Tema 6. DDL y DML

1. Introducción al lenguaje DDL

El DDL es la parte del lenguaje SQL que realiza la función de definición de datos del SGBD. Fundamentalmente, se encarga de la creación, modificación y eliminación de los objetos de la base de datos (es decir de los **metadatos**). Por supuesto es el encargado de la creación de las tablas.

Los elementos, llamados objetos, de la base de datos: tablas, vistas, columnas, índices,... se almacenan en el diccionario de datos. Por otro lado, muchos Sistemas Gestores de Bases de Datos aportan elementos para organizar estos objetos (como catálogos y esquemas).

Los objetos son manipulados y creados por los usuarios. En principio solo los administradores y los usuarios propietarios pueden acceder a cada objeto, salvo que se modifiquen los privilegios del objeto para permitir el acceso a otros usuarios.

Hay que tener en cuenta que ninguna instrucción DDL puede ser anulada por una instrucción `ROLLBACK` (la instrucción `ROLLBACK` está relacionada con el uso de transacciones, que se comentarán más adelante) por lo que hay que tener mucha precaución a la hora de utilizarlas. Es decir, las instrucciones DDL generan acciones que no se pueden deshacer. Salvo que dispongamos de alguna copia de seguridad o de otros elementos de recuperación.

2. Creación de bases de datos

Esta es una tarea administrativa que se comenta de forma simple. **Se trata de una tarea más propia de los administradores de bases de datos**, entre cuyo perfil no nos encontramos.

Crear la base de datos implica indicar los archivos y ubicaciones que se utilizarán para la misma, además de otras indicaciones técnicas y administrativas que no se comentarán en este tema.

Lógicamente solo es posible crear una base de datos si se tienen privilegios de DBA (DataBase Administrator).

El comando SQL de creación de una base de datos es `CREATE DATABASE`. Este comando crea una base de datos con el nombre que se indique.

Ejemplo:

```
CREATE DATABASE prueba;
```

En muchos sistemas, como Postgresql, eso basta para crear la bases de datos. El rol actual se convierte automáticamente en el propietario de la nueva base de datos. Es privilegio del propietario de una base de datos eliminarla más tarde (lo que también elimina todos los objetos que contiene, incluso si tienen un propietario diferente).

> Dado que necesita estar conectado al servidor de la base de datos para ejecutar el comando `CREATE DATABASE`, la pregunta sigue siendo cómo se puede crear la primera base de datos. Esta

> siempre la crea el comando `initdb` cuando se inicializa el área de almacenamiento de datos. Esta base de datos se llama `postgres`. Entonces, para crear la primera base de datos "ordinaria" podemos conectarnos a la recién creada `postgres`. Una segunda base de datos, `template1`, también se crea durante la inicialización del clúster de la base de datos. Siempre que se crea una nueva base de datos dentro del clúster, `template1` básicamente se clona. Esto significa que cualquier cambio que realice `template1` se propagará a todas las bases de datos creadas posteriormente. Debido a esto, evita crear objetos a template1 menos que desees que se propaguen a cada base de datos recién creada. También hay un programa que se puede ejecutar desde el shell para crear nuevas bases de datos, `createdb`.

A veces, podemos querer crear una base de datos para otra persona y hacer que esta se convierta en el propietario de la nueva base de datos, para que puedan configurarla y administrarla ellos mismos. Para lograrlo, debemos usar uno de los siguientes comandos:

```
CREATE DATABASE dbname OWNER rolename;
```

conectado a la base de datos; o bien

```
createdb -O rolename dbname
```

desde el shell de nuestro sistema operativo.

2.1 Objetos de la base de datos

Según los estándares actuales, una base de datos es un conjunto de objetos pensados para gestionar datos. Estos objetos están contenidos en esquemas, los esquemas suelen estar asociados al perfil de un usuario en particular.

En SQL estándar, existe el concepto de catálogo, que sirve para almacenar esquemas, y estos sirven para almacenar objetos. Así el nombre completo de un objeto vendría dado por:

```
catálogo.esquema.objeto
```

Es decir, los objetos pertenecen a esquemas y estos a catálogos.

En casi todos los sistemas de bases de datos hay un catálogo por defecto, de modo que si no se indica catálogo alguno al crear objetos, estos se almacenan allí. Del mismo modo, hay esquemas por defecto.

En Postgresql, podemos decir que el equivalente a los catálogos definidos en el estándar son las bases de datos. Además, también existen esquemas. Un catálogo (base de datos) puede tener varios esquemas; normalmente, cuando se crea una base de datos, se genera dentro un esquema llamado `public`.

Podemos decir entonces que se mantiene esta estructura dentro de Postgresql: **Cluster > Catálogo (Base de Datos) > Esquema > Tabla > Columnas y Filas**

> Si quieres saber más sobre la gestión (creación, destrucción, parámetros de configuración, *tablespaces*, ...) de bases de datos en Postgresql, puedes leer el capítulo 22 de su documentación: https://www.postgresql.org/docs/current/managing-databases.html.

3. Manejo de tablas

Como sabemos, una tabla de una base de datos relacional consta de filas y columnas. Normalmente, el número de columnas no varía mucho con el tiempo (aunque puede hacerlo), y el número de filas sí que suele cambiar con bastante frecuencia.

Cada columna tiene un tipo de datos. El tipo de datos restringe el conjunto de valores posibles que se pueden asignar a una columna y asigna semántica a los datos almacenados en la columna para que puedan usarse para cálculos. Por ejemplo, una columna declarada de tipo numérico no aceptará cadenas de texto arbitrarias, y los datos almacenados en dicha columna se pueden utilizar para cálculos matemáticos. Por el contrario, una columna declarada de un tipo de cadena de caracteres aceptará casi cualquier tipo de datos, pero no se presta a cálculos matemáticos, aunque se encuentran disponibles otras operaciones como la concatenación de cadenas.

PostgreSQL incluye un conjunto considerable de tipos de datos integrados que se adaptan a muchas aplicaciones. Los usuarios también pueden definir sus propios tipos de datos. La mayoría de los tipos de datos incorporados tienen nombres y semántica obvios, por lo que los veremos un poco más adelante . Algunos de los tipos de datos utilizados con frecuencia son `integer` para números enteros, `numeric` para números fraccionarios, `text` para cadenas de caracteres, `date` para fechas, `time` valores de hora del día y `timestamp` para valores que contienen fecha y hora.

3.1 Creación de tablas

Para crear una tabla, usaremos el comando `CREATE TABLE` con el nombre apropiado. En este comando, se especifica al menos un nombre para la nueva tabla, los nombres de las columnas y el tipo de datos de cada columna. Por ejemplo:

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

Esto crea una tabla nombrada `my_first_table` con dos columnas. La primera columna se llama `first_column` y tiene un tipo de datos de `text`; la segunda columna tiene el nombre `second_column` y el tipo `integer`. Los nombres de tabla y columna siguen la sintaxis de identificador de Postgresql. Ten en cuenta que la lista de columnas está separada por comas y entre paréntesis.

> Los identificadores SQL y las palabras clave deben comenzar con una letra (a- z, pero también letras con signos diacríticos y letras no latinas) o un guión bajo (_). Los caracteres posteriores en un identificador o palabra clave pueden ser letras, guiones bajos, dígitos (0- 9) o signos de dólar ($). Ten en cuenta que los signos de dólar no están permitidos en los identificadores de acuerdo con las reglas del estándar SQL, por lo que su uso puede hacer que las aplicaciones sean menos portables.

> El estándar SQL no definirá una palabra clave que contenga dígitos o comience o termine con un guión bajo, por lo que los identificadores de esta forma son seguros contra posibles conflictos con futuras extensiones del estándar.

Un ejemplo más real podría ser:

```
CREATE TABLE productos (
    num_producto    integer,
    nombre          text,
    precio          numeric
);
```

> Existe un límite en la cantidad de columnas que puede contener una tabla. Dependiendo de los tipos de columna, se encuentra entre 250 y 1600. Sin embargo, definir una tabla con casi tantas columnas es muy inusual y, a menudo, un diseño cuestionable.

3.1.1 Valores por defecto

Una columna se puede definir con un valor por defecto. Cuando se inserta una nueva fila y no se especifican valores para algunas de las columnas, esas columnas se llenarán con sus respectivos valores predeterminados.

Si no se declara explícitamente ningún valor predeterminado, el valor predeterminado es el valor nulo. Esto generalmente tiene sentido porque se puede considerar que un valor nulo representa datos desconocidos.

En una definición de tabla, los valores predeterminados se enumeran después del tipo de datos de la columna. Por ejemplo:

```
CREATE TABLE productos (
    num_producto integer,
    nombre text,
    precio numeric DEFAULT 9.99
);
```

El valor predeterminado puede ser una expresión, que se evaluará siempre que se inserte el valor predeterminado (no cuando se cree la tabla). Un ejemplo común es que una columna `timestamp` tenga un valor predeterminado de CURRENT_TIMESTAMP, de modo que se establezca en el momento de la inserción de la fila. Otro ejemplo común es generar un "número de serie " para cada fila. En PostgreSQL, esto generalmente se hace con algo como:

```
CREATE TABLE productos (
    num_producto integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

donde la función `nextval()` proporciona valores sucesivos de un objeto de secuencia. Esta disposición es lo suficientemente común como para que haya una abreviatura especial para ella:

```
CREATE TABLE productos (
    num_producto SERIAL,
    ...
);
```

Hablaremos más adelante de los tipos `SERIAL`.

3.2 Borrado de tablas

Si ya no necesitas una tabla, puedes eliminarla usando el comando `DROP TABLE` . Por ejemplo:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Intentar eliminar una tabla que no existe es un error. Sin embargo, es común en los archivos de script SQL intentar eliminar incondicionalmente cada tabla antes de crearla, ignorando cualquier mensaje de error, para que el script funcione independientemente de que la tabla exista o no. (Si lo deseas, puede usar la variante `DROP TABLE IF EXISTS` para evitar los mensajes de error, pero esto no es SQL estándar).

4. Tipos de datos

Postgresql tiene un conjunto de tipos muy rico. Además, los usuarios también pueden definir sus propios tipos a través de la sentencia `CREATE TYPE`.

La siguiente tabla muestra algunos de los tipos incorporados por Postgresql.

> Recuerda que el paréntesis no es algo propio de la sintaxis, sino que muestra algo optativo.

Nombre	Alias	Descripción
`bigint`	`int8`	Entero con signo de 8 bytes
`bigserial`	`serial8`	Entero autoincrementado de 8 bytes
`boolean`	`bool`	Valor booleano (`true`/`false`)
`character [ (n) ]`	`char [(n)]`	Cadena de caracteres de longitud fija
`character varying [ (n)]`	`varchar [ (n) ]`	Cadena de caracteres de longitud variable
`date`		Fecha (día, mes, año)
`double precision`	`float8`	Número en coma flotante de doble precisión (8 bytes)

Nombre	Alias	Descripción
`inet`		Dirección IP (IPv4 o IPv6)
`integer`	`int`, `int4`	Entero con signo (4 bytes)
`interval [ fields ] [ (p) ]`		Intervalo de tiempo
`numeric [ (p,s) ]`	`decimal [ (p,s) ]`	Número exacto de precisión seleccionable
`real`	`float4`	Número en coma flotante de precisión simple (4 bytes)
`smallint`	`int2`	Entero con signo de 2 bytes
`smallserial`	`serial2`	Entero autoincrementado de 2 bytes
`serial`	`serial4`	Entero autoincrementado de 4 bytes
`text`		Cadena de caracteres de longitud variable
`time [ (p) ] [without time zone]`		Hora del día sin zona horaria
`time [ (p) ] without time zone`	`timez`	Hora del día con zona horaria
`timestamp [ (p) ] [without time zone]`		Fecha y hora sin zona horaria
`timestamp [ (p) ] without time zone`	`timestampz`	Fecha y hora con zona horaria

4.1 Tipos numéricos

Los tipos numéricos consisten en enteros de 2, 4 y 8 bytes, números en coma flotante de 4 y 8 bytes, y números decimales de precisión seleccionable.

Nombre	Tamaño	Descripción	Rango
`smallint`	2 bytes	Entero pequeño	-32768 a 32676
`integer`	4 bytes	Número entero (más habitual)	-2147483648 a +2147483647
`bigint`	8 bytes	Número entero grande	-9223372036854775808 a +9223372036854775807
`decimal`	variable	precision indicada por usuario, exacto	hasta 131072 dígitos antes del punto decimal; hasta 16383 dígitos después del punto decimal
`numeric`	variable	precision indicada por usuario, exacto	hasta 131072 dígitos antes del punto decimal; hasta 16383 dígitos después del punto decimal

Nombre	Tamaño	Descripción	Rango
`real`	4 bytes	precisión variable, inexacto	6 dígitos decimales de precisión
`double precision`	8 bytes	precisión variable, inexacto	15 dígitos decimales de precisión
`smallserial`	2 bytes	entero autoincrementado pequeño	1 a 32767
`serial`	4 bytes	entero autoincrementado	1 a 2147483647
`bigserial`	8 bytes	entero autoincrementado grande	1 a 9223372036854775807

4.1.1 Tipos enteros

Los tipos enteros no aceptan decimales. Lo más normal es elegir el tipo `integer`. Si lo que prima es el espacio, y el rango nos sirve, podemos utilizar `smallint`. Si necesitamos más espacio, podemos utilizar `bigint`.

4.1.2 Tipos numéricos (`numeric`)

Los tipos `numeric` pueden almacenar números con una gran cantidad de dígitos. Son los más adecuados para guardar cantidades exactas. Sin embargo, los cálculos son más lentos que los realizados con los tipos enteros o en coma flotante.

La **precisión** de un `numeric` es el número total de dígitos significativos de todo el número. Por otro lado, la **escala** es del total de dígitos, la cantidad de dígitos de la parte decimal. Por ejemplo, el número 23.5141 tiene precisión 6 y escala 4.

> Podemos considerar los tipos enteros como tipos con escala 0.

La precisión y escala de un `numeric` se puede configurar de la siguiente forma:

```
NUMERIC(precision, escala)
```

Si la escala es 0, podemos usar:

```
NUMERIC(precision)
```

Si elegimos

```
NUMERIC
```

sin precisión ni escala, crearemos una columna que podrá almancenar valores con cualquier precisión y escala, hasta el límite de la implementación.

Si el número de decimales que tratamos de insertar es superior a la escala definida, Postgresql redondeará el valor para poder almacenarlo con el número de decimales definido. Si el número de dígitos a la izquierda del punto decimal es superior al definido, lanzará un error.

4.1.3 Tipos en coma flotante

Los tipos `real` y `double precision` son tipos numéricos inexactos de precisión variable. Estos tipos se implementan siguiendo el estándar IEEE 754 https://es.wikipedia.org/wiki/IEEE_754

Inexacto significa que algunos valores no se pueden convertir exactamente al formato interno y se almacenan como aproximaciones, por lo que el almacenamiento y la recuperación de un valor pueden mostrar ligeras discrepancias. La gestión de estos errores y cómo se propagan a través de los cálculos es el tema de toda una rama de las matemáticas y la informática y no se discutirá aquí, excepto por los siguientes puntos:

- Si necesitas almacenamiento y cálculos exactos (por ejemplo, para cantidades monetarias), utiliza el tipo `numeric` en su lugar.
- Si deseas hacer cálculos complicados con estos tipos para algo importante, especialmente si confías en cierto comportamiento en casos de límites (infinito, subdesbordamiento), debes evaluar la implementación con cuidado.
- Es posible que comparar dos valores de coma flotante para determinar la igualdad no siempre funcione como se esperaba.

> Si quieres saber más sobre números en coma flotante en Postgresql puedes consultra aquí https://www.postgresql.org/docs/current/datatype-numeric.html#DATATYPE-FLOAT

4.1.4 Tipos seriales

Los tipos `smallserial`, `serial` y `bigserial` no son realmente tipos, sino una notación conveniente para crear columnas que sirvan como identificadores (es decir, claves primarias). Al identificar una columna como `SERIAL`, Postgresql hace varias cosas por nosotros:

```
CREATE TABLE tablename (
    colname SERIAL
);
```

Es equivalente a lo siguiente:

```
CREATE SEQUENCE tablename_colname_seq AS integer;
CREATE TABLE tablename (
    colname INTEGER NOT NULL DEFAULT nextval('tablename_colname_seq')
```

```
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

De esta forma, lo que realmente sucede es que creamos una columna entera, y asingamos su valor por defecto a un generador de secuencia. La restricción `NOT NULL` se asegura de que no se inserte ningún valor nulo explícitamente. La última sentencia, en la que se asigna a la columna como propietaria de la secuencia, hace que si se borra la tabla, también se borre la secuencia.

4.2 Tipos de cadenas de caracteres

En Postgresql tenemos, básicamente, 3 tipos de cadenas de caracteres:

- `character(n)`, `char(n)`: cadena de caracteres de longitud fija.
- `character varying(n), varchar(n)`: cadena de caracteres de longitud variable con límite.
- `text`: cadena de caracteres de longitud variable sin límite.

Para los dos primeros tipo, el valor `(n)` es la cantidad caracteres (que no de bytes) que puede albergar como máximo.

> No hay diferencia de rendimiento entre estos tres tipos, aparte del mayor espacio de almacenamiento cuando se usa el tipo con relleno en blanco y algunos ciclos de CPU adicionales para verificar la longitud cuando se almacena en una columna con longitud limitada. Si bien el `char(n)` tiene ventajas de rendimiento en algunos otros sistemas de bases de datos, no existe tal ventaja en PostgreSQL; de hecho, el tipo `char(n)` suele ser el más lento de los tres debido a sus costos de almacenamiento adicionales. En la mayoría de las situaciones, se debe usar `text` o `varchar`.

4.3 Tipos de datos con fechas

Los hemos trabajado ampliamente en temas anteriores.

> Si quieres saber más sobre los tipos de datos de fechas y horas, puedes consultar aquí https://www.postgresql.org/docs/current/datatype-datetime.html.

4.4 Columnas generadas

Quizás este no es el mejor apartado para mencionar esta funcionalidad de Postgresql, si bien es cierto que puede ayudarnos a contextualizarnos.

Postgresql permite definir columnas de todos los tipos de datos anteriores (y muchos más). Y además, nos permite definir columnas cuyos valores se obtienen como el procesamiento de otros valores. De alguna forma, las columnas generadas son a las columnas lo que las vistas a las tablas.

Hay dos tipos de columnas generadas: **almacenadas** y **virtuales**. Una columna generada almacenada se calcula en el momento de escritura (inserción o actualización) y ocupa espacio de almacenamiento. Una columna generada virtual no ocupa espacio de almacenamiento y se calcula al leer dicha fila. Postgresql **solamente implementa las columnas generadas almacenadas**.

```
CREATE TABLE televisores (
    ...,
    diagonal_in numeric,
    diagonal_cm numeric GENERATED ALWAYS AS (diagonal_in * 2.54) STORED
);
```

Una columna generada no se puede escribir directamente con una sentencia del DML. Sí que tenemos que especificar la palabra `DEFAULT`.

La columnas generadas tienen algunas restricciones:

- La expresión de generación no puede usar subconsultas o referenciar valores fuera de la fila actual.
- No puede referenciar otra columna generada
- No se pueden referenciar valores de columnas del sistema salvo `tableoid`.

5. Restricciones

Los tipos de datos son una manera muy sencilla de limitar los datos que se pueden almacenar en las columnas de una tabla. Sin embargo, para la mayoría de las aplicaciones estas restricciones son demasiado limitadas. Por ejemplo, una columna que almacene el precio de un producto probablemente solamente pueda aceptar valores positivos. Sin embargo, no hay un tipo de dato *número positivo*. Otra problema puede ser querer restringir los valores de una columna que referencian a otras columnas o filas. Por ejemplo, en una tabla de información sobre productos, solamente debería haber una fila por cada identificador de producto.

SQL define una serie de restricciones sobre columnas y tablas. Estas restricciones nos proporcionan mucho más control sobre los datos que se van a almacenar. Si un usuario trata de almacenar datos en una columna que violan alguna de esas restricciones, se lanzará un error.

5.1 Restricciones *check*

La restricción *check* es el tipo más genérico de restricción. Este permite especificar que el valor de una columna debe satisfacer una condición booleana. Por ejemplo, para requerir que el precio de producto sea positivo podríamos utilizar el siguiente DDL:

```
CREATE TABLE producto (
    num_producto    INTEGER,
    nombre          TEXT,
    precio          NUMERIC CHECK (precio > 0)

);
```

Como se puede comprobar, la restricción se puede escribir a continuación del tipo de dato. Una restricción de tipo *check* consiste en la palabra reservada `check` seguida de una expresión entre paréntesis. Esta expresión debería incluir el nombre de la columna a restringir (si no, no tendría mucho sentido).

También se puede definir la restricción con un nombre, lo cual puede clarificar los mensajes de error que nos devuelva Postgresql en caso de violar la restricción.

```
CREATE TABLE producto (
    num_producto    INTEGER,
    nombre          TEXT,
    precio          NUMERIC CONSTRAINT precio_positivo CHECK (precio > 0)

);
```

Estas restricciones también se puede incluir tras la definición del resto de columnas:

```
CREATE TABLE producto (
    num_producto    INTEGER,
    nombre          TEXT,
    precio          NUMERIC,
    CONSTRAINT precio_positivo CHECK (precio > 0)

);
```

Una restricción puede hacer referencia a más de una columna. Por ejemplo, supongamos que almacenamos el precio regular y el precio descontado, y queremos asegurarnos de que el precio descontado es siempre menor que el precio regular:

```
CREATE TABLE producto (
    num_producto        INTEGER,
    nombre              TEXT,
    precio              NUMERIC,
    precio_descontado   NUMERIC,
    CONSTRAINT precio_positivo CHECK (precio > 0),
    CONSTRAINT precio_descontado_positivo CHECK (precio_descontado > 0),
    CONSTRAINT descuento_valido CHECK (precio_descontado < precio)
);
```

> Antes hacíamos referencia a restricciones de **columna** y restricciones de **tabla**. En Postgresql, las restricciones de columna son aquellas que vienen definidas justo después del tipo de dato de una columna, y las restricciones de tabla son aquellas que se definen aparte. Por tanto, podemos ver que toda restricción de columna puede ser descrita como una restricción de tabla.

> **Para nosotros, siempre será preferible el uso de la notación de restricción de tabla con nombre, es decir: `CONSTRAINT nom_restriccion ...`. Más adelante daremos una regla mnemotécnica para nombrar las restricciones**.

5.2 Restricción de obligatoriedad o *not-null*.

La restricción de obligatoriedad provoca que una determinada columna no pueda albergar valores nulos, y por tanto, sea obligatorio porporcionar un valor de dicha columna para nuevas filas.

```
CREATE TABLE producto (
    num_producto    INTEGER NOT NULL,
    nombre          TEXT NOT NULL,
    precio          NUMERIC

);
```

Las restricciones `NOT NULL` siempre se escriben **como restricciones de columna**. Si quisiéramos escribirlas como *restricciones de tabla* las tendríamos que transformar en una restricción *check*: `CHECK (columna IS NOT NULL)`, pero en Postgresql las restricciones `NOT NULL` son más eficientes.

> Será el único caso de restricción que no definiremos como **restricción de tabla** añadiendo un nombre, dado que son más eficients que su equivalente, como hemos visto en el párrafo anterior.

5.3 Restricciones de unicidad o `unique`

Las restricciones de unicidad aseguran que los datos contenidos en **una sola columna** o en **un conjunto de columnas** es único para todas las filas de la tabla, y por tanto no se repite.

```
CREATE TABLE producto (
    num_producto    INTEGER UNIQUE,
    nombre          TEXT,
    precio          NUMERIC

);
```

Se puede escribir como una restricción de tabla de la siguiente forma:

```
CREATE TABLE producto (
    num_producto    INTEGER,
    nombre          TEXT,
    precio          NUMERIC,
    CONSTRAINT num_producto_unico UNIQUE (num_producto)

);
```

La única forma de definir una restricción de unicidad que afecte a más de una columna es como una restricción de tabla.

```
CREATE TABLE temperatura_diaria (
    id_ciudad       integer,
    fecha           date,
```

```
    maxima          numeric,
    minima          numeric,
    media           numeric,
    CONSTRAINT ciudad_fecha_unicas UNIQUE (id_ciudad, fecha)
);
```

> Hay que tener en cuenta que los valores nulos no cuentan como repetidos para las columnas definidas como únicas.

5.4 Restricción de clave primaria

Una restricción de clave primaria indica que una columna, o un grupo de columnas, se puede utilizar como un identificador único para las filas de la tabla. Esto requiere que los valores sean únicos y no nulos. Entonces, las siguientes dos definiciones de tabla aceptan los mismos datos:

```
CREATE TABLE producto (
    num_producto        INTEGER UNIQUE NOT NULL,
    nombre              TEXT,
    precio              NUMERIC
);
```

```
CREATE TABLE producto (
    num_producto        INTEGER PRIMARY KEY,
    nombre              TEXT,
    precio              NUMERIC
);
```

> Tal y como lo define el modelo relacional, **una tabla solamente puede tener una restricción de clave primaria**.

Las restricciones de clave primaria también se pueden definir como *restricciones de tabla*:

```
CREATE TABLE producto (
    num_producto        INTEGER,
    nombre              TEXT,
    precio              NUMERIC,
    CONSTRAINT pk_productos PRIMARY KEY (num_producto)
);
```

El formato como *restricción de tabla* es el único posible cuando queremos definir una clave primaria **compuesta** por más de un atributo:

```
CREATE TABLE temperatura_diaria (
    id_ciudad       integer,
```

```
    fecha           date,
    maxima          numeric,
    minima          numeric,
    media           numeric,
    CONSTRAINT pk_temperatura_diaria PRIMARY KEY (id_ciudad, fecha)
);
```

Cuando se añade una restricción de clave primaria a una tabla, se crea un índice para la columna o conjunto de columnas afectadas por la restricción.

5.5 Restricciones de clave externa

Una restricción de clave externa nos permite implementar la **regla de integridad referencial**, es decir, especifica que el valor de una columna (o grupo de columnas) deben coincidir con los valores de la columna a la cual hacen referencia.

Supongamos la tabla `producto` de los ejemplos anteriores:

```
CREATE TABLE producto (
    num_producto        INTEGER,
    nombre              TEXT,
    precio              NUMERIC,
    CONSTRAINT pk_producto PRIMARY KEY (num_producto)
);
```

Supongamos también que tenemos otra tabla donde se almacenan los pedidos de esos productos. Queremos asegurar que los pedidos se realizan solamente de productos que existan realmente. Para ello, podemos definir una restricción de clave externa:

```
CREATE TABLE pedido (
    id_pedido       INTEGER,
    num_producto    INTEGER REFERENCES producto (num_producto),
    cantidad        INTEGER,
    CONSTRAINT pk_pedido PRIMARY KEY (id_pedido)
);
```

De esta forma, sería imposible insertar una fila en `pedido` que tuviera un valor no nulo en la columna `num_producto` que no fuese un valor que actualmente está insertado en la tabla `producto`.

Decimos de esta forma que la tabla `pedido` referencia a la tabla `producto`, y que la tabla `producto` es referenciada.

La restricción anterior se puede escribir de forma abreviada:

```
CREATE TABLE pedido (
    id_pedido       INTEGER,
```

```
    num_producto    INTEGER REFERENCES producto,
    cantidad        INTEGER,
    CONSTRAINT pk_pedido PRIMARY KEY (id_pedido)
);
```

Si una tabla referencia a otra, quiere decir que está haciendo referencia a su clave primaria (y ya hemos visto que una tabla no debe tener más de una clave primaria).

Una restricción de clave externa puede estar definida sobre más de un atributo (por ejemplo, cuando una clave primaria sea compuesta, y la referenciemos desde otra tabla).

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Una tabla puede tener más de una restricción de clave externa: bien porque esté asociada con tablas diferentes, o bien porque esté asociada más de una vez con la misma tabla. Aquí, dos ejemplos:

```
CREATE TABLE producto (
    num_producto        INTEGER,
    nombre              TEXT,
    precio              NUMERIC,
    CONSTRAINT pk_producto PRIMARY KEY (num_producto)
);

CREATE TABLE pedido (
    id_pedido           INTEGER,
    direccion_envio     TEXT,
    CONSTRAINT pk_pedido PRIMARY KEY (id_pedido)
);

CREATE TABLE items_pedido (
    num_producto        INTEGER,
    id_pedido           INTEGER,
    cantidad            INTEGER,
    CONSTRAINT pk_items_pedido PRIMARY KEY (num_producto, id_pedido),
    CONSTRAINT fk_items_pedido_producto FOREIGN KEY (num_producto)
REFERENCES producto,
    CONSTRAINT fk_items_pedido_pedido FOREIGN KEY (id_pedido) REFERENCES
pedido
);
```

Y aquí el ejemplo de dos claves externas a una misma tabla, ambas con significado diferente:

```
CREATE TABLE AEROPUERTO
(
    ID_AEROPUERTO           INTEGER,
    NOMBRE                  VARCHAR(100),
    CIUDAD                  VARCHAR(100),
    CONSTRAINT PK_AEROPUERTO PRIMARY KEY (ID_AEROPUERTO)
);

CREATE TABLE AVION
(
    ID_AVION                INTEGER,
    NOMBRE                  VARCHAR(50),
    MAX_PASAJEROS           INTEGER,
    TIPO_PASILLO            VARCHAR(30),
    CONSTRAINT PK_AVION PRIMARY KEY (ID_AVION)
);


CREATE TABLE VUELO
(
    ID_VUELO                INTEGER,
    DESDE                   INTEGER,
    HASTA                   INTEGER,
    SALIDA                  TIMESTAMP,
    LLEGADA                 TIMESTAMP,
    PRECIO                  NUMERIC(7,2),
    DESCUENTO               INTEGER,
    ID_AVION                INTEGER,
    CONSTRAINT PK_VUELO PRIMARY KEY (ID_VUELO),
    CONSTRAINT FK_VUELO_AERO1 FOREIGN KEY (DESDE) REFERENCES AEROPUERTO,
    CONSTRAINT FK_VUELO_AERO2 FOREIGN KEY (HASTA) REFERENCES AEROPUERTO,
    CONSTRAINT FK_VUELO_AVION FOREIGN KEY (ID_AVION) REFERENCES AVION
);
```

5.5.1 Políticas de borrado para restricciones de clave externa

Supongamos el ejemplo anterior de las tablas `pedido`, `producto`, ... Sabemos que las claves externas no permiten la creación de pedidos que no se relacionen con ningún producto. Pero, ¿qué sucede si se elimina un producto después de que se crea un pedido que hace referencia a él? SQL también le permite manejar eso. Si recordamos la teoría sobre el modelo relacional, tenemos algunas opciones:

- No permitir la eliminación de un producto referenciado
- Eliminar los pedidos también
- ¿Algo más?

Para ilustrar esto, implementemos la siguiente política en el ejemplo anterior: cuando alguien quiere eliminar un producto al que todavía se hace referencia en un pedido (vía `items_pedido`), lo rechazamos. Si alguien elimina un pedido, los artículos del pedido también se eliminan:

```
CREATE TABLE producto (
    num_producto        INTEGER,
    nombre              TEXT,
    precio              NUMERIC,
    CONSTRAINT pk_producto PRIMARY KEY (num_producto)
);

CREATE TABLE pedido (
    id_pedido           INTEGER,
    direccion_envio     TEXT,
    CONSTRAINT pk_pedido PRIMARY KEY (id_pedido)
);

CREATE TABLE items_pedido (
    num_producto        INTEGER,
    id_pedido           INTEGER,
    cantidad            INTEGER,
    CONSTRAINT pk_items_pedido PRIMARY KEY (num_producto, id_pedido),
    CONSTRAINT fk_items_pedido_producto FOREIGN KEY (num_producto)
REFERENCES producto ON DELETE RESTRICT,
    CONSTRAINT fk_items_pedido_pedido FOREIGN KEY (id_pedido) REFERENCES
pedido ON DELETE CASCADE
);
```

El borrado en cascada y la restricción del borrado son dos de las operaciones más habituales. `RESTRICT` previene el borrado de filas referenciadas. `NO ACTION` significa que si alguna fila referenciada existe aun cuando la restricción es comprobada, se lanzará un error. Este es el comportamiento por defecto y no hay por qué especificarlo (la diferencia entre `RESTRICT` y `NO ACTION` es algo sutil, y necesitamos conocer las transacciones para comprenderla bien).

`CASCADE` especifica que si se borra una fila referenciada, las filas que hacían referencia a esta también se borrarán.

> ¡CUIDADO! El borrado en cascada es una herramienta, pero peligrosa.

> Existen otras dos opciones, que son menos habituales: `SET NULL` y `SET DEFAULT`. Su comportamiento es el obvio: `SET NULL` deja los valores referenciantes a `NULL`, y `SET DEFAULT` les establece el valor por defecto para esa columna.

De forma análoga a `ON DELETE`, podemos añadir `ON UPDATE`, que gestiona todas estas políticas pero asociadas a la actualización del valor de la clave primaria.

> Si bien este es un comportamiento que se puede dar, nosotros trataremos de no modificar nunca el valor de una clave primaria, sobre todo en el caso de escoger como tipos de datos `integer` o `serial` (o alguno de sus derivados).

5.6 Pautas para nombrar las restricciones

Para facilitar la trazabilidad de los errores en caso de que se produzcan al violar una restricción, deberíamos ser nosotros quiénes pongan nombre a las mismas, y no delegar esta tarea Postgresql, ya que los nombres que nosotros proporcionemos serán más inteligibles y podrán tener un cierto sentido.

> Como norma general, Postgresql permite que el nombre de las restricciones sea de un **máximo de 63 caracteres**.

Para nombrar las restricciones, usaremos la siguiente regla: `tiporestriccion_nombredescriptivo`

Para el tipo, usaremos las siguientes siglas

- `ck`: restricción check
- `uk`: restricción de unicidad
- `pk`: restricción de clave primaria
- `fk`: restricción de clave externa

El nombre descriptivo dependerá del tipo de restricción.

- `ck_pedido_total_positivo`: para las restricciones `check`, se puede incluir el nombre de la tabla (o una abreviatura) y una descripción de la condición.
- `uk_temperatura_fecha`: para la restricciones `unique`, se puede incluir el nombre de la talba y la columna o columnas afectadas por la restricción.
- `pk_producto`: para las de clave primaria, basta con incluir el nombre de la tabla.
- `fk_items_pedido_producto`: para las de clave externa, se puede incluir la tabla referenciante y la tabla referenciada.

6. Manejo de tablas II: modificación de tablas

Si bien seguramente no vamos a estar modificando el esquema de nuestras tablas todos los días, sí que es posible que una vez que las hayamos creado tengamos que incluir algún tipo de modificación. Si una tabla ya tiene datos insertados o si otros objetos de la base de datos hacen referencia a ella, no podremos eliminarla y volver a crearla con las nuevas especificaciones, y por tanto tendremo que modificarla. PostgreSQL proporciona una familia de comandos para realizar modificaciones en tablas existentes.

Las operaciones permitidas son:

- Agregar columnas
- Quitar columnas
- Agregar restricciones
- Eliminar restricciones
- Cambiar los valores predeterminados
- Cambiar tipos de datos de columna
- Cambiar el nombre de las columnas
- Cambiar el nombre de las tablas

6.1 Añadir nuevas columnas

Para añadir una nueva columna a una tabla puedes usar el siguiente comando:

```
ALTER TABLE producto ADD COLUMN descripcion text;
```

La nueva columna se rellenará con el valor por defecto, si está definido, o por valores `NULL` en otro caso.

También puedes definir restricciones en la columna al mismo tiempo, utilizando la sintaxis habitual:

```
ALTER TABLE producto ADD COLUMN descripcion text CHECK (descripcion !=
'');
```

De hecho, podemos utilizar prácticamente todos los elementos sintácticos que vimos al utilizar `CREATE TABLE`.

6.2 Eliminar una columna

Para eliminar una columna puedes utilizar el siguiente comando:

```
ALTER TABLE producto DROP COLUMN descripcion;
```

Todos los datos que hubiera en esta columna desaparecerán, así como las restricciones que afectaran a esta columna. Si la columna es referenciada por alguna clave externa, Postgresql no borrará esa restricción. Se le puede pedir que lo haga si añadimos `CASCADE`.

```
ALTER TABLE producto DROP COLUMN descripcion CASCADE;
```

6.3 Añadir y eliminar una restricción

Añadir una restricción a una tabla es muy sencillo. Tan solo tenemos que mezclar la sintaxis de `ALTER TABLE` con la que ya conocemos de definición de restricciones:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES
product_groups;
```

Para agregar una restricción de prohibición de nulos, que no se puede escribir como una restricción de tabla, podemos utilizar esta sintaxis:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

La restricción se comprobará inmediatamente, por lo que los datos de la tabla deben satisfacer la restricción antes de poder agregarla.

Para eliminar una restricción, necesitamos saber su nombre. Si le dio un nombre, entonces es fácil. De lo contrario, el sistema asignó un nombre generado, que debemos averiguar. El comando `psql` puede resultar útil aquí; también podemos buscar las restricciones a través de Pgadmin.

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

Al igual que con la eliminación de una columna, debemos agregar `CASCADE` si deseamos eliminar una restricción de la que depende otra cosa. Un ejemplo es que una restricción de clave externa dependa de una restricción de clave primaria o de unidad en las columnas referenciadas.

Esto funciona igual para todos los tipos de restricciones, excepto las restricciones no nulas. Para eliminar una restricción no nula, que como sabemos no tiene nombre, debemos utilizar:

```
ALTER TABLE productos ALTER COLUMN product_no DROP NOT NULL;
```

6.4 Cambiar un valor por defecto

Para establecer un nuevo valor predeterminado para una columna, usaremos un comando como:

```
ALTER TABLE productos ALTER COLUMN precio SET DEFAULT 7.77;
```

Ten en cuenta que esto no afecta a ninguna fila existente en la tabla, solo cambia el valor predeterminado para comandos `INSERT` futuros.

Para eliminar cualquier valor predeterminado, podemos utilizar:

```
ALTER TABLE productos ALTER COLUMN precio DROP DEFAULT;
```

Esto es efectivamente lo mismo que establecer el valor predeterminado en nulo. Como consecuencia, no es un error eliminar un valor predeterminado donde no se ha definido uno, porque el valor predeterminado es implícitamente el valor nulo.

6.5 Cambiar el tipo de dato

Para convertir una columna a un tipo de datos diferente, utiliza un comando como:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

Esto solo tendrá éxito si cada entrada existente en la columna se puede convertir al nuevo tipo mediante una conversión implícita. Si se necesita una conversión más compleja, puedes agregar una cláusula `USING` que especifique cómo calcular los nuevos valores a partir de los antiguos.

```
ALTER TABLE con_fecha ALTER COLUMN fecha TYPE timestamp USING
fecha::timestamp;
```

PostgreSQL intentará convertir el valor predeterminado de la columna (si existe) al nuevo tipo, así como cualquier restricción que involucre a la columna. Pero estas conversiones pueden fallar o producir resultados inesperados. A menudo es mejor eliminar cualquier restricción en la columna antes de modificar su tipo y luego volver a agregar las restricciones modificadas adecuadamente.

6.6 Renombrar una columna

Para renombrar una columna usaremos expresiones como:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

6.7 Renombrar una tabla

Para renombrar una tabla usaremos expresiones como:

```
ALTER TABLE products RENAME TO items;
```

7. Manipulación de datos

Los apartados anteriores nos mostraron cómo crear tablas y otras estructuras para contener sus datos. Ahora es el momento de llenar las tablas con datos. Este apartado cubre cómo insertar, actualizar y eliminar datos de las tablas.

7.1 Operaciones de inserción

Cuando se crea una tabla, no contiene datos. Lo primero que debemos hacer antes de que una base de datos pueda ser de mucha utilidad es insertar datos. Los datos se insertan conceptualmente una fila a la vez. Por supuesto, también puede insertar más de una fila, pero no hay forma de insertar menos de una fila. Incluso si solo conoce algunos valores de columna, se debe crear una fila completa.

Para crear una nueva fila, se utiliza el comando `INSERT`. El comando requiere el nombre de la tabla y los valores de las columnas. Por ejemplo, pensemos en la siguiente definición de tabla:

```
CREATE TABLE producto (
    num_producto        INTEGER,
```

```
    nombre              TEXT,
    precio              NUMERIC,
    CONSTRAINT pk_productos PRIMARY KEY (num_producto)
);
```

Un ejemplo de comando de inserción de una fila sería:

```
INSERT INTO producto VALUES (1, 'Queso', 9.99);
```

Los valores de los datos se deben listar en el orden que han sido definidas las columnas, separados por comas. Normalmente, esos datos serán literales (constantes), pero también pueden ser expresiones escalares.

> Expresiones escalares como **subconsultas escalares**.

La siguiente sintaxis nos muestra como podemos modificar el orden de las columnas en la tabla a la hora de insertar.

```
INSERT INTO productos (num_producto, nombre, precio) VALUES (1, 'Queso',
9.99);
INSERT INTO productos (nombre, precio, num_producto) VALUES ('Queso',
9.99, 1);
```

> Esto no modifica de forma permanente el orden de las columnas, sino que nos permite indicar otro orden para los valores a insertar.

Si no tieneS valores para todas las columnas, puedeS omitir algunas de ellas. En ese caso, las columnas se llenarán con sus valores predeterminados (que pueden ser el valor `null`). Por ejemplo:

```
INSERT INTO productos (num_producto, nombre) VALUES (1, 'Cheese');
INSERT INTO productos VALUES (1, 'Cheese');
```

La segunda forma es una extensión de PostgreSQL. Llena las columnas de la izquierda con tantos valores como se den, y el resto quedará predeterminado.

Para mayor claridad, también puede solicitar valores predeterminados explícitamente, para columnas individuales o para toda la fila:

```
INSERT INTO productos (num_producto, nombre) VALUES (1, 'Queso', DEFAULT);
INSERT INTO productos DEFAULT VALUES;
```

Se pueden insertar múltiples filas en un solo comando:

```
INSERT INTO productos (num_producto, nombre, precio) VALUES
        (1, 'Queso', 9.99),
        (2, 'Pan', 0.99),
        (3, 'Leche', 1.25);
```

También es posible insertar el resultado de una consulta (que puede devolver cero filas, una fila o más de una fila)

```
INSERT INTO productos (num_producto, nombre, precio)
  SELECT num_producto, nombre, precio FROM nuevos_productos
    WHERE fecha_publicacion = 'today';
```

> Si lo que se van a insertar es una gran cantidad de filas, hay que considerar el uso de `COPY`. No es tan flexible como `INSERT`, pero es mucho más eficiente. Puedes consultar este enlace de la documentación (en inglés) para obtener más información sobre como mejorar el rendimiento.

7.2 Actualización de datos

La modificación de datos que ya se encuentran en la base de datos se denomina actualización. Se pueden actualizar filas individuales, todas las filas de una tabla o un subconjunto de todas las filas. Cada columna se puede actualizar por separado; las otras columnas no se ven afectadas.

Para actualizar filas existentes, se usa `UPDATE`. Estas sentencias tienen tres partes:

- El nombre de la tabla y la columna para actualizar
- El nuevo valor de la columna
- Qué fila (s) actualizar

Para escoger qué fila (o filas) se van a actualizar hay que especificar qué condiciones debe cumplir una fila para que se actualice. La única forma de garantizar que solamente se actualice una fila concreta es declarando la condición basada en la clave primaria.

Por ejemplo, este comando actualiza todos los productos que tienen un precio de 5 para que tengan un precio de 10:

```
UPDATE productos SET precio = 10 WHERE precio = 5;
```

Esto puede hacer que se actualicen cero, una o muchas filas. No es un error intentar una actualización que no coincide con ninguna fila.

Veamos ese comando en detalle. Primero está la palabra clave `UPDATE`, seguida del nombre de la tabla. Como de costumbre, el nombre de la tabla puede estar calificado por esquema; de lo contrario, se busca en la ruta. La siguiente es la palabra clave `SET`, seguida por el nombre de la columna, un signo igual y el nuevo

valor de la columna. El nuevo valor de columna puede ser cualquier expresión escalar, no solo una constante. Por ejemplo, si desea aumentar el precio de todos los productos en un 10%, se puede usar:

```
UPDATE productos SET precio = precio * 1.10;
```

Como puedes ver, la expresión del nuevo valor puede referirse a los valores existentes en la fila. También hemos omitido en este caso la cláusula `WHERE`. Si se omite, significa que se actualizan todas las filas de la tabla. Si está presente, solo se actualizan las filas que coinciden con la condición `WHERE`. Ten en cuenta que el signo igual en la cláusula `SET` es una asignación, mientras que el de la cláusula `WHERE` es una comparación, si bien esto no crea ninguna ambigüedad.

Puedes actualizar más de una columna en una sentencia `UPDATE` enumerando más de una asignación en la cláusula `SET`. Por ejemplo:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

7.3 Eliminación de filas

Al igual que solamente podemos insertar filas completas en una tabla, también podemos borrar solamente filas completas. Esta eliminación de filas también se hace basándonos en una condición (como las operaciones de actualización); solamente podemos garantizar que una operación de borrado afectará como mucho a una fila si la condición se basa en la clave primaria.

El comando para borrar filas es `DELETE`. Por ejemplo, si queremos eliminar todas las filas de la tabla `productos` que tengan un precio igual a 10:

```
DELETE FROM productos WHERE precio = 10;
```

Si queremos borrar todas las filas, podríamos usar la siguiente sentencia:

```
DELETE FROM productos;
```

7.4 Cómo devolver datos de filas modificadas

En algunas ocasiones es de utilidad obtener datos de filas recién modificadas. Los comandos `INSERT`, `UPDATE` y `DELETE` tienen la claúsula opcional `RETURNING` que nos permite realizar esta operación. Esta realiza una consulta adicional para recoger los datos solicitados, y es especialmente útil en algunas situaciones, como los identificadores de tipo `SERIAL`.

En una sentencia `INSERT`, los datos disponibles en la claúsula `RETURNING` es la fila recién insertada. Es especialmente útil para campos calculados o identificadores, como decíamos antes:

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING
id;
```

> También podemos utilizar `RETURNING` en un `INSERT ... SELECT`.

En una sentencia `UPDATE` podemos devolver con `RETURNING` los nuevos valores modificados:

```
UPDATE productos SET precio = precio * 1.10 WHERE precio <= 99.99
    RETURNING nombre, precio AS nuevo_precio;
```

En una sentencia `DELETE`, los datos disponibles son el contenido de la fila eliminada:

```
DELETE FROM productos
    WHERE fecha_tope = 'today'
    RETURNING *;
```

8. Transacciones

Es fundamental conocer el concepto de transacción en los sistemas gestores de bases de datos, como PostgreSQL. Una **transacción empaqueta varios pasos en una operación**, de forma que se completen todos o ninguno. Los estados intermedios entre los pasos no son visible para otras transacciones ocurridas en el mismo momento.

En el caso de que ocurra algún fallo que impida que se complete la transacción, ninguno de los pasos se ejecutan y no afectan a los objetos de la base de datos.

Los pasos dentro de una transacción son varias sentencias SQL que deben de completarse todas para que queden registradas. Para **comenzar una transacción utilizamos el comando `BEGIN`**. Para indicar al sistema que han **terminado correctamente todas las sentencias SQL, utilizamos el comando `COMMIT`**. Hay ocasiones en las que tenemos que desechar algunos de los pasos que se están realizando. Para **cancelar la transacción** comenzada utilizamos el **comando `ROLLBACK`**. Veamos estos comando con ejemplos.

> Si quieres profundizar más en este tema, puedes consultar la documentación oficial sobre concurrencia en Postgresql

Para poder utilizar estos comando mencionados (`BEGIN`, `COMMIT` y `ROLLBACK`) debemos de desactivar el `AUTOCOMMIT`. Ésta opción es a nivel de cliente y por defecto está activada. De forma que toda sentencia ejecutada queda confirmada y registrada en la base de datos. Para poder desactivarlo, podemos ejecutar la sentencia:

```
SET AUTOCOMMIT OFF
```

8.1 Comando BEGIN

Cuando utilizamos este comando el sistema permite que se ejecuten todas las sentencias SQL que necesitemos y las registra en un fichero. A continuación os dejo un ejemplo donde se comienza una transacción donde deben de completarse satisfactoriamente todas las sentencias.

```
BEGIN;

UPDATE cuentas SET balance = balance - 100.00 WHERE n_cuenta = 0127365;

UPDATE cuentas SET balance = balance + 100.00 WHERE n_cuenta = 0795417;
```

8.2 Comando COMMIT

Cuando ejecutamos este comando estamos confirmando que todas las sentencias son correctas. Así pues, hasta que no se ejecute el comando `COMMIT`, las sentencias no quedarán registradas. Por ejemplo, si cerramos la conexión antes de ejecutar este comando no se verá afectada ninguna de las tablas de la base de datos. A continuación os dejo un ejemplo en el cual se comienza una transacción, se ejecutan una serie de pasos y confirmamos que todas las sentencias están correctas.

```
BEGIN;

INSERT INTO cuentas (n_cuenta, nombre, balance) VALUES (0679259, 'Pepe',
200);

UPDATE cuentas SET balance = balance - 137.00 WHERE nombre = 'Pepe';

UPDATE cuentas SET balance = balance + 137.00 WHERE nombre = 'Juan';

SELECT nombre, balance FROM cuentas WHERE nombre = 'Pepe' AND nombre =
'Juan';

COMMIT;
```

8.3 Comando ROLLBACK

Con este comando podemos desechar las sentencias que se hayan ejecutado en una transacción y evitar así que se modifiquen los datos de nuestra base de datos. A continuación os dejo un ejemplo donde vamos a anular la transacción una transacción:

```
BEGIN;
"SENTENCIAS SQL"
```

```
ROLLBACK;
```

8.3.1 Uso de `SAVEPOINT`

El comando `SAVEPOINT` establece un nuevo punto de guardado en la transacción actual. Un punto de guardado es una marca especial dentro de una transacción que permite revertir todos los comandos que se ejecutan después de que se estableció, restaurando el estado de la transacción a lo que era en el momento del punto de guardado.

La sintaxis es `SAVEPOINT nombre_punto_guardado`.

8.3.2 `ROLLBACK TO SAVEPOINT`

Podemos utilizar la variante `ROLLBACK TO SAVEPOINT` para deshacer los cambios realizados hasta un determinado punto de guardado; los anteriores no se deshacen, pero quedarían pendientes de confirmación:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

La transacción anterior insertaría los valores `1` y `3`, pero no el `2`.

Para establecer y destruir después un punto de guardado, podemos usar el comando `RELEASE SAVEPOINT`

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

La anterior transacción insertaría tanto `3` como `4`.