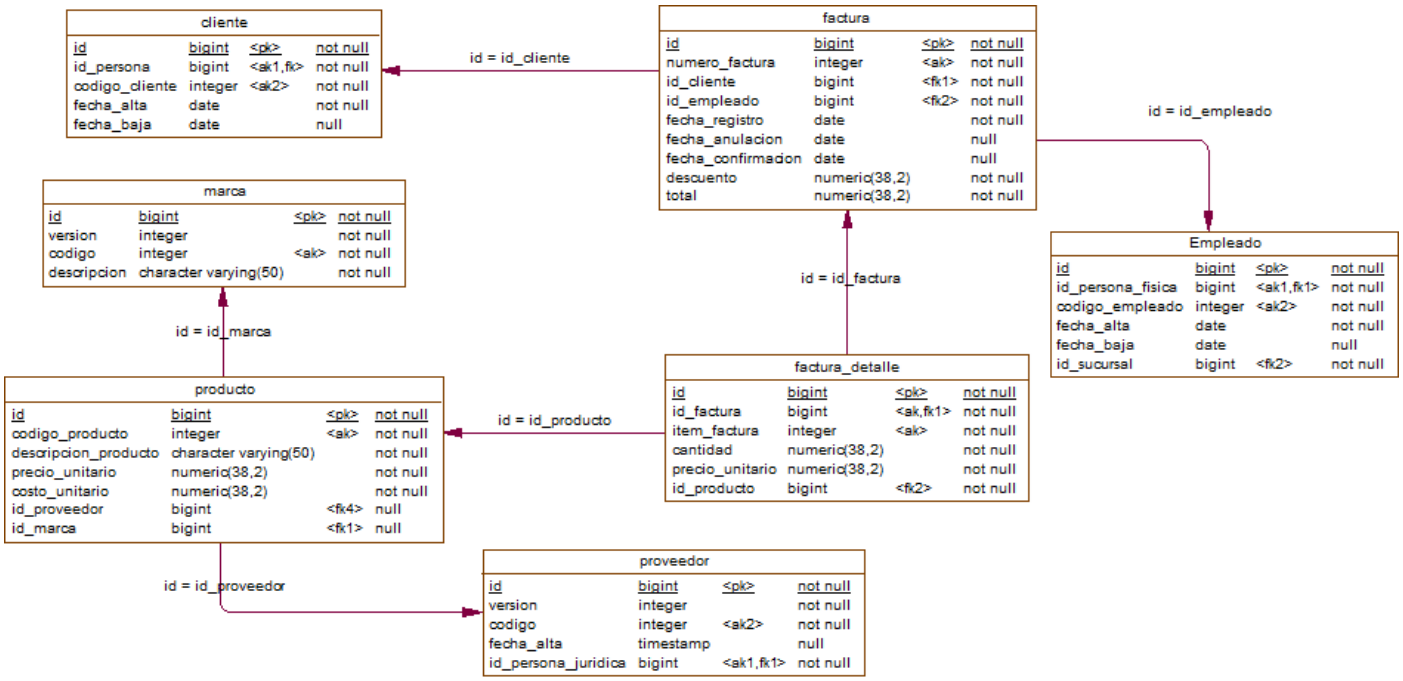
BASES DE DATOS

RECUPERATORIO PARCIAL 2

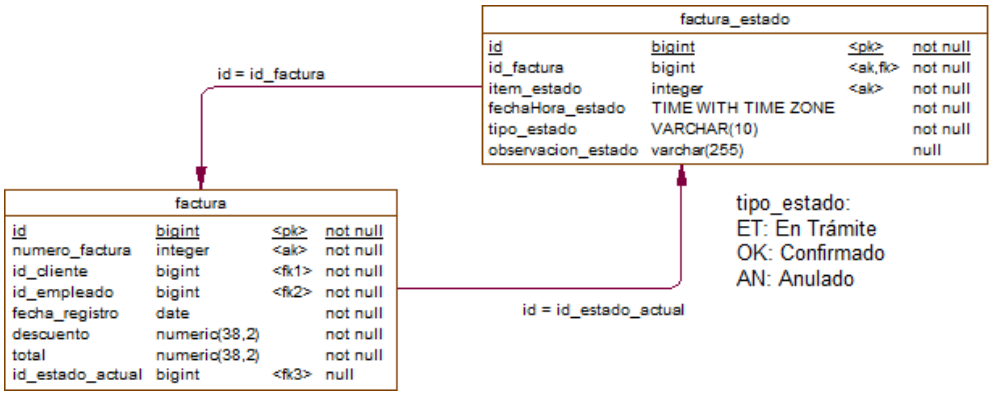
27/11/2024

Seguimos trabajando con el modelo físico del sistema de “Gestión” que, con sus variantes, venimos utilizando en este cuatrimestre.

Modelo físico de Ventas



Agregado de transición de estados a factura



Ejercicio 1: Actualización de estructura y pasaje de datos

Escribir un script SQL que haga lo siguiente:

- Modifique la estructura de la tabla de factura para utilizar una tabla factura_estado, que contemple las transiciones de estados de acuerdo al modelo de la figura 2. El estado "ET" es inicial; los estados "Confirmado" y "Anulado" son finales.
- Trasladar los datos de las facturas confirmadas o anuladas, a la nueva estructura.
- La columna *id_estado_actual* representa el estado actual de la factura. Asignarle valor a partir del último estado registrado
- Eliminar información redundante.

Recomendación: realizar los pasos en el orden correcto para no perder información!.

Solución Sugerida

```
-- Paso 1: Crear la tabla factura_estado
CREATE TABLE venta.factura_estado (
    id BIGINT PRIMARY KEY,  -- Se usará BIGINT en lugar de BIGSERIAL
    id_factura BIGINT NOT NULL,
    item_estado INTEGER NOT NULL,
    fechaHora_estado TIMESTAMP NOT NULL,
    tipo_estado VARCHAR(10) NOT NULL,
    CONSTRAINT factura_estado_factura_fk FOREIGN KEY (id_factura) REFERENCES venta.factura (id),
    constraint factura_estado_uk unique (id_factura,item_estado)
);

-- Crear una secuencia para asignar valores al ID de factura_estado
CREATE SEQUENCE venta.factura_estado_seq;

-- Paso 2: Agregar la columna id_estado_actual en la tabla factura
ALTER TABLE venta.factura
ADD COLUMN id_estado_actual BIGINT,
ADD CONSTRAINT factura_estado_actual_fk FOREIGN KEY (id_estado_actual)
REFERENCES venta.factura_estado (id);
```

```
-- Paso 3: Insertar datos en factura_estado
-- Insertar el estado inicial "ET" (item_estado = 1)
INSERT INTO venta.factura_estado
(id, id_factura, item_estado, fechaHora_estado, tipo_estado)
SELECT NEXTVAL('venta.factura_estado_seq'), id, 1, fecha, 'ET'
FROM venta.factura;

-- Insertar el estado "OK" para facturas confirmadas (item_estado = 2)
INSERT INTO venta.factura_estado
(id, id_factura, item_estado, fechaHora_estado, tipo_estado)
SELECT NEXTVAL('venta.factura_estado_seq'), id, 2, fecha_confirmacion, 'OK'
FROM venta.factura
WHERE fecha_confirmacion IS NOT NULL;

-- Insertar el estado "AN" para facturas anuladas (item_estado = 2)
INSERT INTO venta.factura_estado
(id, id_factura, item_estado, fechaHora_estado, tipo_estado)
SELECT NEXTVAL('venta.factura_estado_seq'), id, 2, fecha_anulacion, 'AN'
FROM venta.factura
WHERE fecha_anulacion IS NOT NULL;

-- Paso 4: Asignar id_estado_actual en la tabla factura según el estado final
-- Para las fact OK o anuladas
UPDATE venta.factura f
SET id_estado_actual = fe.id
FROM venta.factura_estado fe
WHERE fe.id_factura = f.id
AND fe.item_estado = 2;

-- Para las fact ET
UPDATE venta.factura f
```

```
SET id_estado_actual = fe.id
FROM venta.factura_estado fe
WHERE fe.id_factura = f.id
AND fe.item_estado = 1
AND NOT EXISTS (SELECT 1
 FROM venta.factura_estado fe2
 WHERE fe2.id_factura = f.id
   AND fe2.item_estado = 2
);

-- Paso 5: Eliminar columnas redundantes de la tabla factura
ALTER TABLE venta.factura
DROP COLUMN fecha_anulacion,
DROP COLUMN fecha_confirmacion;
```

Ejercicio 2: Función PostgreSQL

Escribir una función en PostgreSQL llamada f_calcular_promedio_facturacion() que devuelva el importe promedio de facturación, recibiendo como parámetros el año y el mes de facturación. La función también recibirá un parámetro de tipo VARCHAR(50) denominado p_marca, que contendrá '*' si se deben considerar todas las marcas, o el nombre específico de una marca de interés cuyo promedio de facturación se desee calcular.

Si el parámetro p_marca contiene el nombre de una marca específica (es decir, distinto de '*'), la función debe verificar que esta marca exista en la columna descripcion de la tabla producto.marca. Si la marca no existe, la función debe retornar un mensaje de error.

Incluir un ejemplo de sentencia de invocación de la función. Considerar únicamente facturas confirmadas: utilizar la tabla factura_estado.

```
CREATE OR REPLACE FUNCTION f_calcular_promedio_facturacion(
    p_anio INTEGER,
    p_mes INTEGER,
    p_marca VARCHAR(50)
) RETURNS NUMERIC(38, 2) AS $$

DECLARE
    v_promedio_facturacion NUMERIC(38, 2);
    v_marca_id BIGINT;

BEGIN
    -- Validar si se especificó una marca específica
    IF p_marca <> '*' THEN
        -- Verificar si la marca existe en la tabla producto.marca
        SELECT id INTO v_marca_id
        FROM producto.marca
        WHERE descripcion = p_marca;

        -- Si no existe, lanzar un error
        IF NOT FOUND THEN
            RAISE EXCEPTION 'La marca especificada no existe.';
        END IF;

        -- Calcular el promedio de facturación SOLO para productos de la marca específica
        SELECT AVG(fd.cantidad * fd.precio_unitario) INTO v_promedio_facturacion
        FROM venta.factura f
        JOIN venta.factura_detalle fd ON f.id = fd.id_factura
        JOIN producto.producto p ON fd.id_producto = p.id
        JOIN venta.factura_estado fe ON f.id_estado_actual = fe.id
        WHERE EXTRACT(YEAR FROM f.fecha_registro) = p_anio
          AND EXTRACT(MONTH FROM f.fecha_registro) = p_mes
          AND p.id_marca = v_marca_id
          AND fe.tipo_estado = 'OK'; -- Solo facturas confirmadas
    ELSE
        -- Calcular el promedio considerando todas las marcas
        SELECT AVG(total) INTO v_promedio_facturacion
        FROM venta.factura f
        JOIN venta.factura_estado fe ON f.id_estado_actual = fe.id
        WHERE EXTRACT(YEAR FROM f.fecha_registro) = p_anio
          AND EXTRACT(MONTH FROM f.fecha_registro) = p_mes
          AND fe.tipo_estado = 'OK'; -- Solo facturas confirmadas
    END IF;

    -- Retornar el promedio calculado (o 0 si no hay datos)
    RETURN COALESCE(v_promedio_facturacion, 0);
END;
$$ LANGUAGE plpgsql;

SELECT f_calcular_promedio_facturacion(2023, 2, 'ASÚS');

SELECT f_calcular_promedio_facturacion(2023, 2, '*');
```

Ejercicio 3: Facturación mensual a clientes

Escribir una consulta SQL que muestre los datos de los clientes que hayan tenido facturación mensual superior al 50% del promedio mensual en algún mes del año 2023. La consulta debe incluir:

- código del cliente
- apellido y nombre si el cliente es una persona física, o denominacion si es una persona jurídica
- mes en el que obtuvo la facturación superior al 50% del promedio mensual
- total facturado en ese mes

Los clientes con mayor facturación mensual deben aparecer primero.

```
SELECT
    c.codigo AS codigo_cliente,
    CASE
        WHEN pf.id IS NOT NULL THEN pf.apellido || ', ' || pf.nombre
        WHEN pj.id IS NOT NULL THEN pj.denominacion
        ELSE 'Desconocido'
    END AS nombre_cliente,
    EXTRACT(MONTH FROM f.fecha_registro)::INTEGER AS mes,
    SUM(f.total) AS total_facturado,
    f_calcular_promedio_facturacion(2023,  EXTRACT(MONTH  FROM  f.fecha_registro)::INTEGER,  '*')  AS
promedio_mensual,
    f_calcular_promedio_facturacion(2023, EXTRACT(MONTH FROM f.fecha_registro)::INTEGER, '*') * 0.5 AS
umbral_50
FROM
    venta.factura f
JOIN
    persona.cliente c ON f.id_cliente = c.id
LEFT JOIN
    persona.persona_fisica pf ON c.id_persona = pf.id_persona
LEFT JOIN
    persona.persona_juridica pj ON c.id_persona = pj.id_persona
JOIN
    venta.factura_estado fe ON f.id_estado_actual = fe.id
WHERE
    fe.tipo_estado = 'OK'
    AND EXTRACT(YEAR FROM f.fecha_registro) = 2023
GROUP BY
    c.codigo,
    pf.id, pf.apellido, pf.nombre,
    pj.id, pj.denominacion,
    EXTRACT(MONTH FROM f.fecha_registro)
HAVING
    SUM(f.total) > f_calcular_promedio_facturacion(2023, EXTRACT(MONTH FROM f.fecha_registro)::INTEGER,
'*') * 0.5
ORDER BY
    total_facturado DESC;
```

Los valores promedio_mensual y umbral_50, podrían no estar ya que no figuran el enunciado, pero se exponen para en caso de probar las consultas contrastar los resultados.

Variante:

```
-------------------------------------------------------------
-- variante para ejercicio 3 sin uso de función de ej 2
-------------------------------------------------------------
CREATE TEMP TABLE promedios_mensuales AS
SELECT
    EXTRACT(YEAR FROM f.fecha_registro) AS anio,
    EXTRACT(MONTH FROM f.fecha_registro) AS mes,
    AVG(f.total) AS promedio_mensual
FROM
    venta.factura f
JOIN
    venta.factura_estado fe ON f.id_estado_actual = fe.id
WHERE
```

```
    fe.tipo_estado = 'OK'
    AND EXTRACT(YEAR FROM f.fecha_registro) = 2023
GROUP BY
    EXTRACT(YEAR FROM f.fecha_registro),
    EXTRACT(MONTH FROM f.fecha_registro);
SELECT
    c.codigo AS codigo_cliente,
    CASE
        WHEN pf.id IS NOT NULL THEN pf.apellido || ', ' || pf.nombre
        WHEN pj.id IS NOT NULL THEN pj.denominacion
        ELSE 'Desconocido'
    END AS nombre_cliente,
    EXTRACT(MONTH FROM f.fecha_registro)::INTEGER AS mes,
    SUM(f.total) AS total_facturado,
    pm.promedio_mensual,
    pm.promedio_mensual * 0.5 AS umbral_50
FROM
    venta.factura f
JOIN
    persona.cliente c ON f.id_cliente = c.id
LEFT JOIN
    persona.persona_fisica pf ON c.id_persona = pf.id_persona
LEFT JOIN
    persona.persona_juridica pj ON c.id_persona = pj.id_persona
JOIN
    venta.factura_estado fe ON f.id_estado_actual = fe.id
JOIN
    promedios_mensuales pm
    ON EXTRACT(YEAR FROM f.fecha_registro) = pm.anio
    AND EXTRACT(MONTH FROM f.fecha_registro) = pm.mes
WHERE
    fe.tipo_estado = 'OK'
    AND EXTRACT(YEAR FROM f.fecha_registro) = 2023
GROUP BY
    c.codigo,
    pf.id, pf.apellido, pf.nombre,
    pj.id, pj.denominacion,
    EXTRACT(MONTH FROM f.fecha_registro),
    pm.promedio_mensual
HAVING
    SUM(f.total) > pm.promedio_mensual * 0.5
ORDER BY
    total_facturado DESC;
```

Ejercicio 4: Actualización de precios

Se requiere implementar un mecanismo que registre los cambios en los precios de los productos dentro del sistema. Para ello, se debe crear una tabla de historial de precios (producto.historial_precios) destinada a almacenar información sobre las actualizaciones realizadas en el campo precio_unitario de la tabla producto.producto.

La tabla de historial debe incluir las siguientes columnas:

- Identificador del producto.
- Fecha y hora en que se realizó el cambio.
- Valor del precio anterior antes de la actualización.

Además, se debe desarrollar un trigger que, al detectar una actualización en el campo precio_unitario, registre automáticamente un nuevo registro en la tabla de historial con los datos requeridos.

```
CREATE TABLE producto.historial_precios (
    id BIGINT,
    id_producto BIGINT NOT NULL,
    fecha_hora TIMESTAMP NOT NULL,
    precio_anterior NUMERIC(38, 2) NOT NULL,
    CONSTRAINT historial_precios_producto_fk FOREIGN KEY (id_producto) REFERENCES producto.producto
(id),
   CONSTRAINT pk_historial_precios PRIMARY KEY (id)
);
CREATE SEQUENCE producto.historial_precios_seq;
CREATE OR REPLACE FUNCTION registrar_cambio_precio()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO producto.historial_precios (id,id_producto, fecha_hora, precio_anterior)
    VALUES (
        nextval('producto.historial_precios_seq'),
        NEW.id,                          -- Identificador del producto
        CURRENT_TIMESTAMP,               -- Fecha y hora actual
        OLD.precio_unitario              -- Precio anterior
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_registrar_cambio_precio
AFTER UPDATE ON producto.producto
FOR EACH ROW
WHEN (OLD.precio_unitario IS DISTINCT FROM NEW.precio_unitario)
-- Solo si el precio cambió (1)
    EXECUTE FUNCTION registrar_cambio_precio();
```

Ejercicio 5: Concurrencia

Para nuestro sistema de gestión suponga que se debe realizar gestión de inventarios (control de stock). Se creó la siguiente tabla inventario.inventario que registra el stock de cada producto:

```
CREATE TABLE inventario.inventario (
    id bigint NOT NULL,
    id_producto bigint NOT NULL,
    cantidad integer NOT NULL,
    fecha_actualizacion timestamp NOT NULL DEFAULT NOW(),
    CONSTRAINT inventario_pk PRIMARY KEY (id),
    CONSTRAINT inventario_producto_fk FOREIGN KEY (id_producto) REFERENCES producto.producto (id)
);
```

a) Proporcionar un ejemplo práctico de cómo usar el nivel de aislamiento Serializable para evitar conflictos concurrentes durante la confirmación de una venta.

b) Escribir un ejemplo que implemente bloqueo pesimista para garantizar la consistencia al actualizar el inventario.

c) ¿Qué bloqueos se generan en cada caso?

d) ¿Cuándo se liberan los bloqueos? ¿Es en ambos casos igual?

a) Isolation Level

```
-- Establecer el nivel de aislamiento predeterminado para la sesión
SET default_transaction_isolation = 'serializable';

-- Iniciar la transacción
BEGIN;
/*
Alternativa: BEGIN TRANSACTION ISOLATION LEVEL
En este caso solo se setea el isolation level de esa transacción. Ambos son válidos para nuestro ejercicio.
*/

-- Verificar el stock disponible para el producto
SELECT cantidad
FROM inventario.inventario
WHERE id_producto = 1;

-- Si hay suficiente stock, actualizar el inventario
UPDATE inventario.inventario
SET cantidad = cantidad - 5,
    fecha_actualizacion = NOW()
WHERE id_producto = 1;

-- Confirmar la transacción
COMMIT;
```

b) Bloqueo pesimista

```
BEGIN;

-- Bloquear la fila del producto para garantizar que ninguna otra transacción la modifique
SELECT cantidad
FROM inventario.inventario
WHERE id_producto = 1
FOR UPDATE;

-- Verificar si hay suficiente stock
DO $$
BEGIN
    IF (SELECT cantidad FROM inventario.inventario WHERE id_producto = 1) < 5 THEN
        RAISE EXCEPTION 'Stock insuficiente para el producto 1';
    END IF;
END;
$$;
```

```
-- Actualizar el stock
UPDATE inventario.inventario
SET cantidad = cantidad - 5,
    fecha_actualizacion = NOW()
WHERE id_producto = 1;

-- Confirmar la transacción
COMMIT;
```

c) Bloqueos generados

Nivel de aislamiento Serializable: Se generan bloqueos a nivel de fila. Se pueden generar bloqueos adicionales para prevenir anomalías relacionadas con lecturas repetidas.

Bloqueo pesimista (FOR UPDATE): Se genera un bloqueo de fila (ROW LOCK) en la fila correspondiente al producto en el inventario. Este bloqueo evita tanto las lecturas no confirmadas como las actualizaciones concurrentes

d) Liberación de bloqueos

En ambos casos, los bloqueos se liberan al finalizar la transacción.

Ejercicio 6: SQL Dinámico.

Este es un fragmento de un ejemplo de un apunte de SQL Dinámico:

```
DECLARE
    @fechaInicio DATE = '2022-01-01',@fechaFin DATE = '2022-12-31',@fecha DATE,@sql VARCHAR(300);

SET @fecha = @fechaInicio;

WHILE (@fecha <= @fechaFin)
   BEGIN
       SET @sql = 'ALTER TABLE #ventas_por_producto ADD ['
                  + CAST(YEAR(@fecha) AS VARCHAR(4)) + '-'
                  + RIGHT('0' + CAST(MONTH(@fecha) AS VARCHAR(2)), 2)
                  + '] NUMERIC(38, 2) NULL;';

       EXEC(@sql);

       SET @fecha = DATEADD(MONTH, 1, @fecha);
   END;
```

Describir:

a) ¿Qué hace este fragmento de código? ¿Cuál es el nombre que asigna a las columnas?

b) ¿Cuántas columnas se generarán si @fechaInicio y @fechaFin van desde '2000-06-01' a '2024-11-20'? ¿Cuál es el nombre de la primera y de la última columna generadas?

a) ¿Qué hace?

Este código utiliza SQL dinámico para agregar nuevas columnas a una tabla temporal llamada #ventas_por_producto.

Las columnas se crean para cada mes dentro del rango de fechas definido por @fechaInicio y @fechaFin. Nombre de las columnas asignadas:

Las columnas tienen un nombre en el formato: [YYYY-MM], donde:YYYY es el año. MM es el mes, con dos dígitos (por ejemplo, "01" para enero).

Ejemplo:

Para el mes de enero de 2022, el nombre de la columna será [2022-01].

Para diciembre de 2022, será [2022-12].

b) Columnas generadas

@fechaInicio corresponde a junio de 2000.

@fechaFin corresponde a noviembre de 2024.

Desde junio de 2000 a noviembre de 2024, son 24 años y 6 meses.

Total de meses = 24×12+6=294 meses

Se generarán 294 columnas.

La primera columna : [2000-06].

Última columna generada: [2024-11].

Ejercicio 7: Administración de bases de datos

Mencione al menos cuatro tipos de usuarios de bases de datos. Describa sus características; indique diferencias.

Según el apunte de la Unidad 6: DBA.

- **NAIF**
  Los usuarios NAIF son usuarios finales que no saben mucho sobre tecnología, ni SQL, y usan el sistema llamando a uno de las aplicaciones que ya se han desarrollado. Por ejemplo, un cajero de banco seleccionará la opción del menú “Transferir” de la aplicación bancaria para mover $15.000 de la cuenta A a la cuenta B. Este proceso le pide al cajero la cantidad de dinero que se debe mover, la cuenta de la que proviene el dinero y la cuenta a la que se destinará el dinero.
- **OCASIONAL**
  Los usuarios con conocimientos de lenguaje SQL que por alguna razón se les permite conectarse al servidor de BBDD para explorarlo y/o consultar, durante breves períodos, y con permisos de acceso limitados, en BBDD no críticas.
- **ESPECIALIZADOS**
  Son usuarios avanzados que escriben programas de bases de datos que no se adaptan a la forma tradicional de procesar datos. Entre estas aplicaciones se encuentran los sistemas de diseño asistido por computadora, las bases de conocimiento y los sistemas expertos, los sistemas que almacenan datos con tipos de datos complejos (como datos audiovisuales) y los sistemas que modelan el entorno.
- **SOFISTICADOS**
  Saben cómo usar el sistema de BBDD sin escribir programas. Este grupo está formado por analistas que utilizan consultas para ver los datos de la BBDD. Por ejemplo, las herramientas de procesamiento analítico en línea (OLAP) facilitan el trabajo de los analistas al permitirles ver los resúmenes de datos de diferentes maneras. Por ej., los analistas pueden ver el total de productos por categorías, por número de ventas o por una combinación de categorías y número de ventas.
- **DESARROLLADOR**
  Profesionales informáticos que escriben aplicaciones para interactuar con las BBDD. Los programadores de aplicaciones pueden elegir entre muchas herramientas para desarrollar interfaces de usuario.
  Tienen distintos niveles de permisos dependiendo del entorno de desarrollo al que estén accediendo.
- **DE APLICACIÓN**
  Usuarios que sólo se utilizan para que una aplicación se conecte a las BBDD. Sólo tiene acceso desde el equipo en que se ejecute dicha aplicación, y tiene permisos determinados por su funcionalidad. Se los utiliza configurando su acceso en la cadena de conexión de la app.
- **DISEÑADOR**
  Usuario que colabora durante el proceso de análisis y diseño de la aplicación, generando desde los modelos conceptuales hasta el físico para dicha aplicación.
  Durante la etapa de desarrollo interactúa con la BBDD para crear sus esquemas y objetos hasta conseguir el definitivo. Luego lo entrega al DBA para que se implemente en los entornos de TEST y PRODUCCIÓN.
- **DBA**
  El Administrador de Base de Datos es una persona responsable de administrar y mantener el sistema de base de datos en una organización, garantizando que la base de datos sea accesible, segura y optimizada para el rendimiento. Desempeñan un papel crucial en el control y la supervisión de todos los aspectos de una base de datos, incluida su estructura, almacenamiento de datos, controles de acceso, copias de seguridad, recuperación y ajuste del rendimiento.
- **SYSADMIN**
  El Administrador de Sistemas interactúa con el DBA para brindar el apoyo necesario en lo que respecta a los requerimientos del Sistema Operativo, el hardware y las estructuras de almacenamiento y memoria necesarias para que el DBMS funcione de manera óptima.