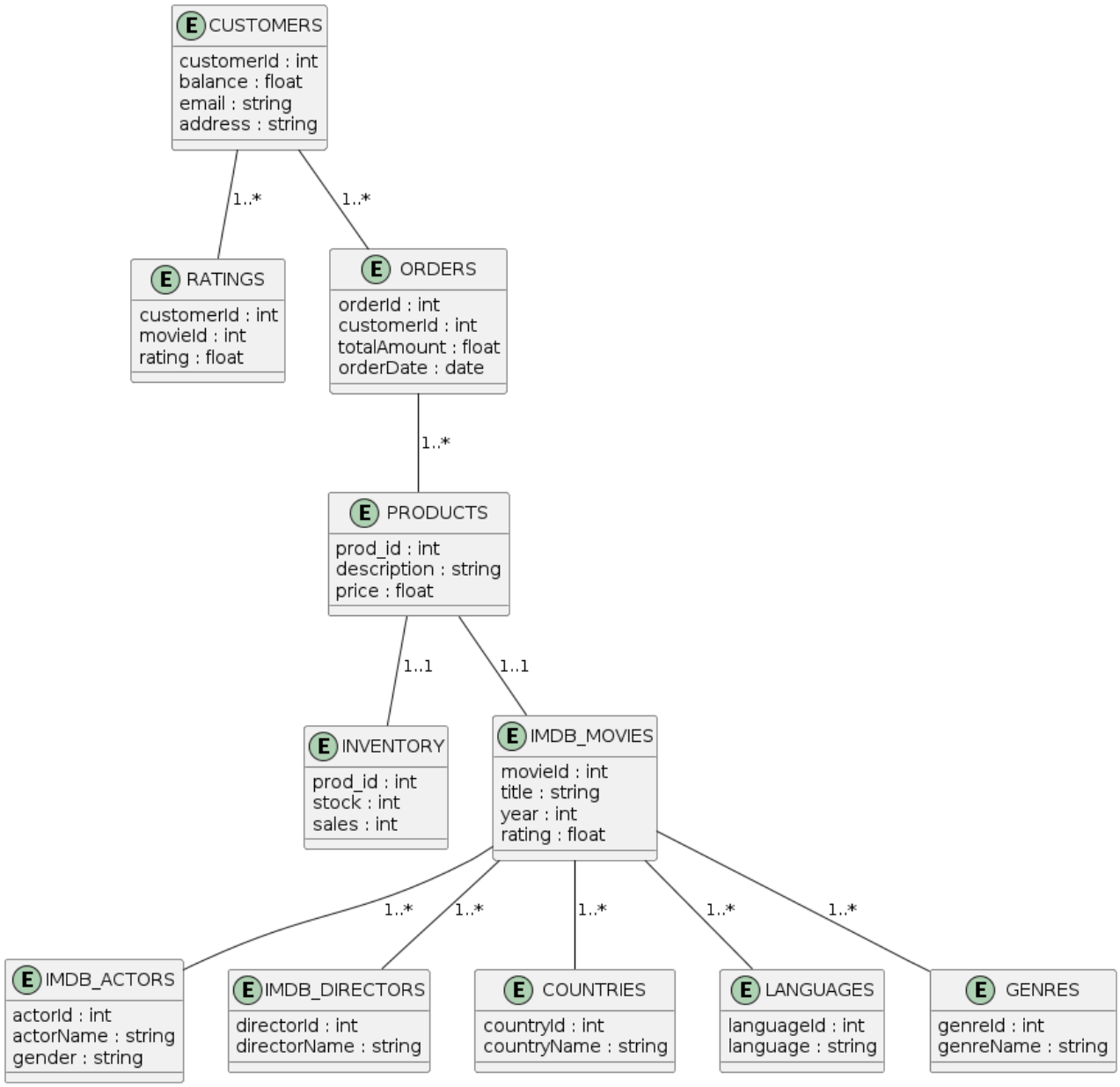
Memoria Sistemas Informáticos Práctica 2 por Bernardo Andrés Zambrano Ferreira

1. Diseño de la BD

1.1 Ingeniería inversa y script `actualiza.sql`

- Obtener el Diagrama Entidad-Relación correspondiente.



Claves Primarias y Foráneas:

Claves Primarias (Primary Keys):

1. **Unicidad**:
    - Las claves primarias aseguran que cada fila en una tabla sea única. Por ejemplo, `customerid` en la tabla `customers` garantiza que no haya duplicados, manteniendo la integridad de los

datos y facilitando la identificación de cada registro.

```
ALTER TABLE public.customers
    ADD CONSTRAINT customers_pk PRIMARY KEY (customerid);
```

2. **Eficiencia en la Búsqueda**:
    - Definir una clave primaria crea automáticamente un índice en esa columna, mejorando significativamente la eficiencia de las consultas. Buscar un cliente por `customerid` será mucho más rápido debido a este índice.

```
ALTER TABLE public.orders
    ADD CONSTRAINT orders_pk PRIMARY KEY (orderid);
```

Claves Foráneas (Foreign Keys):

1. **Integridad Referencial**:
    - Las claves foráneas garantizan que los valores en una columna correspondan a valores en la columna primaria de otra tabla. Por ejemplo, `customerid` en `orders` debe existir en `customers`, evitando referencias inválidas.

```
ALTER TABLE public.orders
    ADD CONSTRAINT orders_fk_customerid FOREIGN KEY (customerid)
REFERENCES public.customers (customerid) ON DELETE CASCADE;
```

2. **Acciones en Cascada**:
    - La opción `ON DELETE CASCADE` asegura que si se elimina un registro en la tabla primaria, todos los registros relacionados en la tabla secundaria también se eliminarán, manteniendo la base de datos limpia y libre de registros huérfanos.

```
ALTER TABLE public.orderdetail
    ADD CONSTRAINT orderdetail_fk_orderid FOREIGN KEY (orderid)
REFERENCES public.orders (orderid) ON DELETE CASCADE;
```

3. **Normalización y Organización de Datos**:
    - Las claves foráneas ayudan a mantener una estructura de datos lógica y bien organizada, promoviendo la normalización y reduciendo la redundancia.

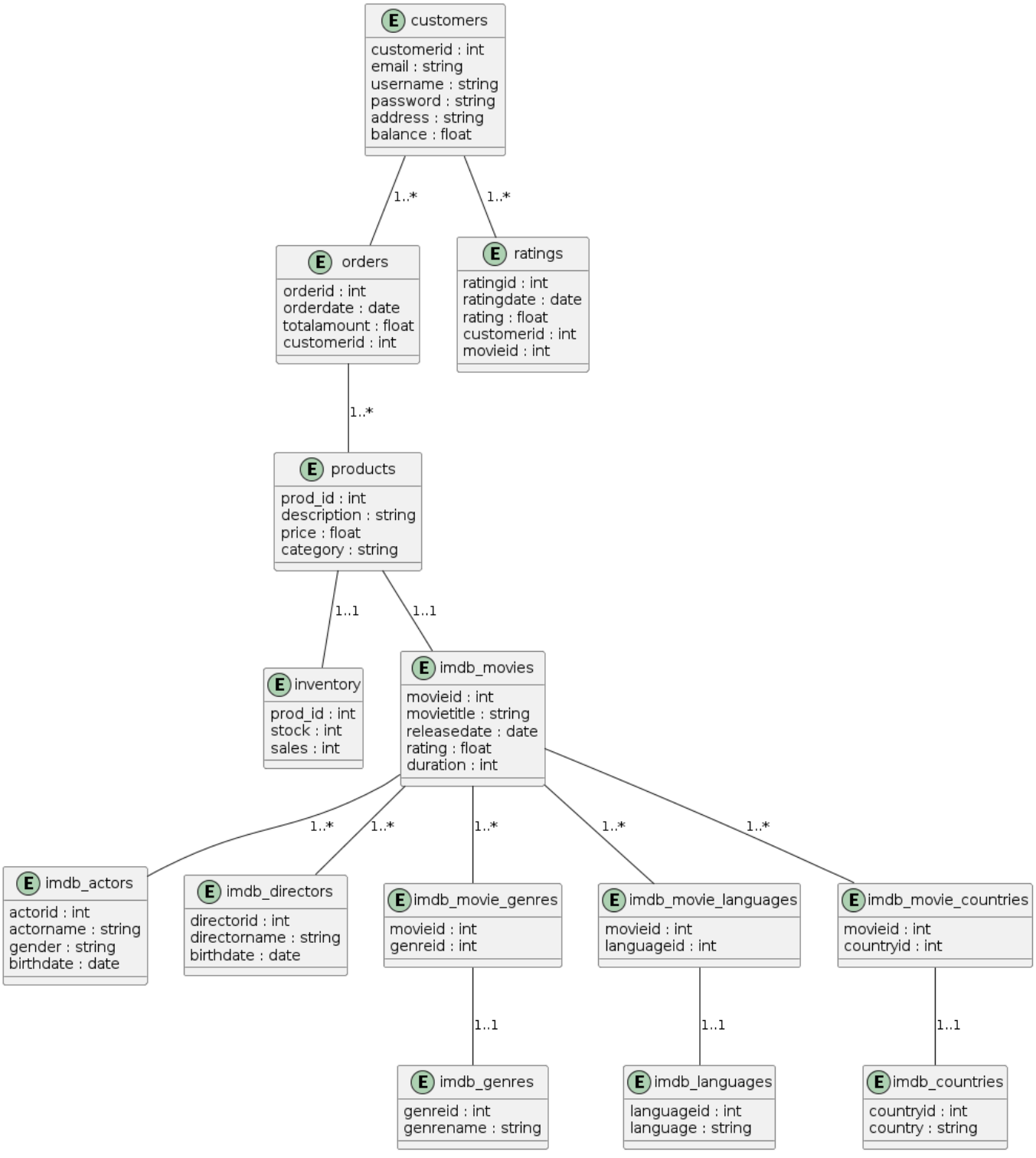
Resto de las Decisiones:

Uso de Transacciones:

- Agrupar operaciones en transacciones garantiza que todas las operaciones se completen de manera exitosa o ninguna lo haga, evitando estados intermedios inconsistentes en la base de datos.

```
BEGIN;
-- operaciones
COMMIT;
```

Diagrama tras actualiza



1.2 Consulta `setPrice.sql`

1. **Encapsulación en Transacción**: Utilizar un bloque anónimo permite gestionar las operaciones de manera atómica y segura.
2. **Declaración de Variables**: Se declara `annual_increase` para definir el porcentaje de incremento anual, facilitando futuras modificaciones sin cambiar múltiples líneas de código.
3. **Cálculo del Incremento Anual**: La función `POWER` calcula el nuevo precio multiplicando el precio actual por `(1 + annual_increase)` elevado al número de años transcurridos desde la fecha del pedido.
4. **Precisión Financiera**: La función `ROUND` asegura que los nuevos precios se redondeen a dos decimales, manteniendo la precisión necesaria para operaciones financieras.
5. **Subconsulta Correlacionada**: Esta subconsulta dentro del `UPDATE` permite acceder a los precios actuales de los productos y las fechas de los pedidos, asegurando que los datos utilizados sean correctos y actualizados.
6. **Integridad Referencial**: Las uniones (`JOIN`) entre `orderdetail`, `products` y `orders` aseguran que las relaciones entre las tablas se mantengan válidas y que los datos sean consistentes.
7. **Conversión de Tipos**: Convertir el resultado a `numeric` garantiza la consistencia del tipo de dato para la columna `price`.

Código:

```
DO $$
DECLARE
    annual_increase NUMERIC := 0.02;
BEGIN
    -- Actualizar la columna price de la tabla orderdetail
    UPDATE public.orderdetail AS od
    SET price = ROUND(
        (
            SELECT p.price * POWER(1 + annual_increase, EXTRACT(YEAR FROM
current_date) - EXTRACT(YEAR FROM o.orderdate))
            FROM public.products AS p
            JOIN public.orders AS o ON od.orderid = o.orderid
            WHERE od.prod_id = p.prod_id
        )::numeric, 2);
END $$;
```

1.3 Procedimiento almacenado `setOrderAmount`

Explicación Resumida:

Este procedimiento almacenado `setOrderAmount` actualiza los precios en `orderdetail` y calcula los importes netos y totales en `orders`, aplicando un incremento anual del 2%.

1. **Procedimiento Almacenado**: Encapsula la lógica compleja de actualización, facilitando la reutilización y el mantenimiento.
2. **Actualización de Precios**: Usa `POWER` para calcular el incremento anual del 2% y `ROUND` para redondear a dos decimales, asegurando precisión financiera.
3. **Cálculo de Importes en `orders`**: Utiliza una subconsulta para calcular `netamount` y `totalamount` solo si son nulos, evitando sobreescribir valores existentes.

Código:

```
CREATE OR REPLACE PROCEDURE setOrderAmount()
LANGUAGE plpgsql
AS $$
BEGIN
    -- Actualizar la columna price en orderdetail
    UPDATE public.orderdetail AS od
    SET price = ROUND(
            (
                SELECT p.price * POWER(1 + 0.02, EXTRACT(YEAR FROM
current_date) - EXTRACT(YEAR FROM o.orderdate))
                FROM public.products AS p
                JOIN public.orders AS o ON od.orderid = o.orderid
                WHERE od.prod_id = p.prod_id
            )::numeric, 2);

    -- Actualizar las columnas netamount y totalamount en las órdenes
afectadas solo si son nulos
    UPDATE public.orders AS o
    SET netamount = subquery.netamount,
        totalamount = ROUND(subquery.netamount * (1 + o.tax / 100), 2)
    FROM (
        SELECT od.orderid,
               SUM(od.price * od.quantity) AS netamount
        FROM public.orderdetail AS od
        JOIN public.orders AS o ON od.orderid = o.orderid
        GROUP BY od.orderid
    ) AS subquery
    WHERE o.orderid = subquery.orderid
      AND o.netamount IS NULL
      AND o.totalamount IS NULL;
END $$;
```

1.4 Función `getTopSales`

Explicación Resumida:

La función `getTopSales` devuelve las películas más vendidas en dos años específicos.

1. **Encapsulación de Lógica Compleja**: La función se define en PL/pgSQL para encapsular la lógica de la consulta.
2. **Cálculo y Filtrado**: Usa subconsultas y `ROW_NUMBER` para calcular las ventas anuales de cada película y filtrar solo la más vendida por año.
3. **Parámetros de Entrada y Salida**: Recibe dos años como parámetros y devuelve los resultados como registros (`SETOF RECORD`).

Código:

```
CREATE OR REPLACE FUNCTION getTopSales(year1 INT, year2 INT, OUT Year INT,
OUT Film VARCHAR, OUT sales BIGINT)
RETURNS SETOF RECORD
AS $$
BEGIN
    RETURN QUERY
    SELECT sales_year, movietitle, total_sales
    FROM (
        SELECT EXTRACT(YEAR FROM o.orderdate)::INT AS sales_year,
               m.movietitle,
               SUM(od.quantity) AS total_sales,
               ROW_NUMBER() OVER(PARTITION BY EXTRACT(YEAR FROM
o.orderdate)::INT ORDER BY SUM(od.quantity) DESC) AS rank
        FROM public.orders o
        JOIN public.orderdetail od ON o.orderid = od.orderid
        JOIN public.products p ON od.prod_id = p.prod_id
        JOIN public.imdb_movies m ON p.movieid = m.movieid
        WHERE EXTRACT(YEAR FROM o.orderdate) BETWEEN year1 AND year2
        GROUP BY EXTRACT(YEAR FROM o.orderdate), m.movietitle
        ORDER BY total_sales DESC
    ) AS sales_rank
    WHERE rank = 1;
END;
$$ LANGUAGE plpgsql;
```

1.5 Función `getTopActors`

La función `getTopActors` devuelve los actores con más actuaciones en un género específico, incluyendo detalles sobre su debut y director.

1. **Encapsulación de Lógica Compleja**: La función en PL/pgSQL encapsula toda la lógica necesaria para obtener actores, sus películas, y directores, facilitando la reutilización y el mantenimiento.
2. **Uso de Subconsultas y Joins**: Utiliza múltiples `JOIN` para combinar datos de actores, películas, géneros y directores.
3. **Subconsulta para Debut**: La subconsulta interna selecciona la película de debut de cada actor en el género especificado, ordenando por año.
4. **Filtro y Ordenación**: Filtra actores con más de 4 películas en el género y los ordena por el número de actuaciones.

Código:

```
CREATE OR REPLACE FUNCTION getTopActors(genre CHAR, OUT Actor CHAR, OUT

 Num INT, OUT Debut INT, OUT Film CHAR, OUT Director CHAR)
RETURNS SETOF RECORD AS $$
DECLARE
    actor_record RECORD;
BEGIN
    FOR actor_record IN
```

```
        SELECT ia.actorname,
               COUNT(*) AS num_movies,
               MIN(im.year) AS debut_year,
               m.movietitle AS movie_title,
               d.directorname AS director
        FROM public.imdb_actors ia
        JOIN public.imdb_actormovies am ON ia.actorid = am.actorid
        JOIN public.imdb_movies im ON am.movieid = im.movieid
        JOIN public.imdb_moviegenres mg ON im.movieid = mg.movieid
        JOIN public.products p ON p.movieid = im.movieid
        JOIN public.orderdetail od ON p.prod_id = od.prod_id
        JOIN public.orders o ON od.orderid = o.orderid
        JOIN public.imdb_directormovies dm ON dm.movieid = im.movieid
        JOIN public.imdb_directors d ON dm.directorid = d.directorid
        JOIN (
            SELECT DISTINCT ON (am.actorid) am.actorid, m.movietitle
            FROM public.imdb_actormovies am
            JOIN public.imdb_movies m ON am.movieid = m.movieid
            JOIN public.imdb_moviegenres mg ON m.movieid = mg.movieid
            WHERE mg.genre = getTopActors.genre
            ORDER BY am.actorid, m.year
        ) AS m ON ia.actorid = m.actorid
        WHERE mg.genre = getTopActors.genre
        GROUP BY ia.actorname, m.movietitle, d.directorname
        HAVING COUNT(*) > 4
        ORDER BY num_movies DESC
    LOOP
        Actor := actor_record.actorname;
        Num := actor_record.num_movies;
        Debut := actor_record.debut_year;
        Film := actor_record.movie_title;
        Director := actor_record.director;
        RETURN NEXT;
    END LOOP;

    RETURN;
END;
$$ LANGUAGE plpgsql;
```

1.6 Conversión de atributos multivaluados en relaciones

Decisiones de Diseño Más Relevantes y sus Justificaciones:

1. **Uso de Transacciones**:
    - Encapsular todas las operaciones dentro de una transacción asegura que todas las modificaciones se realicen de manera consistente y que, en caso de error, ninguna operación se aplique parcialmente.
2. **Normalización de Datos**:

- Crear tablas separadas para `countries`, `genres` y `languages` y referenciar estas tablas desde `movies` evita la duplicación de datos y facilita su gestión.

3. **Uso de Claves Foráneas**:
   - Las claves foráneas aseguran que las referencias entre tablas sean válidas. Esto previene inconsistencias y asegura que todos los registros en `movies` estén correctamente asociados a datos válidos en las tablas relacionadas.

4. **Migración de Datos**:
   - Migrar los datos existentes desde las columnas multivaluadas a las nuevas tablas normalizadas elimina redundancias y asegura que cada país, género y idioma esté representado una sola vez en la base de datos.

5. **Actualización de Referencias**:
   - Actualizar la tabla `movies` para incluir las referencias a las nuevas tablas asegura que las relaciones entre datos sean correctas y completas.

6. **Eliminación de Columnas Obsoletas**:
   - Eliminar las columnas multivaluadas originales evita redundancias y posibles inconsistencias futuras, simplificando la estructura de la tabla `movies`.

Código:

```
BEGIN;

-- Tabla countries
CREATE TABLE countries (
    country_id SERIAL PRIMARY KEY,
    country_name VARCHAR(100) NOT NULL
);

-- Tabla genres
CREATE TABLE genres (
    genre_id SERIAL PRIMARY KEY,
    genre_name VARCHAR(100) NOT NULL
);

-- Tabla languages
CREATE TABLE languages (
    language_id SERIAL PRIMARY KEY,
    language_name VARCHAR(100) NOT NULL
);

-- Tabla movies
ALTER TABLE public.movies
ADD COLUMN country_id INT,
ADD COLUMN genre_id INT,
ADD COLUMN language_id INT,
```

```
ADD CONSTRAINT fk_country FOREIGN KEY (country_id) REFERENCES
countries(country_id),
ADD CONSTRAINT fk_genre FOREIGN KEY (genre_id) REFERENCES genres(genre_id),
ADD CONSTRAINT fk_language FOREIGN KEY (language_id) REFERENCES
languages(language_id);

-- Migración de datos
INSERT INTO countries (country_name)
SELECT DISTINCT unnest(moviecountries) FROM movies;

INSERT INTO genres (genre_name)
SELECT DISTINCT unnest(moviegenres) FROM movies;

INSERT INTO languages (language_name)
SELECT DISTINCT unnest(movielanguages) FROM movies;

UPDATE movies m
SET country_id = c.country_id
FROM countries c
WHERE m.moviecountries @> ARRAY[c.country_name];

UPDATE movies m
SET genre_id = g.genre_id
FROM genres g
WHERE m.moviegenres @> ARRAY[g.genre_name];

UPDATE movies m
SET language_id = l.language_id
FROM languages l
WHERE m.movielanguages @> ARRAY[l.language_name];

ALTER TABLE movies
DROP COLUMN moviecountries,
DROP COLUMN moviegenres,
DROP COLUMN movielanguages;

COMMIT;
```

1.7 Trigger `updOrders`

1. **Encapsulación en Función**:
    - Encapsular la lógica de actualización en una función permite mantener el código organizado y facilita su reutilización.
2. **Actualización de Total del Pedido**:
    - La función recalcula el total del pedido cada vez que se añade, actualiza o elimina un artículo del carrito, asegurando que el total esté siempre actualizado.
3. **Disparador (Trigger)**:

- Crear un disparador asegura que la función se ejecute automáticamente después de cualquier inserción, actualización o eliminación en `orderdetail`.

4. **Uso de `AFTER INSERT OR UPDATE OR DELETE`**:
    - Configurar el disparador para que se ejecute después de estas operaciones asegura que cualquier cambio en los detalles del pedido se refleje inmediatamente en el total del pedido.

Código:

```
CREATE OR REPLACE FUNCTION update_orders()
RETURNS TRIGGER AS $$
BEGIN
    -- Actualizar el total del pedido cuando se añada, actualice o elimine
un artículo del carrito
    UPDATE public.orders
    SET total_amount = (
        SELECT COALESCE(SUM(quantity * unit_price), 0)
        FROM public.orderdetail
        WHERE orderid = NEW.orderid
    )
    WHERE orderid = NEW.orderid;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER updOrders
AFTER INSERT OR UPDATE OR DELETE ON public.orderdetail
FOR EACH ROW
EXECUTE FUNCTION update_orders();
```

1.8 Trigger `updRatings`

1. **Encapsulación en Función**:
    - Encapsular la lógica de actualización en una función permite mantener el código organizado y facilita su reutilización.

2. **Actualización de Conteo y Promedio de Valoraciones**:
    - La función recalcula el número total de valoraciones y el promedio de valoraciones cada vez que se añade, actualiza o elimina una valoración.

3. **Disparador (Trigger)**:
    - Crear un disparador asegura que la función se ejecute automáticamente después de cualquier inserción, actualización o eliminación en `ratings`.

4. **Uso de `AFTER INSERT OR UPDATE OR DELETE`**:

- Configurar el disparador para que se ejecute después de estas operaciones asegura que cualquier cambio en las valoraciones se refleje inmediatamente en el total y promedio de valoraciones de la película.

Código:

```
CREATE OR REPLACE FUNCTION update_movie_ratings()
RETURNS TRIGGER AS $$
DECLARE
    total_ratings numeric;
    new_rating_count integer;
    new_rating_mean numeric;
BEGIN
    -- Calcular el nuevo número total de valoraciones para la película
    SELECT COUNT(*) INTO total_ratings
    FROM public.ratings
    WHERE movieid = NEW.movieid;

    -- Actualizar el campo ratingcount en la tabla imdb_movies
    UPDATE public.imdb_movies
    SET ratingcount = total_ratings
    WHERE movieid = NEW.movieid;

    -- Calcular el nuevo ratingmean para la película
    SELECT COALESCE(AVG(rating), 0) INTO new_rating_mean
    FROM public.ratings
    WHERE movieid = NEW.movieid;

    -- Actualizar el campo ratingmean en la tabla imdb_movies
    UPDATE public.imdb_movies
    SET ratingmean = new_rating_mean
    WHERE movieid = NEW.movieid;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER updRatings
AFTER INSERT OR UPDATE OR DELETE ON public.ratings
FOR EACH ROW
EXECUTE FUNCTION update_movie_ratings();
```

1.9 Trigger `updInventoryAndCustomer`

1. **Encapsulación en Función**:
    - Encapsular la lógica de actualización en una función permite mantener el código organizado y facilita su reutilización.
2. **Condicional para Estado de Pago**:

- La función solo ejecuta las actualizaciones cuando el estado del pedido es 'Paid', asegurando que los cambios en el inventario y el balance del cliente solo ocurran cuando una compra ha sido efectivamente pagada.

3. **Actualización de Inventario**:
   - Al decrementar la cantidad en la tabla `inventory`, la función asegura que el inventario refleje correctamente las ventas realizadas.
4. **Actualización del Balance del Cliente**:
   - Al descontar el precio total de la compra del balance del cliente, la función asegura que el saldo del cliente sea preciso.
5. **Disparador (Trigger)**:
   - Crear un disparador asegura que la función se ejecute automáticamente después de cualquier actualización en `orders`.

Código:

```
CREATE OR REPLACE FUNCTION update_inventory_and_customer()
RETURNS

 TRIGGER AS $$
BEGIN
    IF NEW.status = 'Paid' THEN
        -- Actualizar la tabla inventory
        UPDATE public.inventory
        SET quantity = quantity - NEW.quantity
        WHERE product_id = NEW.product_id;

        -- Descuentar el precio total de la compra en la tabla customers
        UPDATE public.customers
        SET balance = balance - NEW.total_price
        WHERE customerid = NEW.customerid;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER updInventoryAndCustomer
AFTER UPDATE ON public.orders
FOR EACH ROW
EXECUTE FUNCTION update_inventory_and_customer();
```

2. Integración con Python

2.1 Mostrar tabla con SQLAlchemy

1. **Conexión a la Base de Datos**:
    - Usar `create_engine` para establecer la conexión a la base de datos PostgreSQL permite gestionar eficientemente las conexiones, aprovechando el pool de conexiones de SQLAlchemy.

```
engine =
create_engine('postgresql://alumnodb:alumnodb@localhost:5432/si1')
```

2. **Definición de la Consulta**:
    - Definir la consulta SQL como una cadena de texto con parámetros (`text("SELECT * FROM getTopSales(:year1, :year2)")`) permite parametrizar las consultas de manera segura.

```
query = text("SELECT * FROM getTopSales(:year1, :year2)")
```

3. **Ejecución de la Consulta**:
    - Usar el contexto `with engine.connect() as connection` asegura que la conexión se maneje correctamente, cerrándose automáticamente al finalizar la operación.

```
with engine.connect() as connection:
    result = connection.execute(query.bindparams(year1=2021,
year2=2022))
```

4. **Manejo de Resultados**:
    - Usar `result.fetchmany(10)` para obtener los primeros 10 resultados de la consulta permite manejar grandes volúmenes de datos de manera eficiente.

```
for row in result.fetchmany(10):
    print(row)
```

Código:

```
from sqlalchemy import create_engine, text

def execute_get_top_sales():
    # Establecer la conexión a la base de datos
    engine =
create_engine('postgresql://alumnodb:alumnodb@localhost:5432/si1')

    # Definir la consulta como una cadena de texto
    query = text("SELECT * FROM getTopSales(:year1, :year2)")
```

```
    # Ejecutar la consulta y obtener los resultados
    with engine.connect() as connection:
        result = connection.execute(query.bindparams(year1=2021,
year2=2022))

        for row in result.fetchmany(10):
            print(row)

if __name__ == "__main__":
    execute_get_top_sales()
```

3. Optimización

3.1 Estudio del impacto de un índice

3.1.1 Consulta `estadosDistintos.sql`

Resultados Sin Índices:

```
DROP INDEX
DROP INDEX
 num_estados
-------------
         185
(1 row)

QUERY PLAN
------------------------------------------------------------------------------
-------------------------------------------------
 Aggregate  (cost=5089.32..5089.33 rows=1 width=8)
   ->  Gather  (cost=1529.04..5089.31 rows=5 width=118)
         Workers Planned: 1
         ->  Hash Join  (cost=529.04..4088.81 rows=3 width=118)
               Hash Cond: (o.customerid = c.customerid)
               ->  Parallel Seq Scan on orders o  (cost=0.00..3558.37
rows=535 width=4)
                     Filter: (date_part('year'::text,
(orderdate)::timestamp without time zone) = '2017'::double precision)
               ->  Hash  (cost=528.16..528.16 rows=70 width=122)
                     ->  Seq Scan on customers c  (cost=0.00..528.16
rows=70 width=122)
                           Filter: ((country)::text = 'Peru'::text)
(10 rows)
```

- **Número de estados**: 185
- **Plan de ejecución sin índices**:
  - **Costo total**: 5089.32

- **Escaneo secuencial en `orders`**: La consulta realiza un escaneo secuencial completo en la tabla `orders` para filtrar los registros del año 2017, lo que es costoso en términos de rendimiento.
- **Escaneo secuencial en `customers`**: Similarmente, la consulta realiza un escaneo secuencial en la tabla `customers` para filtrar por país `Peru`.

Resultados Con Índices:

```
CREATE INDEX
CREATE INDEX
 num_estados
-------------
         185
(1 row)

QUERY PLAN
------------------------------------------------------------------------------
-----------------------------------------------------
 Aggregate  (cost=1684.17..1684.18 rows=1 width=8)
   ->  Hash Join  (cost=200.17..1684.16 rows=5 width=118)
         Hash Cond: (o.customerid = c.customerid)
         ->  Bitmap Heap Scan on orders o  (cost=19.46..1501.07 rows=909
width=4)
               Recheck Cond: (date_part('year'::text,
(orderdate)::timestamp without time zone) = '2017'::double precision)
               ->  Bitmap Index Scan on index_orderdate  (cost=0.00..19.24
rows=909 width=0)
                     Index Cond: (date_part('year'::text,
(orderdate)::timestamp without time zone) = '2017'::double precision)
         ->  Hash  (cost=179.83..179.83 rows=70 width=122)
               ->  Bitmap Heap Scan on customers c  (cost=4.83..179.83
rows=70 width=122)
                     Recheck Cond: ((country)::text = 'Peru'::text)
                     ->  Bitmap Index Scan on index_country
(cost=0.00..4.81 rows=70 width=0)
                           Index Cond: ((country)::text = 'Peru'::text)
(12 rows)
```

- **Número de estados**: 185
- **Plan de ejecución con índices**:
  - **Costo total**: 1684.17
  - **Bitmap Index Scan**: La consulta utiliza índices para realizar escaneos más eficientes.
    - **`index_orderdate`**: Permite un escaneo rápido basado en el año de `orderdate`, reduciendo el número de filas a revisar.
    - **`index_country`**: Filtra rápidamente los clientes del país `Peru`.

Mejora de Costes

- **Reducción de Costos**: El costo total de la consulta se reduce significativamente de 5089.32 a 1684.17. Esta reducción se debe a la utilización de índices que permiten acceder directamente a los registros relevantes sin necesidad de escanear completamente las tablas.
- **Eficiencia**:
  - **Escaneo Secuencial vs. Escaneo de Índices**: Los escaneos secuenciales son reemplazados por escaneos de índices, que son mucho más rápidos y eficientes.
  - **Uso de `Bitmap Index Scan`**: Mejora adicionalmente la eficiencia al permitir operaciones paralelas y consolidar resultados antes de acceder a las tablas.

La implementación de índices específicos para las columnas consultadas reduce drásticamente los costos de ejecución y mejora el rendimiento general de las consultas.

3.2 Estudio del impacto de cambiar la forma de realizar una consulta

Planes de Ejecución de las Consultas Alternativas

Consulta 1:

```
SELECT customerid
FROM customers
WHERE customerid NOT IN (
    SELECT customerid
    FROM orders
    WHERE status = 'Paid'
);
```

Plan de Ejecución:

- **Seq Scan on customers**: Escaneo secuencial en `customers`.
- **SubPlan**: Escaneo de índice en `orders` con condición `status = 'Paid'`.

Consulta 2:

```
SELECT customerid
FROM (
    SELECT customerid
    FROM customers
    UNION ALL
    SELECT customerid
    FROM orders
    WHERE status = 'Paid'
) AS A
GROUP BY customerid
HAVING COUNT(*) = 1;
```

Plan de Ejecución:

- **HashAggregate**: Agregación con condición de grupo `COUNT(*) = 1`.

- **Append**: Unión de escaneo secuencial en `customers` y escaneo de índice en `orders`.

Consulta 3:

```
SELECT customerid
FROM customers
EXCEPT
SELECT customerid
FROM orders
WHERE status = 'Paid';
```

Plan de Ejecución:

- **HashSetOp Except**: Operación de conjunto para `EXCEPT`.
- **Append**: Unión de escaneo secuencial en `customers` y escaneo de índice en `orders`.

Comparación de las Consultas

1. **Consulta que devuelve algún resultado nada más comenzar su ejecución**:
    - **Consulta 2**: La consulta que utiliza `UNION ALL` con `GROUP BY` y `HAVING COUNT(*) = 1` tiene la ventaja de poder devolver resultados parciales rápidamente debido a la naturaleza del escaneo y agregación.
2. **Consulta que se puede beneficiar de la ejecución en paralelo**:
    - **Consulta 3**: La consulta que utiliza `EXCEPT` puede beneficiarse de la ejecución en paralelo ya que ambas subconsultas (`customers` y `orders`) pueden ser escaneadas simultáneamente, y el operador de conjunto `EXCEPT` se presta bien a la paralelización.

Conclusión

- **Consulta 1**: Utiliza un escaneo secuencial combinado con un subplan de escaneo de índice, lo que puede ser menos eficiente.
- **Consulta 2**: Agrega y filtra resultados usando `UNION ALL`, que puede devolver resultados parciales rápidamente.
- **Consulta 3**: Utiliza una operación de conjunto (`EXCEPT`) que puede beneficiarse significativamente de la ejecución en paralelo.

3.3 Estudio del impacto de la generación de estadísticas

3.3.1 Planificación y análisis de consultas

Preguntas y Respuestas

1. **¿Qué hace el generador de estadísticas?**
    - `ANALYZE` recopila y almacena información sobre la distribución de datos en las tablas, ayudando al planificador de consultas a tomar decisiones más informadas y eficientes.

2. **Usar la sentencia ANALYZE, no VACUUM ANALYZE**:
    - `ANALYZE` actualiza solo las estadísticas de las tablas, mientras que `VACUUM ANALYZE` también recupera espacio y organiza las páginas de la tabla.
3. **EXPLAIN ANALYZE no calcula estadísticas (¡y ejecuta la consulta!)**:
    - `EXPLAIN ANALYZE` ejecuta la consulta y proporciona el plan de ejecución y tiempos reales, pero no actualiza ni calcula estadísticas.
4. **¿Por qué la planificación de las dos consultas es la misma hasta que se generan las estadísticas?**
    - Sin estadísticas, el planificador de consultas utiliza un plan genérico basado en heurísticas. Con estadísticas, puede optimizar los planes de ejecución basándose en los datos actuales.

Análisis de Resultados y Mejoras de Costes

Sin Índices:

- **Costo elevado**: Ambas consultas realizan un escaneo secuencial completo en la tabla `orders`, resultando en un alto coste de ejecución.

Con Índices:

- **Mejora parcial**: La consulta para `status IS NULL` mejora significativamente usando `Index Only Scan`, reduciendo el coste. La consulta para `status = 'Shipped'` aún realiza un escaneo secuencial paralelo.

Con `ANALYZE`:

- **Planificación óptima**: La ejecución de `ANALYZE` mejora la consulta para `status IS NULL` y reduce los costes para otros estados (`Paid`, `Processed`) utilizando `Bitmap Index Scan` y `Bitmap Heap Scan`.

Script `countStatus.sql`:

```
DROP INDEX IF EXISTS INDEX_STATUS;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS IS NULL;

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';

CREATE INDEX INDEX_STATUS ON public.orders(STATUS);

EXPLAIN
```

```sql
SELECT COUNT(*)
FROM public.orders
WHERE STATUS IS NULL;

EXPLAIN
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';

ANALYZE VERBOSE public.orders;

EXPLAIN
SELECT COUNT(*)
FROM public.orders
WHERE STATUS IS NULL;

EXPLAIN
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Shipped';

EXPLAIN
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Paid';

EXPLAIN
SELECT COUNT(*)
FROM public.orders
WHERE STATUS = 'Processed';
```

Conclusión

- **Impacto del Índice y Estadísticas**: Los índices y la generación de estadísticas mejoran significativamente los costes de ejecución de las consultas. Los índices optimizan el acceso a los datos y `ANALYZE` proporciona la información necesaria para elegir los mejores planes de ejecución.