UD 03: Introduccón a SQL

- UD 03: Introduccón a SQL
  - 1. PostgreSQL
  - 2. ¿Qué otras alternativas podríamos usar?
  - 3. Instalación de PostgreSQL
    - 3.1 Instalación en Windows
    - 3.2 Instalación en Linux
    - 3.3 Instalación en Mac
  - 4. Introducción a SQL
    - 4.1 Objetivos
    - 4.2 Historia
    - 4.3 Funcionamiento
      - 4.3.1 Algunos conceptos para usar SQL con Postgresql
      - 4.3.2 Usuarios que intervienen
      - 4.3.3. Crear nueva conexión
  - 5. Consultas de datos con SQL. DQL.
    - 5.1 Sintaxis sencilla del comando `SELECT`
    - 5.2 Consultar un subconjunto de columnas
      - 5.2.1 Uso de alias
    - 5.3 Uso de `DISTINCT` para obtener valores únicos
    - 5.4 Ordenación de los datos con `ORDER BY`.
    - 5.5 Uso de condiciones
      - 5.5.1 Uso de `LIKE` y `ILIKE`
      - 5.5.2 Combinación de condiciones con `AND` y `OR`.
      - 5.5.3 Uso de valores nulos
      - 5.5.4 Precedencia de operadores
    - 5.6 Concatenación de textos
  - 6 Realización de cálculos en SQL
    - 6.1 Cálculo de porcentajes
    - 6.2 Funciones
      - 6.2.1 Funciones numéricas
        - 6.2.1.1 Funciones que trabajan fila a fila
        - 6.2.1.2 Funciones que agregan valores
      - 6.2.2 Funciones de cadenas de caracteres
      - 6.2.3 *Funciones* condicionales
- Bibliografía

1. PostgreSQL

Durante este curso vamos a aprender SQL utilizando el sistema gestor de bases de datos PostgreSQL. Éste, también conocido como Postgres, es un sistema de base de datos de código abierto, robutos y que puede manejar grandes cantidades de datos.

Algunas razones para utilizarlo son:

- Es gratuito.
- Está disponible para Windows, Linux, MacOS, Unix, ...
- Su implementación de SQL sigue de cerca los estándares ANSI.
- Es ampliamente utilizado para análisis y minería de datos, por lo que es fácil encontrar ayuda a través de la comunidad de usuarios.
- Su extensión geoespacial, PostGIS, permite analizar datos geométricos y realizar funciones de mapeo.
- Se utiliza con frecuencia junto con aplicaciones web, sobre todo con lenguaje Python.
- Está disponible gratuítamente en cuentas *freemium* en la nube (como Heroku).

2. ¿Qué otras alternativas podríamos usar?

Por supuesto, también podríamos usar otro sistema de base de datos, como Oracle Database, Microsoft SQL Server o MySQL; la mayoría del código que aprenderemos durante las próximas unidades didáctcas se traducen fácilmente a cualquier implementación de SQL. Sin embargo, algunos ejemplos, especialmente más adelante en el libro, no lo hacen, y tendremos que esforzarnos en buscar soluciones equivalentes si queremos adaptarlas a otro sistema gestor. Cuando corresponda, señalaremos si un código de ejemplo sigue el estándar ANSI SQL y puede ser portado a otros sistemas o si es específico de PostgreSQL.

3. Instalación de PostgreSQL

Vamos a comenzar instalando la base de datos PostgreSQL y la herramienta gráfica pgAdmin, que es un software que facilita la administración y uso de la base de datos, importar y exportar datos, escribir consultas y todo tipo de tareas administrativas.

Una gran ventaja de trabajar con PostgreSQL es que es de código abierto, y podemos encontrar *binarios* para los sistemas operativos más conocidos.

> Para cualquier sistema operativo, para descargar el instalador de PostgreSQL podemos visitar la web https://www.postgresql.org/download/

3.1 Instalación en Windows

Para Windows, es muy útil usar el instalador provisto por la compañía *EntrepriseDB*, que ofrece soporte y servicios para los usuarios de PostgreSQL. El paquete contiene el sistema gestor de PostgreSQL con pgAdmin y otro componente llamado *Stack Builder*, propio de la compañía, que instala la extensión *espacial* PostGIS entre otras herramientas.

Para descargar el software, visitamos la web https://www.postgresql.org/download/, que nos lleva a este otro enlace para la descarga: https://www.enterprisedb.com/downloads/postgres-postgresql-downloads.

Downloads

PostgreSQL Core Distribution

The core of the PostgreSQL object-relational database management system is available in several source and binary formats.

Binary packages

Pre-built binary packages are available for a number of different operating systems:

- BSD
  - FreeBSD
  - OpenBSD
- Linux
  - Red Hat family Linux (including CentOS/Fedora/Scientific/Oracle variants)
  - Debian GNU/Linux and derivatives
  - Ubuntu Linux and derivatives
  - SUSE and openSUSE
  - Other Linux
- macOS
- Solaris
- Windows

Windows installers

Interactive installer by EDB

Download the installer certified by EDB for all supported PostgreSQL versions.

This installer includes the PostgreSQL server, pgAdmin; a graphical tool for managing and developing your databases, and StackBuilder; a package manager that can be used to download and install additional PostgreSQL tools and drivers. Stackbuilder includes management, integration, migration, replication, geospatial, connectors and other tools.

This installer can run in graphical or silent install modes.

The installer is designed to be a straightforward, fast way to get up and running with PostgreSQL on Windows.

Advanced users can also download a zip archive of the binaries, without the installer. This download is intended for users who wish to include PostgreSQL as part of another application installer.

Platform support

The installers are tested by EDB on the following platforms. They can generally be expected to run on other comparable versions:

PostgreSQL Version	64 Bit Windows Platforms	32 Bit Windows Platforms
12	2019, 2016, 2012 R2	
11	2019, 2016, 2012 R2	
10	2016, 2012 R2 & R1, 7, 8, 10	2008 R1, 7, 8, 10
9.6	2012 R2 & R1, 2008 R2, 7, 8, 10	2008 R1, 7, 8, 10
9.5	2012 R2 & R1, 2008 R2	2008 R1

> En el momento de redactar estos apuntes, la versión estable más actualizada de PostgreSQL es la 12.3

> En este vídeo podemos aprender los pasos a seguir en la instación de Windows (aunque es para una versión anterior, nos puede servir perfectamente): https://www.youtube.com/results?search_query=sql+postgresql+

> PENDIENTE DE TERMINAR

3.2 Instalación en Linux

> PENDIENTE DE TERMINAR

3.3 Instalación en Mac

Seguimos las indicaciones similares a las de Windows, pero descargamos en enlace para Mac.

> PENDIENTE DE TERMINAR

4. Introducción a SQL

4.1 Objetivos

SQL es el lenguaje fundamental de los SGBD relacionales. Se trata de uno de los lenguajes más utilizados de la historia de la informática y sigue siendo de aprendizaje casi obligatorio para cualquier profesional relacionado con la computación.

SQL es un **lenguaje declarativo**, lo que implica que se centra en definir **qué** se desea hacer, por encima de **cómo** hacerlo (que es la forma de trabajar de los lenguajes de programación de aplicaciones como C o Java). La razón de este matiz, es que los lenguajes declarativos se parecen más al lenguaje natural humano y parecen más apropiados para trabajar con bases de datos (especialmente con las relacionales).

Se trata de un lenguaje que intenta agrupar todas las funciones que se le pueden pedir a una base de datos, por lo que es el lenguaje utilizado tanto por administradores como por programadores o incluso usuarios avanzados.

Pretende cumplir la **quinta regla de Codd**, que dicta que *el lenguaje de la base de datos debe de ser capaz de realizar cualquier instrucción sobre la misma*. En sistemas gestores como PostgreSQL esta regla se cumple completamente: toda la gestión y administración del sistema de bases de datos se puede realizar utilizando solo lenguaje SQL.

4.2 Historia

El nacimiento del lenguaje SQL data de 1970 cuando **E. F. Codd** publica su libro: *"Un modelo de datos relacional para grandes bancos de datos compartidos"*. Ese libro dictaría las direcrices de las bases de datos relacionales. Apenas dos años después **IBM** (para quien trabajaba Codd) utiliza las directrices de Codd para crear el *Standard English Query Language* (**Lenguaje Estándar Inglés para Consultas**) al que se le llamó **SEQUEL**. Más adelante se le asignaron las siglas **SQL** (*Standard Query Language*, lenguaje estándar de consulta) aunque en inglés se siguen pronunciando *secuel*. En español se pronuncia *esecuele*.

En 1979 Oracle presenta la primera implementación comercial del lenguaje. Poco después se convertía en un estándar en el mundo de las bases de datos avalado por los organismos ISO y ANSI. En el año 1986 se toma como lenguaje estándar por **ANSI** de los SGBD relacionales. Un año después lo adopta **ISO**, lo que convierte a SQL en estándar mundial como lenguaje de bases de datos relacionales.

En 1989 aparece el estándar ISO (y ANSI) llamado SQL89 o **SQL1**. En 1992 aparece la nueva versión estándar de SQL (a día de hoy sigue siendo la más conocida) llamada **SQL92**. En 1999 se aprueba un nuevo SQL estándar que incorpora mejoras que incluyen triggers, procedimientos, funciones,... y otras características de las bases de datos objeto-relacionales; dicho estándar se conoce como SQL99 o **SQL2000**.

Tras ese estándar, se publicaron nuevos estándares en los años 2003, 2006, 2008, 2011 y 2016. Por lo tanto, el último estándar es el del año 2016 (SQL2016). Pero la mayoría de las mejoras añadidas en los últimos estándares se refieren al uso de otros lenguajes (especialmente los referentes a XML y JSON) con el propio SQL, a la incorporación de elementos orientados a objetos o a la mejora en el tratamiento de fechas.

> PostgreSQL es compatible con la mayoría de las características principales de SQL: 2016. De las 179 características obligatorias requeridas para la plena conformidad del estandar, PostgreSQL cumple al menos con 160. Además, hay una larga lista de características opcionales compatibles. Vale la pena

> señalar que en el momento de la escritura, ninguna versión actual de ningún sistema gestor de bases de datos afirma la plena conformidad con SQL:2016.

4.3 Funcionamiento

4.3.1 Algunos conceptos para usar SQL con Postgresql

- **Usuario**: en PostgreSQL un *usuario*, *grupo* y *rol* son lo mismo, con la única diferencia que los usuarios tienen por defecto privilegio de conexión. Las sentencias DDL de creación `CREATE USER` y `CREATE GROUP` son alias de `CREATE ROLE`.

- **Privilegio**: se trata del permiso de un usuario para realizar una determinada acción. Esto permite que el sistema gestor controle qué acciones pueden realizar los usuarios.
- **Rol**: un rol es un conjunto de privilegios, que se agrupan para facilitar su gestión.
- **Base de datos**: una base de datos será un conjunto de tablas, funciones y otros objetos.

4.3.2 Usuarios que intervienen

Por defecto, al instalar PostgreSQL se crea un nuevo usuario llamado `postgres`. Durante la instalación hemos establecido la contraseña de este usuario. Dicho usuario es el adminnistrador de la base de datos.

Este usuario lo deberíamos utilizar solo para tareas administrativas, tales como crear otros usuarios que tengan unos privilegios más concretos.

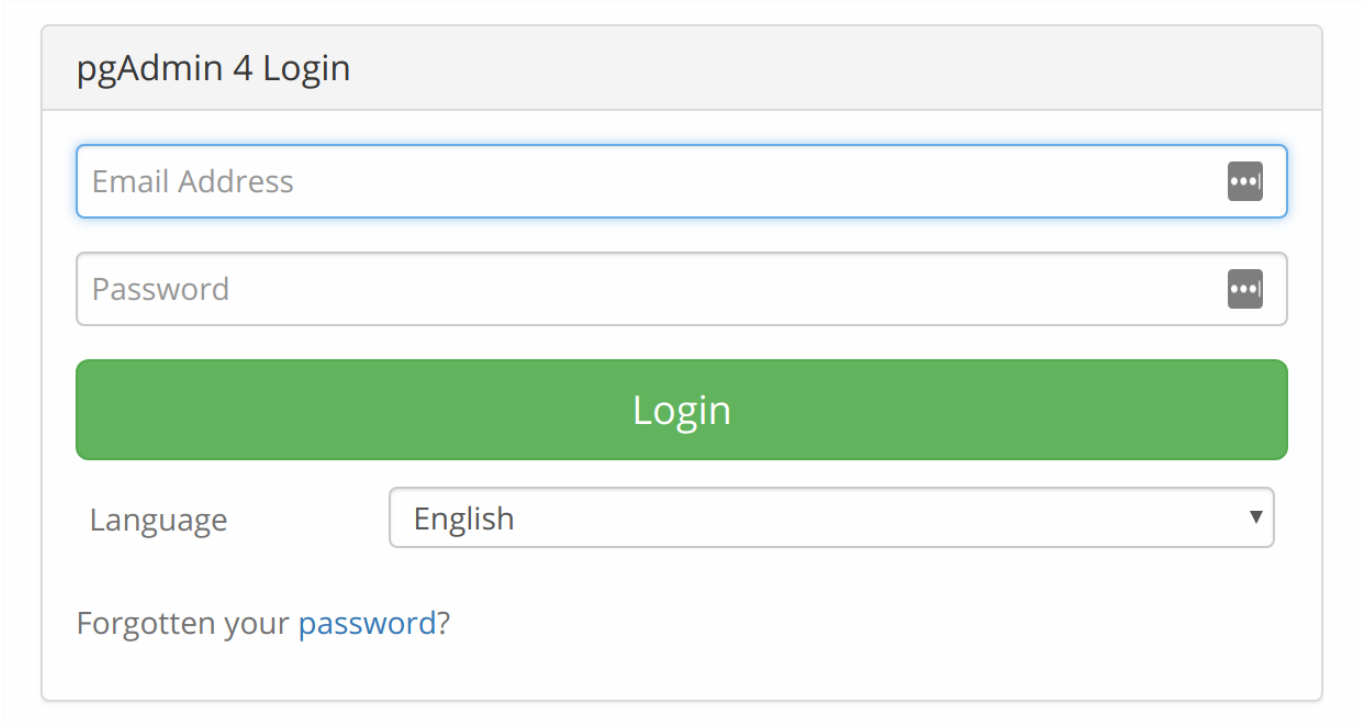
> En el servidor de clase, no tendrás acceso a este usuario. Tendrás un usuario personalizado para poder conectarte a las diferentes bases de datos durante todo el curso.

> En la instalación que hagas en casa, sí que podrás utilizarlo, aunque es recomendable utilizarlo para crear otros usuarios que utilices con privilegios más específicos.

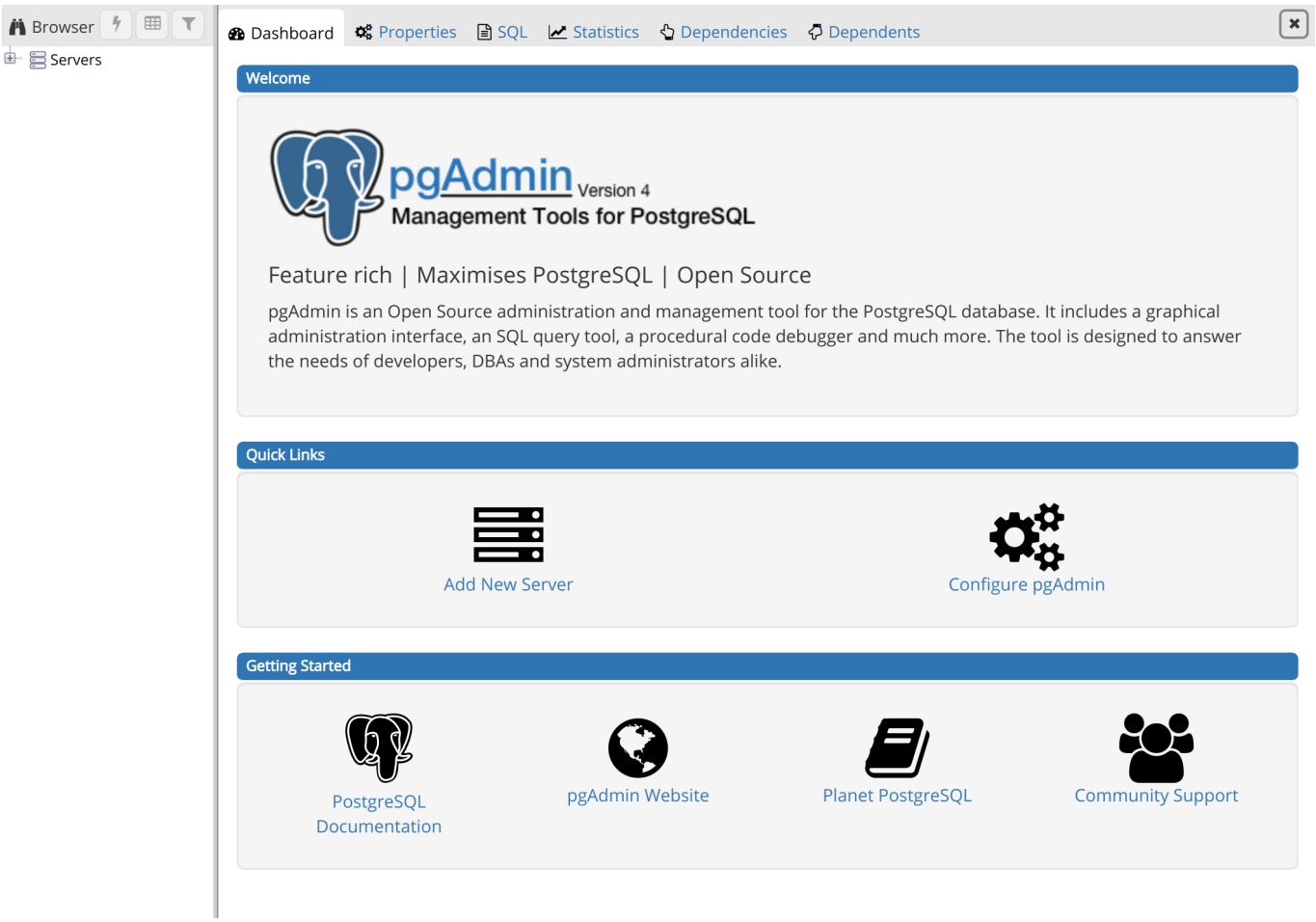
4.3.3. Crear nueva conexión

Depende del programa cliente que vayamos a utilizar. Si usamos pgAdmin 4, podemos seguir unas instrucciones parecidas a estas para hacerlo.

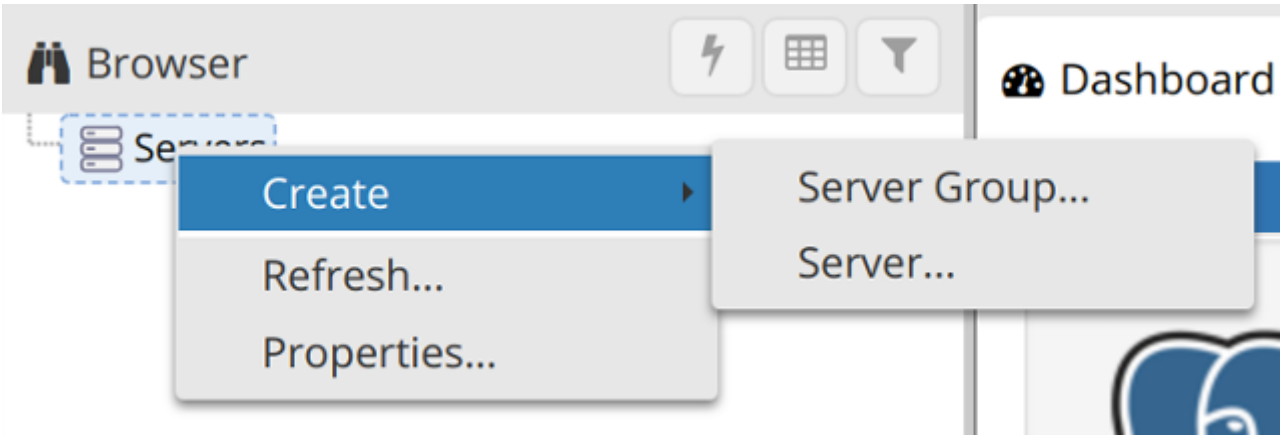
Cuando ejecutamos pgAdmin 4 por primera vez aparece una pantalla como esta:



Introducimos las credenciales de acceso y entonces aparece la pantalla de bienvenida de pgAdmin 4:

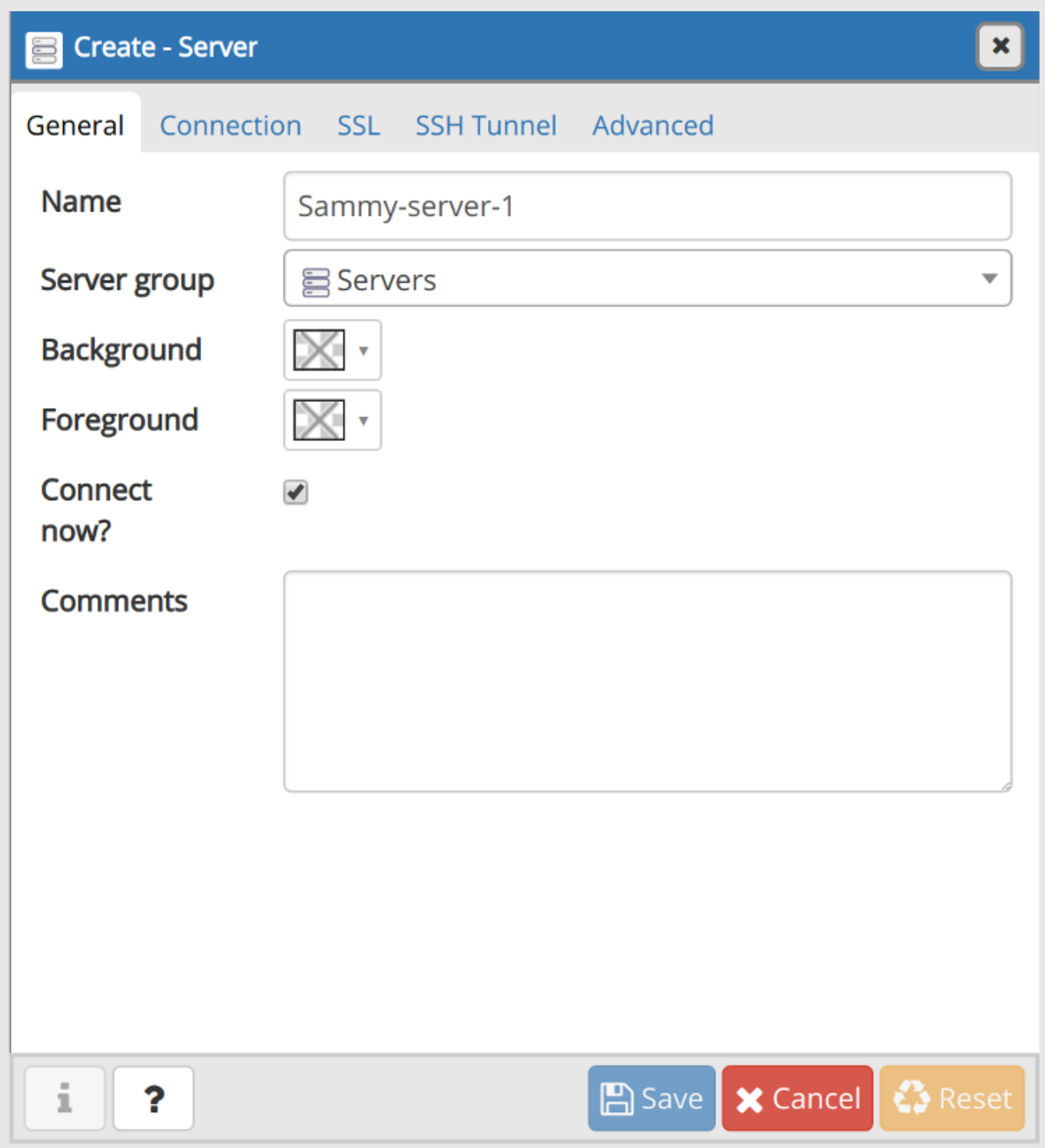


A continuación, podemos pulsar en *Browser* > *Server*. Aparece un menú contextual y podemos pulsar en *Create* > *Server*.



Esto hará que en el navegador aparezca una ventana en la que se puede introducir la información sobre el servidor, nuestro usuario (rol) y la base de datos.

En la pestaña General, introducimos el nombre para este servidor. Puede ser cualquier cosa que deseemos, pero puede resultarle útil un nombre descriptivo.

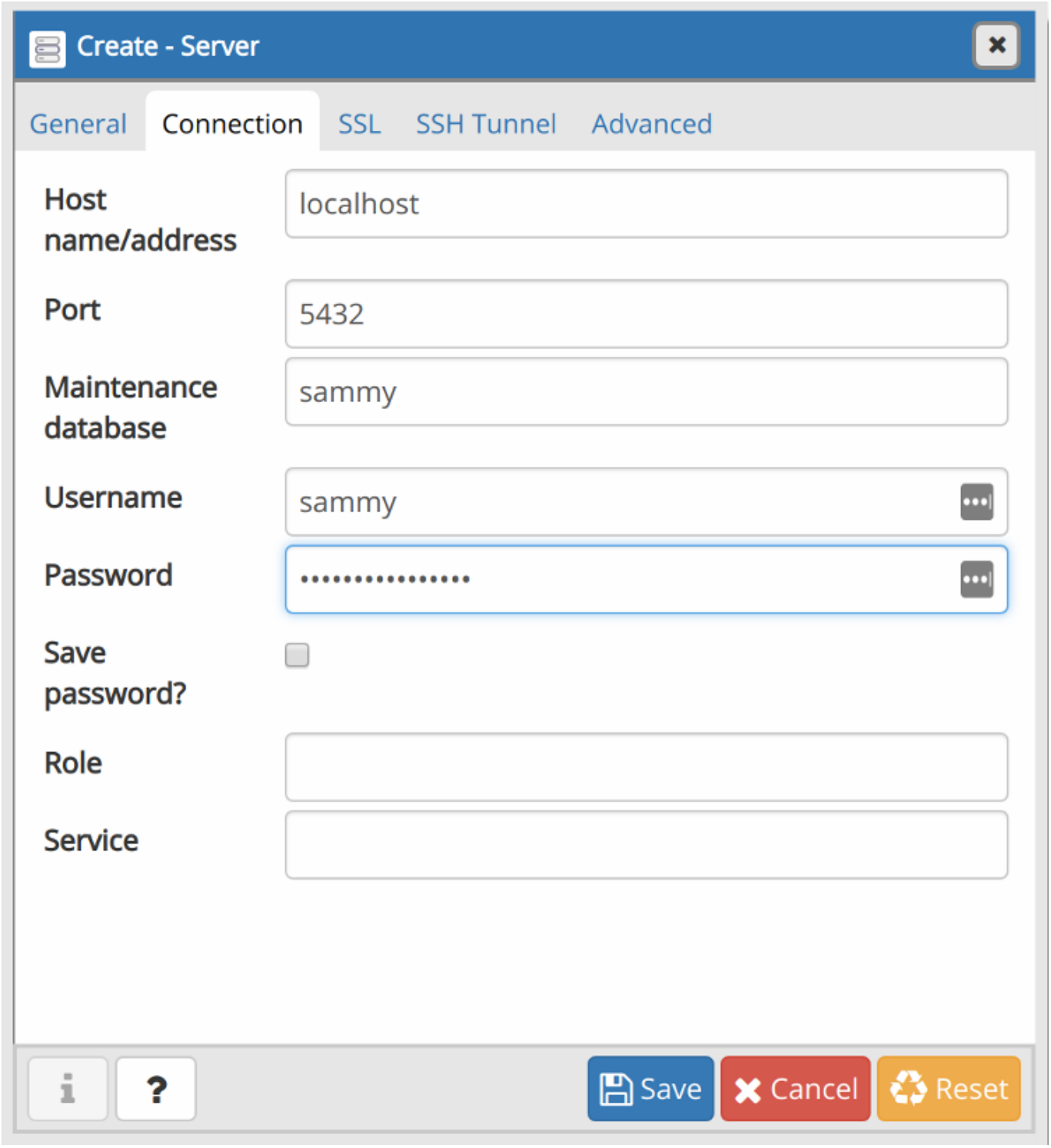


A continuación, hacemos clic en la pestaña *Connection*. En el campo de *Host name/address*, escribimos `localhost`. El valor preestablecido de *Port* debería ser `5432`, que funcionará para esta configuración, ya que es el puerto que PostgreSQL utiliza de forma predeterminada.

En el campo *Maintenance database*, puedes escribir el nombre de la base de datos a la que quieres conectarte. Ten en cuenta que esta base de datos ya debe estar creada en el servidor.

> Este paso no es estrictamente necesario; el siguiente sí.

Luego, escribe el nombre de usuario y la contraseña de PostgreSQL en los campos *Username* y *Password*, respectivamente.



Los campos vacíos de las demás pestañas son opcionales y solo es necesario completarlos si se tiene en mente una configuración específica en la que se necesiten. Haz clic en el botón *Save*. La base de datos aparecerá en *Servers*, en el menú *Browser*.

Con esto, hemos establecido de forma correcta la conexión entre pgAdmin4 y nuestra base de datos de PostgreSQL. Desde el panel de pgAdmin puede realizar prácticamente cualquier acción de las que ejecutaríamos desde la línea de comandos de PostgreSQL.

> En el vídeo utilizado en el apartado de instalación de PostgreSQL también podemos ver cómo hacer este paso.

5. Consultas de datos con SQL. DQL.

DQL es la abreviatura del *Data Query Language* (lenguaje de consulta de datos) de SQL. El único comando que pertenece a este lenguaje es el versátil comando `SELECT` Este comando permite fundamentalmente:

- Obtener datos de ciertas columnas de una tabla (proyección).
- Obtener registros (filas) de una tabla de acuerdo con ciertos criterios (selección).
- Mezclar datos de tablas diferentes (asociación, join).
- Realizar cálculos sobre los datos.
- Agrupar datos.

5.1 Sintaxis sencilla del comando `SELECT`

La siguiente instrucción `SELECT` recupera todas las columnas de cada una de las filas de una tabla llamada `mitabla`.

```
SELECT *
FROM mitabla;
```

Esta única línea de código muestra la forma más básica de una consulta SQL. El asterisco que sigue a la palabra clave `SELECT` es un comodín. Un comodín es como unsuplente de un valor: no representa nada en particular y en su lugar representa todo lo que ese valor podría ser. Aquí es el atajo "seleccionar todas las columnas". Si hubiera dado un nombre de columna en lugar del comodín, este comando seleccionaría los valores en esa columna. La palabra clave `FROM` indica sobre que tabla vamos a hacer la consulta. El punto y coma después del nombre de la tabla le dice a PostgreSQL que es el final de la declaración de consulta.

5.2 Consultar un subconjunto de columnas

El asterisco es cómodo si se quieren obtener todas las columnas de una tabla. Pero a menudo es práctico limitar las columnas de la consulta, sobre todo en grandes bases de datos donde una tabla puede tener centenares de atributos.

Para limitar las columnas a obtener como resultado puedes enumerarlas, separadas por comas, entre las palabras claves `SELECT` y `FROM`.

```
SELECT una_columna, otra_columna, yotra_columna FROM latabla;
```

Con estas sintaxis recuperaremos todas las filas de esas tres columnas.

5.2.1 Uso de alias

En ocasiones (lo comprobaremos empíricamente más adelante) puede interesarnos renombrar alguna columna que obtenemos como resultado de una consulta.

> ¡OJO! Esto no modifica en ningún caso el nombre de la columna en la tabla. Sólo estamos dando otro nombre al resultado de una consulta.

A la operación de dar un nombre diferente a una columna del resultado de una consulta se le llama dar un alias. La sintaxis sería así:

```sql
SELECT nombrecolumna AS "otro_nombre"
FROM latabla;
```

Podemos dar un alias a más de una columna en la misma consulta:

```sql
SELECT col1 AS "Columna 1", col2 AS "Columna 2"
FROM latabla;
```

5.3 Uso de `DISTINCT` para obtener valores únicos

En una tabla no es inusula encontrar que una columna tenga valores duplicados en diferentes filas. Por ejemplo, en una tabla para manejar los `alumnos` de un centro educativo, el año de nacimiento coincidirá en muchos de ellos.

Si queremos obtener los valores únicos de esa columna (o de un conjunto de columnas), podemos utilizar la palabra clave `DISTINCT`, que elimina los duplicados y solamente muestra los valores únicos.

> `DISTINCT` no se suele utilizar con el caracter comodín asterisco. De hacerlo, significaría que estamos buscando aquellas filas que sean únicas, diferentes a todas las demás. Por definición, en el modelo relacional no podemos tener filas (o tuplas) duplicadas, con lo cual, usar `SELECT DISTINCT *` no nos aportará nada diferente a usar solamente `SELECT *`.

```sql
SELECT DISTINCT col1, col2
FROM latabla;
```

> La palabra clave `DISTINCT` también funciona en más de una columna a la vez. Si agregamos una columna, la consulta devuelve cada par único de valores.

5.4 Ordenación de los datos con `ORDER BY`.

Los datos pueden tener más sentido y pueden revelar patrones más fácilmente, cuando están organizado en orden en lugar de mezclarse al azar. En SQL, ordenamos los resultados de una consulta utilizando una cláusula que contiene las palabras clave `ORDER BY` seguidas del nombre de la columna o columnas para ordenar, separadas por comas. **La aplicación de esta cláusula no cambia la tabla original, solo el resultado dela consulta**.

```sql
SELECT *
FROM latabla
ORDER BY col1, col2;
```

Por defecto, SQL ordena los datos de forma ascendente, pero este comportamiento se puede modificar con la palabra clave `DESC`.

```
SELECT *
FROM latabla
ORDER BY col1, col2 DESC;
```

En el ejemplo anterior, los datos se ordenarían primero por la `col1` de forma ascendente, y luego por la `col2` de forma descendente.

> También existe la palabra `ASC` para indicar el orden ascendente, pero como podemos comprobar su uso no es obligatorio.

5.5 Uso de condiciones

Se pueden realizar consultas que restrinjan los datos de salida de las tablas. Para ello se utiliza la cláusula `WHERE`. Esta cláusula permite colocar una condición que han de cumplir todos los registros, los que no la cumplan no aparecen en el resultado.

Por ejemplo:

```
SELECT *
FROM productos
WHERE precio < 10;
```

Existen varios operadores de comparación que podemos utilizar en las cláusulas `WHERE`:

Operador	Función	Ejemplo
`=`	Igual a	`WHERE nombre = 'María'`
`<>` o `!=`	Distinto a	`WHERE nombre != 'María'`
`>`	Mayor que	`WHERE precio > 5`
`<`	Menor que	`WHERE precio < 10`
`>=`	Mayor o igual que	`WHERE altura >= 160`
`<=`	Menor o igual que	`WHERE altura <= 180`
`BETWEEN`	Entre (inclusivo)	`WHERE altura BETWEEN 160 and 180`
`IN`	Buscar un valor en un conjunto	`WHERE precio IN (13, 15, 23)`
`LIKE`	Coincide con un patrón	`WHERE apellido LIKE 'San%'`
`ILIKE`	Coincide con un patrón (no sensible a mayúsculas/minúsculas)	`WHERE apellido ILIKE 'san%'`
`NOT`	Niega una condición	`WHERE apellido NOT ILIKE 'san%'`.

5.5.1 Uso de `LIKE` y `ILIKE`

Los operadores de comparación son bastante sencillos, pero `LIKE` e `ILIKE` merecen una explicación adicional. En primer lugar, ambos permiten buscar patrones en cadenas mediante el uso de dos caracteres especiales:

- **Signo de porcentaje (%)**: Un comodín que coincide con cero o más caracteres cualesquiera.
- **Subrayado (_)**: Un comodín que coincide con uno y sólo un caracter.

Por ejemplo, si está tratando de encontrar la palabra `programador`, los siguientes patrones a usar con `LIKE` lo igualarán:

```
LIKE 'p%'
LIKE '%gram%'
LIKE '_rogramador'
LIKE 'progra__dor'
```

> El operador `LIKE` es parte del estándar ANSI SQL, y es *case sensitive* (diferencia entre mayúsculas y minúsculas). EL operador `ILIKE` es propio de PostgreSQL y es s *case insensitive* (no diferencia entre mayúsculas y minúsculas)

5.5.2 Combinación de condiciones con `AND` y `OR`.

La sentencia `WHERE` debe estar formada siempre por esta palabra reservada y **una sola condición**. Ahora bien, esta condición no tiene que ser simple. De hecho, comprobaremos que parte de la potencia de SQL reside en que se pueden manejar condiciones complejas.

Podemos construir condiciones complejas a partir de la **conjunción** o **disyunción** de condiciones más sencillas, utilizando los operadores lógicos `AND` y `OR`.

A	B	A AND B	A OR B
Falso	Falso	Falso	Falso
Falso	Verdadero	Falso	Verdadero
Verdadero	Falso	Falso	Verdadero
Verdadero	Verdadero	Verdadero	Verdadero

> Con el operador `OR` para que el resultado sea verdadero, basta con que una de las dos condiciones de entrada lo sean; con el operador `AND` para que el resultado sea verdadero, las dos condiciones de entrada deben serlo.

Algunos ejemplos de uso de estos operadores podrían ser los siguientes:

```
SELECT *
FROM productos
WHERE precio < 10 OR precio > 20;
```

```
SELECT *
FROM profesores
WHERE escuela = 'Salesianos San Pedro'
AND (edad < 30 or edad > 50);
```

> Como podemos comprobar, podemos hacer uso de los paréntesis para agrupar condiciones y forzar así la precedencia.

5.5.3 Uso de valores nulos

En SQL para valorar los nulos, es frecuente cometer este error:

```
SELECT nombre,apellidos FROM personas
WHERE telefono=NULL
```

Esa expresión no es correcta. No debemos usar los operadores de comparación normales con valores nulos. La consulta anterior no muestra a las personas sin teléfono (que es lo que pretendemos).

Lo correcto es usar el operador destina a comprobar si un determinado dato es nulo:

```
SELECT nombre,apellidos FROM personas
WHERE telefono IS NULL
```

Esa instrucción selecciona a la gente que no tiene teléfono.

Existe también la expresión contraria: `IS NOT NULL` que devuelve verdadero en el caso contrario, ante cualquier valor distinto de nulo.

5.5.4 Precedencia de operadores

A veces las expresiones que se producen en los SELECT son muy extensas y es difícil saber que parte de la expresión se evalúa primero. Por ello es necesario conocer la tabla de precedencia que indica qué operadores tienen prioridad entre sí. Los que están al nivel 1 tienen la máxima prioridad.

La tabla completa es la siguiente (con precedencia de mayor a menor):

Operador	Descripción
.	Separador de nombre de tabla y columna
+, -	Positivo unario, negativo unario
^	Exponenciación
*, /, %	Multiplicación, divisón, módulo
+, -	Suma, resta

Operador	Descripción
`BETWEEN`, `IN`, `LIKE`, `ILIKE`	comparador de rango, pertenencia a un conunto, comparación con cadenas
`<`, `>`, `=`, `<=`, `>=`, `!=`, `<>`	menor, mayor, igual, menor o igual, mayor o igual, distinto
`IS NULL`, `IS NOT NULL`	Es nulo, es diferente de nulo
`NOT`	Negación
`AND`	Conjunción lógica
`OR`	Disyunción lógica.

Las reglas de prioridad se pueden alterar mediante paréntesis.

5.6 Concatenación de textos

El operador de concatenar texto permite unir dos textos. Normalmente se usa para juntar resultados de diferentes expresiones en una miasma columna de una tabla. Todas las bases de datos incluyen algún operador para encadenar textos. En Postgresql el operador es la doble barra vertical `||`.

```
SELECT tipo, modelo, tipo || '-' || modelo "Clave Pieza"
FROM piezas;
```

El resulado podría ser algo así:

Tipo	Modelo	Clave Pieza
AR	6	AR-6
AR	7	AR-7
BI	10	BI-10
...	...	...

En la mayoría de bases de datos, la función `CONCAT` (la estudiaremos más adelante) realiza la misma función.

6 Realización de cálculos en SQL

Los operadores + (suma), - (resta), * (multiplicación) y / (división), se pueden utilizar para hacer cálculos en las consultas. Cuando se utilizan como expresión en una consulta SELECT, no modifican los datos originales sino que como resultado de la vista generada por SELECT, aparece un nueva columna. Ejemplo:

```
SELECT nombre, precio,precio*1.21 FROM productos;
```

Esta consulta obtiene tres columnas; la tercera muestra el resultado de la operación. Al no indicar nombre alguno (a través de un alias), se toma la propia expresión como cabecera de esa columna.

> En casos como este, es bueno indicar un alias para las columnas.

En la siguiente tabla podemos consultar los diferentes operadores disponibles en Postgresql

Operador	Descripción
`+`	Adición
`-`	Sustracción
`*`	Multiplicación
`/`	División (solo cociente, sin resto)
`%`	Módulo (resto de la división entera)
`^`	Exponenciación
`|/`	Raíz cuadrada
`||/`	Raíz cúbica
`!`	Factorial

> **Operaciones y tipos de datos**
>
> En los cálculos (suma, resta, multiplicación y división) el tipo de dato devuelto sigue el siguiente patrón:
>
> - Una operación entre dos enteros devuelve un entero
> - Un valor de tipo `numeric` a cada lado del operador devuelve un valor `numeric`
> - El uso de un número de coma flotante devuelve un valor en coma flotante de precisión doble.
> - Las funciones de exponenciación, raiz y factorial se comportan de forma diferente.

> Como aprenderemos más adelante, mediante la función `CAST` podremos transformar un valor de un tipo de dato a otro.

Algunos ejemplos de consultas podrían ser los siguientes:

Operación	Ejemplo	Resultado
Suma	`SELECT 2 + 2;`	`4`
Resta	`SELECT 9 - 1;`	`8`
Multiplicación	`SELECT 3 * 4;`	`12`
División (entera)	`SELECT 11 / 6;`	`1`
Resto	`SELECT 11 % 6;`	`5`
División (decimales)	`SELECT 11.0 / 6;`	`1.83333333`
Exponenciación	`SELECT 3 ^ 4;`	`81`

Operación	Ejemplo	Resultado
Raíz cuadrada	`SELECT |/ 10;` `SELECT SQRT(10);`	`3.16227766`
Raíz cúbica	`SELECT ||/ 10;`	`2.15443469`
Factorial	`SELECT 4!;`	`24`

6.1 Cálculo de porcentajes

En ocasiones, nos será interesante trabajar con porcentajes para, por ejemplo:

- Calcular qué porcentaje representa un valor sobre el total (por ejemplo, población de mujeres sobre el total)
- Incrementar o decrementar un valor en un porcentaje (por ejemplo, un descuento)

Un ejemplo podría ser este:

```
SELECT provincia, anio, ( mujeres::numeric / (hombres + mujeres)) * 100 as "% de
mujeres", hombres + mujeres as "total"
from demografia_basica;
```

> La expresión `mujeres::numeric` sirve para transformar el valor entero de la columna `mujeres` en un valor decimal. De esta forma, la división que se realiza no es la división entera, sino con decimales. El operador `::` sirve para transformar una columna en otro tipo de dato; también podríamos haber utilizado la función `CAST(value as type)`

Si queremos **incrementar un valor en un determinado porcentaje**, usaríamos una fórmula como esta

$valor + (valor * porcentaje) / 100$

Por ejemplo, si a 50 le queremos hacer un incremento del 21%, al aplicar la fórmula nos resultaría

$50 + (50 * 21) / 100 = 50 + (1050 / 100) = 50 + 10.5 = 60.5$

Si aplicamos algo de matemáticas tenemos algunas de estas fórmulas intermedias

$valor + valor * porcentaje/100 = valor * 1 + valor * porcentaje/100$

Aquí, podemos sacar factor común, y nos quedaríamos con esta fórmula:

$valor * (1 + porcentaje/100)$

Esto simplifica el cálculo, la que incrementar un valor en un 21% sería equivalente a multiplicar dicho valor por 1.21.

```
SELECT nombre, precio,precio*1.21 FROM productos;
```

Si en lugar de incrementar, queremos **decrementar**, la fórmula sería:

$valor * (1 - porcentaje/100)$

Por ejemplo, si a un precio le queremos aplicar un descuento de, pongamos, el 20%, el cálculo podría ser así:

$valor * (1 - porcentaje/100) = valor * (1 - 0.2) = valor * 0.8$

```
SELECT nombre, precio, precio * 0.80 FROM productos;
```

6.2 Funciones

En informática, podemos decir que:

> una **función** es un conjunto de instrucciones, *que puede recibir cero, uno o más valores de entrada*, y que produce, como *resultado, un valor de salida*.

El concepto de función en informática es muy similar al de las matemáticas.

El estándar ANSI SQL, y más en particular Postgresql nos ofrecen un conjunto de funciones predefinidas que nos ayudarán a realizar muchos cálculos y operaciones. A continuación, vamos a conocer un conjunto de ellas.

Las funciones pueden ser utilizadas tanto en la cláusula `SELECT` como en la cláusula `WHERE` de una sentencia.

6.2.1 Funciones numéricas

Son funciones que realizan operaciones con expresiones numéricas.

> Al describir la sintaxis, en ocasiones se hacen uso de los **corchetes** `[]`. Estos no se deben usar explícitamente; tan solo indican que la expresión que contienen es **optativa**.

6.2.1.1 Funciones que trabajan fila a fila

Estas funciones devuelven un valor para cada fila sobre la que trabajan.

> Entenderás mejor el concepto cuando veamos el siguiente conjunto de funciones en el próximo apartado.

Función	Descripción	Ejemplo
`abs(x)`	Valor absoluto de un número	`SELECT abs(-20);` devuelve `20`
`sqrt(x)`	Raíz cuadrada de un número	`SELECT sqrt(144);` devuelve `12`
`cbrt(x)`	Raíz cúbica de un número	`SELECT cbrt(9);` devuelve `9`
`ceiling(x)`	Redondeo hacia arriba	`SELECT ceiling(12.34);` devuelve `13`
`floor(12.34)`	Redondeo hacia abajo	`SELECT floor(12.34);` devuelve `12`
`power(x,y)`	Devuelve el valor `x` elevado a `y`	`SELECT power(2,3);` devuelve `8`

Función	Descripción	Ejemplo
`round(x [,n])`	Redondea el valor `x` con `n` decimales. El valor de `n` es optativo.	`SELECT round(10.4);` devuelve 10
`trunc(x [,n])`	Trunca el número `x` con `n` decimales. El valor de `n` es optativo.	`SELECT round(10.6);` devuelve 10
`mod(x,y)`	Devuelve el resto de dividir `x` entre `y`	`SELECT mod(11,2);` devuelve `1`
`pi()`	Devuelve el valor de *Pi*	`SELECT pi();` devuelve `3.14159265358979`
`random()`	Valor aleatorio entre 0 y 1 (sin estar éstos incluídos)	`SELECT random();` podría devolver `0.8134891239417`
`sin(x)`, `cos(x)`, `tan(x)`, ...	Funciones trigonométricas	`SELECT sin(0);` devuelve `0`, `SELECT cos(0);` devuelve `1`, ...

6.2.1.2 Funciones que agregan valores

Son funciones que agregan o resumen todos los valores que reciben en uno solo.

> Al describir la sintaxis, en ocasiones se hacen uso de la **barra vertical** `|`. Ésta no se debe usar explícitamente; tan sólo indica que en la expresión debemos escoger **una opción** entre lo que hay a la izquierda de la barra y lo que hay a la derecha.

Función	Descripción	Ejemplo
`avg(col)`	Devuelve el valor medio de todos los valores proporcionados	`SELECT avg(mujeres) FROM demografia_basica WHERE anio=2019;` Devuelve el número medio de habitantes que son mujeres por provincia en el año 2019.
`sum(col)`	Devuelve la suma de todos los valores proporcionados	`SELECT sum(hombres + mujeres) FROM demografia_basica WHERE anio=2019;` Devuelve el número total de habitantes de España en 2019.
`count(* | [DISTINCT] col)`		
`max(col)`, `min(col)`	Devuelve los valores máximo o mínimo de una columna.	`SELECT min(hombres) FROM demografia_basica WHERE anio=2019;` Devuelve el menor valor de población de hombres de una provincia en 2019.

Nos quedan otras dos funciones que, para explicar mejor, las vamos a sacar de la tabla.

> **ANTES, ALGO DE ESTADÍSTICA**
> SQL nos ofrece la posibilidad de hacer cálculos estadísticos de una forma sencilla. Ya hemos visto que podemos calcular la media aritmética a partir de la función `avg(...)`. Ésta forma parte de los llamados *parámetros estadísticos de centralización*. Otro de estos parámetros sería la **mediana**. La mediana es el elemento de un conjunto de datos *ordenados* $x_1, x_2, ..., x_n$ **que deja a la izquierda y la derecha la mitad (50%) de los valores**.
> Por ejemplo, para el conjunto $1, 2, 3, 4, 5$, la mediana sería el 3 (en este caso, por ejemplo, coincide con la media aritmética). O para el conjunto $1, 1, 1, 1, 5$, la mediana sería 1 (y la media aritmética sería 1.8).
> De forma más genérica encontramos el concepto de **percentil**. El percentil es una medida de posición usada en estadística que indica, *una vez ordenados los datos de menor a mayor*, **el valor de la variable por debajo del cual se encuentra un porcentaje dado de los datos**. Por ejemplo, el percentil 20º es el valor bajo el cual se encuentran el 20% de los datos. De esta forma, *la mediana sería el percentil 50º*.
> Si quieres saber algo más sobre estadística, te recomiendo que visites estas direcciones: https://www.universoformulas.com/estadistica/descriptiva/, https://es.wikipedia.org/wiki/Mediana_(estad%C3%ADstica), https://es.wikipedia.org/wiki/Percentil.

Y ahora sí, sigamos con SQL. PostgreSQL nos ofrece dos funciones que tienen una sintaxis algo diferente a las anteriores, `percentile_cont(n)` y `percentile_disc(n)`. Ambas son parte del estándar ANSI SQL.

- La función `percentile_disc(n)` calcula percentiles *discretos*. Es decir, devolverá un valor del conjunto de entrada más cercano al percentil solicitado.
- La función `percentile_cont(n)` calcula percentiles *continuos*. Es decir, devolverá un valor *interpolado* entre varios valores según la distribución. Puede pensar que esto es más preciso, pero puede devolver un valor fraccionario entre los dos valores de la entrada.

> ¿Qué significa *interpolado*? La forma más sencilla de ilustrar la diferencia es considerar una lista de 2 números: [1, 2]. `percentile_disc(0.5)` (la mediana disceta) de esta lista será 1, el valor más cercano al centro que está en la lista. Por otro lado, `percentile_cont(0.5)` devolverá 1.5, el promedio de los dos números, ya que no hay ningún elemento individual en la lista que represente la mediana.

La sintaxis de ambas funciones sería así

```
select percentile_disc(n) within group (order by col1) [over (partition by col2)]
from t;
```

Es decir:

- Como argumento de la función `percentile_disc` o `percentile_cont` proporcionamos un número decimal (entre 0 y 1).
- Después, indicamos las palabras `within group` para indicar el conjunto de datos sobre el que realizar el cálculo.
- Como los percentiles necesitan que los datos estén ordenados, indicamos entre paréntesis el orden sobre el que vamos a realizar el cálculo a través de `order by` y el nombre de una columna.

Supongamos que tenemos una tabla llamada `things` con una columna llamada `value`:

```
select percentile_disc(0.5) within group (order by things.value)
from things
```

La anterior sentencia nos calcularía la mediana de la columna `value`.

> Más adelante, podremos hacer un uso más avanzado de estas dos funciones.

> La lista completa de funciones de agregación la puedes encontrar aquí https://www.postgresql.org/docs/current/functions-aggregate.html

6.2.2 Funciones de cadenas de caracteres

Función	Tipo de retorno	Descripción	Ejemplo
`string || no-string`	`text`	Concatenación de texto con una entrada no textual	`'Valor: ' || 42` devuelve `Valor: 42`.
`concat(string, string, ...)`	`text`	Concatenación de texto	`concat('abcde', 2, NULL, 22)` devuelve `abcde222`.
`concat_ws(sep, string, string, ...)`	`text`	Concatenación de texto con separador	`concat_ws(',', 'abcde', 2, NULL, 22)` devuelve `abcde,2,22`.
`length(string)`, `char_length(string)`	`int`	Número de caracteres en la cadena	`char_length('jose')` devuelve `4`
`lower(string)`	`text`	Convierte la cadena a minúsculas	`lower('HOLA')` devuelve `hola`
`upper(string)`	`text`	Convierte la cadena a mayúsculas	`upper('hola')` devuelve `HOLA`
`initcap(string)`	`text`	Convierte la primera letra de cada palabra en mayúsculas, y el resto en minúsculas	`initcap('hola TOMÁS')` devuelve `Hola Tomás`.
`left(string, n)`	`text`	Devuelve los `n` primeros caracteres de la cadena	`left('abcde', 2)` devuelve `ab`.
`right(string, n)`	`text`	Devuelve los `n` primeros caracteres de la cadena, contando desde la derecha	`right('abcde', 2)` devuelve `de`.
`replace(string, from, to)`	`text`	Reemplaza todas las ocurrencias en `string` de la cadena `from` por la cadena `to`	`replace('abcdefabcdef', 'cd', 'XX')` devuelve `abXXefabXXef`

Función	Tipo de retorno	Descripción	Ejemplo
`overlay(string placing string from int [for int])`	`text`	Reemplaza una subcadena en una posición determinada para un número determinado de caracteres	`overlay('Txxxás' placing 'om' from 2 for 3)` devuelve `Tomás`.
`translate(string text, from text, to text)`	`text`	Cualquier carácter de la cadena que coincida con un carácter de `from` se reemplaza por el correspondiente carácter de `to`. Si `from` es más largo que `to`, se eliminan las apariciones de los caracteres adicionales.	`translate('12345', '143', 'ax')` devuelve `a2x5`.
`ltrim(string, characters)`	`text`	Elimina la cadena más larga que contiene solo caracteres de `characters` (un espacio por defecto) desde el principio de la cadena	`ltrim('zzzytest', 'xyz')` devuelve `test`.
`rtrim(string, characters)`	`text`	Elimina la cadena más larga que contiene solo caracteres de `characters` (un espacio por defecto) desde el final de la cadena	`rtrim('testxxzx', 'xyz')` devuelve `test`.
`trim`			
`repeat(string, n)`	`text`	Repite la cadena `string n` veces	`repeat('Pg', 4)` devuelve `PgPgPgPg`
`reverse(str)`	`text`	Devuelve la cadena invertida	`reverse('abcde')` devuelve `edcba`.
`substr(string, from [, count])` `substring(string from from for count))`	`text`	Extrae una subcadena	`substr('alfabeto', 3, 2)` devuelve `fa`.
`strpos(string, substring)` `position(substring in string)`	`int`	Devuleve la posición de una subcadena dentro de una cadena	`str_pos('alto','lt')` devuelve `2`.

Función	Tipo de retorno	Descripción	Ejemplo
`starts_with(string, prefijo)`	`bool`	Devuelve verdadero si la cadena `string` comienza con la cadena `prefijo`.	`starts_with('alfabeto', 'alfa')` devuelve `true`.
`split_part(string text, delimiter text, field int)`	`text`	Divide una cadena por un delimitador y devuelve el campo identificado mediante una posición	`split_part('abc~@~def~@~ghi', '~@~', 2)` devuelve `def`.

> La lista completa de funciones de cadenas de caracteres la puedes encontrar aquí https://www.postgresql.org/docs/current/functions-string.html

6.2.3 *Funciones* condicionales

Algunas de las que vamos a describir a continuación no son funciones ordinarias, pero para nosotros su uso será idéntico, por lo que vamos a plantearlas de la misma forma.

- `CASE`: es una expresión similar a las sentencias `if/else` o `switch` de un lenguaje de programación cualquiera.

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

Cada condición es una expresión que devuelve un resultado booleano. Si el resultado de la condición es verdadero, el valor de la expresión CASE es el resultado que sigue a la condición y el resto de la expresión CASE no se procesa. Si el resultado de la condición no es verdadero, las cláusulas WHEN posteriores se examinan de la misma manera. Si ninguna condición WHEN da como resultado verdadero, el valor de la expresión CASE es el resultado de la cláusula ELSE. Si se omite la cláusula ELSE y ninguna condición es verdadera, el resultado es nulo.

A continuación podemos ver una sintaxis resumida

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

Algunos ejemplos de uso podrían ser estos:

```
select estacion, temperatura_maxima,
        CASE
            WHEN temperatura_maxima > 38 then 'Mucho calor'
            WHEN temperatura_maxima BETWEEN 30 AND 38 then 'Calor'
            ELSE 'No mucho calor'
        END
from climatologia
where provincia = 'Sevilla'
order by temperatura_maxima desc;


select generate_series,
        CASE generate_series
            WHEN 1 THEN 'uno'
            WHEN 2 THEN 'dos'
            WHEN 3 THEN 'tres'
            ELSE 'mayor que tres'
        END
from generate_series(1,5);
```

> La función `generate_series` es una función especial que nos permite generar series de números. Más documentación aquí: https://www.postgresql.org/docs/current/functions-srf.html

Bibliografía

1. Apuntes de Jorge Sánchez (jorgesanchez.net). https://jorgesanchez.net/gbd
2. *Practical SQL. A Begginers guide to storytelling with data* de Anthony DeBarros. https://nostarch.com/practicalSQL
3. https://aws.amazon.com/es/blogs/database/managing-postgresql-users-and-roles/#:~:text=Users%2C%20groups%2C%20and%20roles%20are,for%20the%20CREATE%20ROLE%20statement.
4. https://titiushko.github.io/Tutoriales-Ya/www.postgresqlya.com.ar/temarios/descripcionf38b.html?inicio=0&cod=180&punto=22#:~:text=Las%20funciones%20matem%C3%A1ticas%20realizan%20operaciones,absoluto%20del%20argumento%20%22x%22.
5. https://leafo.net/guides/postgresql-calculating-percentile.html
6. https://www.postgresql.org/docs/current/functions-string.html
7. https://www.digitalocean.com/community/tutorials/how-to-install-configure-pgadmin4-server-mode-es