiler unicas en dim_rental_date
    INSERT INTO dim_rental_date (rental_date, year, month, day)
    SELECT
        DISTINCT r.rental_date,            -- Fecha de alquiler desde la tabla '
            rental'
        EXTRACT(YEAR FROM r.rental_date) AS year,   -- Extraer el ano desde la
            fecha de alquiler
        EXTRACT(MONTH FROM r.rental_date) AS month, -- Extraer el mes desde la
            fecha de alquiler
        EXTRACT(DAY FROM r.rental_date) AS day      -- Extraer el dia desde la
            fecha de alquiler
    FROM rental r
    ON CONFLICT (rental_date) DO NOTHING;  -- Evitar duplicados
END;
$$ LANGUAGE plpgsql;
```

3.5 Procedimiento: `load_dim_store`

Este procedimiento carga los datos de tiendas en la tabla `dim_store`, evitando entradas duplicadas.

```
CREATE OR REPLACE FUNCTION load_dim_store()
RETURNS VOID AS $$
BEGIN
    -- Insertar tiendas unicas en dim_store con sus nombres
    INSERT INTO dim_store (store_id, store_name)
    SELECT
        s.store_id,                    -- ID de tienda desde la tabla 'store'
        'Store ' || s.store_id AS store_name  -- Generar nombre de tienda
            dinamicamente
    FROM store s
    ON CONFLICT (store_id) DO NOTHING;  -- Evitar duplicados
END;
$$ LANGUAGE plpgsql;
```

3.6 Procedimiento: load_star_schema

Este procedimiento ejecuta todos los procedimientos de carga individuales para poblar el esquema estrella. Primero se cargan las tablas de dimensiones, seguidas de la tabla de hechos.

```
CREATE OR REPLACE FUNCTION load_star_schema()
RETURNS VOID AS $$
BEGIN
    -- Cargar las tablas de dimensiones primero
    PERFORM load_dim_film();
    PERFORM load_dim_address();
    PERFORM load_dim_rental_date();
    PERFORM load_dim_store();

    -- Luego cargar la tabla de hechos
    PERFORM load_fact_rentals();
END;
$$ LANGUAGE plpgsql;
```

4 Pruebas y Validacion

Para verificar que el esquema se ha poblado correctamente, se pueden utilizar las siguientes consultas SQL:

```sql
-- Verificar los datos en dim_film
SELECT * FROM dim_film LIMIT 10;

-- Verificar los datos en dim_address
SELECT * FROM dim_address LIMIT 10;

-- Verificar los datos en dim_rental_date
SELECT * FROM dim_rental_date LIMIT 10;

-- Verificar los datos en dim_store
SELECT * FROM dim_store LIMIT 10;

-- Verificar los datos en fact_rentals
SELECT * FROM fact_rentals LIMIT 10;

-- Datos agregados por pelicula
SELECT film_id, SUM(rental_count) AS total_rentals, SUM(total_amount) AS
    total_revenue
FROM fact_rentals
GROUP BY film_id
LIMIT 10;

-- Datos agregados por tienda
SELECT store_id, SUM(rental_count) AS total_rentals, SUM(total_amount) AS
    total_revenue
FROM fact_rentals
GROUP BY store_id
LIMIT 10;

-- Datos agregados por ano y mes
SELECT EXTRACT(YEAR FROM rental_date) AS year, EXTRACT(MONTH FROM rental_date) AS
    month,
        SUM(rental_count) AS total_rentals, SUM(total_amount) AS total_revenue
FROM fact_rentals
GROUP BY year, month
LIMIT 10;
```

5 Resume

La implementación de un esquema estrella para el análisis de datos de alquileres de películas en PostgreSQL ha demostrado ser una solución eficaz para organizar y estructurar grandes volúmenes de información. Este modelo permite transformar los datos transaccionales en un formato más accesible y eficiente para el análisis. A través de la separación de los datos en tablas de hechos y dimensiones, hemos logrado simplificar la manera en que se almacenan y consultan las métricas clave, como el número de alquileres y los ingresos totales.

Las tablas de dimensiones proporcionan información detallada sobre las características de las entidades relevantes, como las películas, las tiendas, las ubicaciones geográficas y las fechas de alquiler. Este diseño mejora considerablemente la flexibilidad al realizar análisis de diferentes niveles de agregación, como por película, por ubicación o por fecha. Al centralizar las medidas en la tabla de hechos, es posible realizar cálculos y generar reportes de manera más coherente y rápida.

Además, se han implementado mecanismos para asegurar la integridad y consistencia de los datos, evitando la duplicación de registros mediante el uso de la cláusula `ON CONFLICT DO NOTHING` en las inserciones. Esta técnica permite que las tablas puedan actualizarse de manera segura, incluso si los procedimientos de carga se ejecutan varias veces. A su vez, el uso de procedimientos almacenados facilita la automatización del proceso de carga, asegurando que el esquema estrella se mantenga actualizado a medida que los datos transaccionales evolucionan.

Otro beneficio importante de este diseño es la capacidad de mantener un bajo acoplamiento entre las tablas de hechos y las tablas de dimensiones, lo que facilita la actualización y mantenimiento del esquema. Las tablas de dimensiones pueden actualizarse de manera independiente de la tabla de hechos, permitiendo que los datos de soporte se enriquezcan sin afectar el almacenamiento de las medidas.

Este enfoque ofrece una estructura escalable y bien organizada para almacenar y consultar grandes volúmenes de datos. Gracias a la organización en torno a medidas clave y dimensiones descriptivas, esta implementación no solo permite un análisis eficiente, sino que también proporciona una base sólida para futuras ampliaciones, adaptándose a nuevos requerimientos o cambios en la fuente de datos original.