Talento Digital para Chile:

Módulo 4
Lenguaje de Consultas a una Base de Datos

nodovirtual.awakelab.cl

jelou futurejob by Adalid Chile

MÓDULO 4 – LENGUAJE DE CONSULTAS A UNA BASE DE DATOS

4.2.- Contenido 2: Construir consultas a una base de datos utilizando el lenguaje estructurado de consultas SQL y a partir de un modelo de datos para la obtención de información que satisface los requerimientos planteados

Objetivo de la jornada

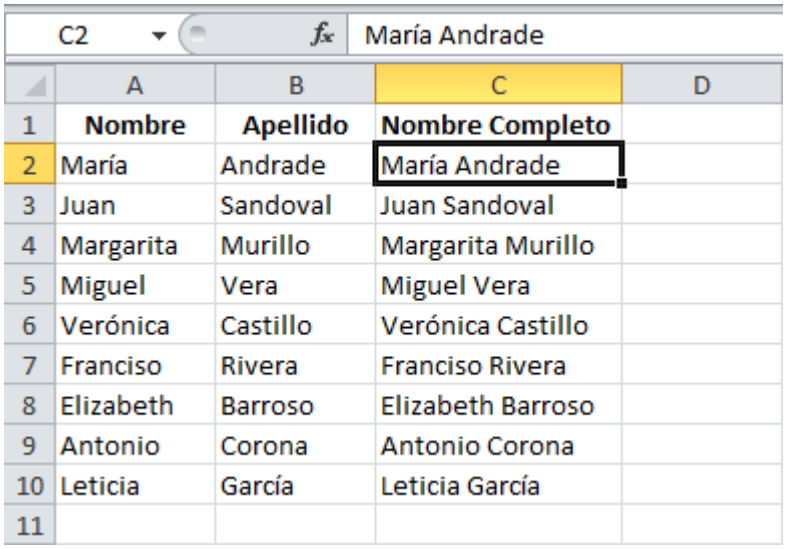
1. Realiza la conexión a una base de datos PostgreSQL utilizando las herramientas utilitarias para su posterior operación.
2. Construye sentencias de creación de una tabla y define campos, tipos de dato, nulidad, llaves primarias y foráneas de acuerdo a un modelo de datos existente para satisfacer un requerimiento
3. Construye consultas utilizando sentencias SQL con condiciones de selección para resolver un problema planteado de selección condicional.
4. Construye consultas utilizando sentencias SQL que requieren la consulta a varias tablas relacionadas a partir de un modelo de datos dado para resolver un problema planteado de selección.
5. Construye consultas utilizando sentencias SQL con funciones de agrupación para resolver un problema planteado que requiere la agrupación de datos.

4.2.1.- Las Bases de Datos Relacionales

Las aplicaciones de hojas de cálculo, como Microsoft Excel, se utilizan ampliamente como una forma de almacenar e inspeccionar datos. Es fácil ordenar los datos de diferentes maneras y ver las características y patrones en los datos con solo mirarlos.

Desafortunadamente, las personas a menudo confunden una herramienta que es buena para inspeccionar y manipular datos con una herramienta adecuada para almacenar y compartir datos complejos y quizás críticos para el negocio. Las dos necesidades suelen ser muy diferentes.

La mayoría de las personas estarán familiarizadas con una o más hojas de cálculo y se sentirán cómodas con los datos organizados en un conjunto de filas y columnas. LibreOffice es otra buena alternativa y hay bastantes más.



C2 — María Andrade

	A	B	C	D
1	**Nombre**	**Apellido**	**Nombre Completo**	
2	María	Andrade	María Andrade	
3	Juan	Sandoval	Juan Sandoval	
4	Margarita	Murillo	Margarita Murillo	
5	Miguel	Vera	Miguel Vera	
6	Verónica	Castillo	Verónica Castillo	
7	Franciso	Rivera	Franciso Rivera	
8	Elizabeth	Barroso	Elizabeth Barroso	
9	Antonio	Corona	Antonio Corona	
10	Leticia	García	Leticia García	
11				

Esta sencilla hoja, arriba, de cálculo incorpora varias características que serán utiles de recordar cuando comencemos a diseñar bases de datos. Por ejemplo, el nombre y los apellidos se mantienen en columnas separadas, lo que facilita la clasificación de los datos por apellido si es necesario. Entonces, ¿qué hay de malo en almacenar información del cliente en una hoja de cálculo? Las hojas de cálculo están bien, siempre y cuando:

- No tenga demasiados clientes
- No tenga muchos detalles complejos para cada cliente
- No sea necesario almacenar ninguna otra información repetida, como los distintos pedidos que ha realizado cada cliente.
- No quiera que varias personas puedan actualizar la información simultáneamente
- Asegurarse de que se realice una copia de seguridad de la hoja de cálculo con regularidad si contiene datos importantes

Las hojas de cálculo son una idea fantástica y son excelentes herramientas para muchos tipos de problemas. Sin embargo, así como no intentaría (o al menos no debería) intentar clavar un clavo con un destornillador, a veces las hojas de cálculo no son la herramienta adecuada para el trabajo.

Imagínese cómo sería si una gran empresa, con decenas de miles de clientes, mantuviera la copia maestra de su lista de clientes en una simple hoja de cálculo. En una gran empresa, es probable que varias personas necesiten actualizar la lista. Aunque el bloqueo de archivos puede garantizar que solo una persona actualice la lista a la vez, a medida que aumenta el número de personas que intentan actualizar la

lista, pasarán más y más tiempo esperando su turno para editar la lista. Lo que nos gustaría es permitir que muchas personas lean, actualicen, agreguen y eliminen filas simultáneamente, y que la computadora se asegure de que no haya conflictos. Claramente, el bloqueo simple de archivos no será adecuado para manejar este problema de manera eficiente.

Otro problema con las hojas de cálculo son sus estrictas dos dimensiones. Supongamos que también quisiéramos almacenar detalles de cada pedido que realizó un cliente. Podríamos comenzar a poner la información de los pedidos junto a cada cliente, pero a medida que aumentara la cantidad de pedidos por cliente, la hoja de cálculo se volvería cada vez más compleja.

Un sistema de administración de bases de datos (DBMS) suele ser un conjunto de bibliotecas, aplicaciones y utilidades que alivian al desarrollador de aplicaciones de la carga de preocuparse por los detalles de almacenamiento y administración de datos. También proporciona facilidades para buscar y actualizar registros. Los DBMS vienen en varios tipos desarrollados a lo largo de los años para resolver tipos particulares de problemas de almacenamiento de datos.

4.2.1.1. El rol de las bases de dato relacionales

Cuando se mira superficialmente, una base de datos relacional (RDB), como PostgreSQL, tiene muchas similitudes con una hoja de cálculo. Sin embargo, cuando conoce la estructura subyacente de una base de datos, puede ver que es mucho más flexible, principalmente debido a su capacidad para relacionar tablas de formas complejas. Puede almacenar de manera eficiente datos mucho más complejos que una hoja de cálculo, y también tiene muchas otras características que lo convierten en una mejor opción como almacén de datos. Por ejemplo, una base de datos puede administrar varios usuarios simultáneamente.

Pensemos en almacenar nuestra lista de clientes simple de una sola hoja en una base de datos, para ver qué beneficios podría tener.

Las bases de datos se componen de tablas, o en terminología más formal, relaciones. Nos ceñiremos al uso del término tablas. Una tabla contiene filas de datos y cada fila de datos consta de varias columnas o atributos.

4.2.1.2. Características de un RDBMS

Varias reglas importantes definen un sistema de gestión de bases de datos relacionales (RDBMS). Todas las filas deben seguir el mismo patrón, ya que todas

tienen el mismo número y tipo de componentes. A continuación, se muestra un ejemplo de un conjunto de filas:

```
{"Chile", "CLP", 1000,56}
{"Bélgica", "EUR", 1,34}
```

Cada una de estas filas tiene tres atributos: un nombre de país (cadena), una moneda (cadena) y un tipo de cambio (un número de punto flotante). En una base de datos relacional, todos los registros que se agregan a este conjunto, o tabla, deben seguir el mismo formulario, por lo que no se permiten los siguientes:

```
{"Germany", "EUR"}
```

Esto tiene muy pocos atributos.

```
{"Switzerland", "CHF", "French", "German", "Italian", "Romansch"}
```

Esto tiene demasiados atributos.

```
{1936.27, "EUR", "Italy"}
```

Esto tiene tipos de atributos incorrectos (orden incorrecto).

Además, en cualquier tabla de filas, no debe haber duplicados. Esto significa que en cualquier tabla en una base de datos relacional correctamente diseñada, no puede haber filas o registros idénticos. Esto podría parecer una restricción bastante draconiana. Por ejemplo, en un sistema que registra los pedidos realizados por los clientes, parecería no permitir que el mismo cliente pida el mismo producto dos veces. Más adelante, veremos que hay una manera fácil de evitar este tipo de problemas/requisitos agregando un atributo.

Cada atributo de un registro debe ser "atómico"; es decir, debe ser un solo dato, no otro registro o una lista de otros atributos. Además, el tipo de atributos correspondientes en cada registro de la tabla (elementos de la columna) debe ser el mismo. Técnicamente, esto significa que deben extraerse del mismo conjunto de valores o dominio. En términos prácticos, significa que todos serán una cadena, un número entero, un valor de punto flotante o algún otro tipo admitido por el sistema de base de datos.

El atributo (o atributos) que se utilizan para distinguir un registro en particular en una tabla de todos los otros registros se llama clave primaria o clave principal (primary key). En una base de datos relacional, cada relación o tabla debe tener una clave principal para cada registro para que sea único, diferente de todos los demás en esa tabla.

Una última regla que determina la estructura de una base de datos relacional es la "integridad referencial" (lo veremos más adelante con mayor detalle). Como señalamos anteriormente, este es el deseo de que todos los registros de la base de datos tengan sentido en todo momento. Los programadores de aplicaciones de bases de datos deben tener cuidado de asegurarse de que su código no rompa la integridad de la base de datos. Considere lo que sucede cuando eliminamos un cliente. Si intentamos eliminar al cliente de la relación con el cliente, también debemos eliminar todos sus pedidos de la tabla de pedidos. De lo contrario, nos quedarán registros de pedidos que no tienen un cliente válido.

Veremos mucho más sobre la teoría y la práctica de las bases de datos relacionales posteriormente. Por ahora, es suficiente saber que el modelo relacional para bases de datos se basa en algunos conceptos matemáticos de conjuntos y relaciones, y que hay algunas reglas que deben ser observadas por los sistemas que se basan en este modelo.

4.2.1.3. Configurando el motor de base de datos

Para este tópico vamos a trabajar con PostgresSQL https://www.postgresql.org/

PostgresSQL es uno de los productos de software de código abierto más exitosos de los últimos tiempos, en el campo de las bases de datos relacionales. PostgreSQL está encontrando una audiencia entusiasta entre los aficionados a las bases de datos y los desarrolladores de código abierto por igual. Cualquiera que esté creando una aplicación con cantidades de datos no triviales puede beneficiarse del uso de una base de datos. PostgreSQL es una excelente implementación de una base de datos relacional, con todas las funciones, de código abierto y de uso gratuito.

PostgreSQL se puede usar desde casi cualquier lenguaje de programación importante que le interese nombrar, incluidos C, C ++, Perl, Python, Java, Tcl y PHP. Sigue muy de cerca el estándar de la industria para lenguajes de consulta, SQL (el cual abordaremos más adelante)

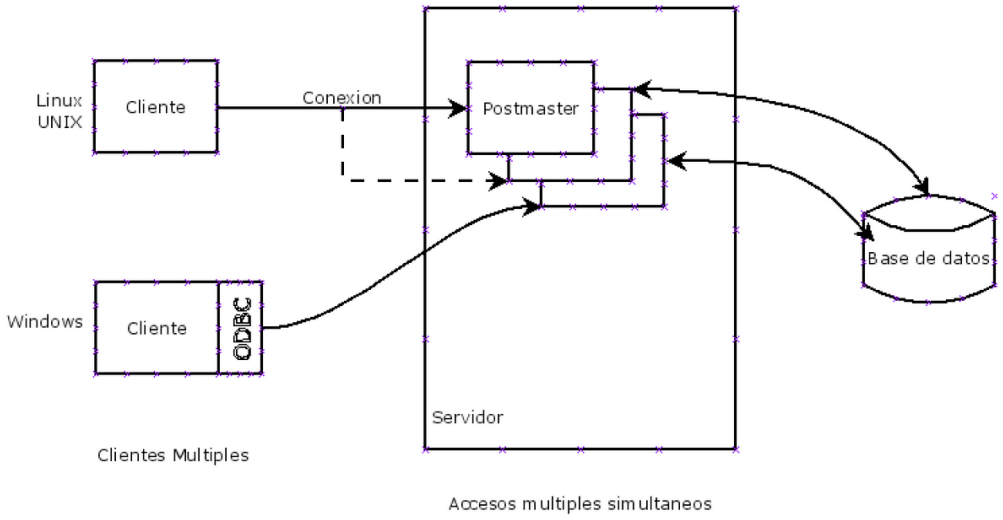
Para descargar este software diríjase a https://www.postgresql.org/download/ y elija/instale la versión para su sistema operativo.

Nota: Como vamos a trabajar localmente no hay necesidad de configurar parámetros especiales. El material ha sido probado con la versión 12.4.

Una de las fortalezas de PostgreSQL se deriva de su arquitectura. Al igual que los sistemas de bases de datos comerciales, PostgreSQL se puede utilizar en un entorno cliente/servidor. Esto tiene muchos beneficios tanto para los usuarios como para los desarrolladores.

El corazón de una instalación de PostgreSQL es el proceso del servidor de base de datos. Funciona en un solo servidor. Las aplicaciones que necesitan acceder a los datos almacenados en la base de datos deben hacerlo a través del proceso de la base de datos. Estos programas cliente no pueden acceder a los datos directamente, incluso si se ejecutan en la misma computadora que el proceso del servidor como es nuestro caso.

Esta separación en cliente y servidor permite distribuir las aplicaciones. Puede utilizar una red para separar a sus clientes de su servidor y desarrollar aplicaciones de cliente en un entorno que se adapte a los usuarios. Por ejemplo, puede implementar la base de datos en UNIX y crear programas cliente que se ejecuten en Microsoft Windows. La Figura muestra una aplicación PostgreSQL distribuida típica.



En la Figura, puede ver varios clientes que se conectan al servidor a través de una red. Para PostgreSQL, esto debe ser una red TCP / IP, una red de área local (LAN) o posiblemente incluso Internet. Cada cliente se conecta al proceso del servidor de la base de datos principal (que se muestra como **postmaster** en la Figura), que crea un nuevo proceso de servidor específicamente para atender las solicitudes de acceso para este cliente.

Concentrar el manejo de datos en un servidor, en lugar de intentar controlar a muchos clientes que acceden a los mismos datos almacenados en un directorio compartido en un servidor, permite a PostgreSQL mantener eficientemente la integridad de los datos, incluso con muchos usuarios simultáneos.

Los programas cliente se conectan mediante un protocolo de mensajes específico de PostgreSQL. Sin embargo, es posible instalar software en el cliente que proporcione una interfaz estándar para que funcione la aplicación, como el estándar Open Database Connectivity (ODBC) o el estándar Java Database Connectivity (JDBC) utilizado por los programas Java, algún conector para Python, PHP y otros lenguajes.

La disponibilidad de un controlador ODBC permite que muchas aplicaciones existentes utilicen PostgreSQL como base de datos, incluidos productos de Microsoft Office como Excel y Access.

La arquitectura cliente/servidor de PostgreSQL permite una división del trabajo. Una máquina servidor adecuada para el almacenamiento y acceso de grandes cantidades de datos se puede utilizar como un depósito de datos seguro. Se pueden desarrollar sofisticadas aplicaciones gráficas para los clientes. Alternativamente, se puede crear un front-end basado en la web para acceder a los datos y devolver los resultados como páginas web a un navegador web estándar, sin ningún software de cliente adicional.

4.2.1.4. Acceso a la base datos a través del terminal

Una instalación de postgreSQL usualmente tiene los directorios:

- **bin**: Aplicaciones y utilidades como **pg_ctl** y **postmaster** (**postgres**)
- **include**: Archivos de encabezado para usar en el desarrollo de aplicaciones PostgreSQL (header files)
- **lib**: Bibliotecas para usar en el desarrollo de aplicaciones PostgreSQL.
- **share**: Archivos de configuración, man pages and documentos (en carpetas separadas).

Una vez que postgreSQL ha sido instalado el cree un **<directorio_de_trabajo>** (con el nombre que usted desee) y configure la variable **PGDATA** para que apunte a tal directorio.

Ejemplo en Linux, OSx:

```
$ export PGDATA="<directorio_de_trabajo>"
```

Luego puede inicializar postgreSQL, utilizando **initdb**.

- **initdb**: Es el binario que inicializa el área de datos de PostgreSQL. El directorio para inicializar debe estar vacío. Se pueden especificar varias opciones en la línea de comando, como la codificación de caracteres o el orden de clasificación.

```
$ initdb
...
The files belonging to this database system will be owned by user
"usuario de la maquina".
This user must also own the server process
....
Success. You can now start the database server using:

pg_ctl -D <directorio_de_trabajo> -l logfile start
```

Al ejecutar

```
pg_ctl -D <directorio_de_trabajo> -l logfile start
waiting for server to start.... done
server started
```

Estamos listos para experimentar.

Nota: podríamos usar la sentencia **postgres** pero mejor usamos la variante que simplifica los comandos **pg_ctl**:

- **postgres**: es el proceso principal de PostgreSQL. El programa se puede iniciar directamente o usando la utilidad **pg ctl**. Se prefiere el segundo método, ya que ofrece una forma más sencilla de controlar el proceso de postgres.

Si examina su **<directorio_de_trabajo>** encontrará una serie de archivos y directorios creados por postgresql.

4.2.1.5. Crear una base de datos desde el shell de postgreSQL

Podemos usar el terminal interactivo de postgresql **psql**; **template1** es una base de datos creada por el propio postgres y está presente en todas las instalaciones.

En nuestro terminal podemos ejecutar **psql -d template1** y deberíamos ver algo similar a:

```
$ psql -d template1
psql (12.4)
Type "help" for help.

template1=#
```

Dentro de **psql** podemos ver una lista de databases existentes con **\l o \list**

```
template1=# \l
```

Este comando mostrará las bases de datos que tiene su sistema en el momento de ejecutarlo.

Ahora crearemos una base de datos utilizando **createdb** desde nuestro terminal (no desde terminal interactivo de postgresql)

```
$ createdb mibasededatos
```

Ahora si podemos ingresar el terminar interactivo **psql** para examinar nuestra propia base de datos, ejecutando **psql -d mibasededatos**

```
$ psql -d mibasededatos
psql (12.4)
Type "help" for help.

mibasededatos=#
```

Utilizando **\list (\l)** podrá ver una nueva base de datos en la lista: **mibasededatos**

Nota: para dejar el terminal interactivo de postgresql puede presionar **CTRL+d** o escribir **quit**.

4.2.1.6. Crear usuarios desde el shell de postgreSQL

Ahora crearemos un usuario, para lo cual, dentro del terminal interactivo, utilizaremos el lenguaje SQL. SQL (Lenguaje de consultas estructurado - Structured Query Language) es un lenguaje de programación ampliamente utilizado que le permite definir y consultar bases de datos. Examinaremos SQL más adelante con mayor profundidad, por lo pronto vamos a usar algunos comandos básicos para captar la esencia de este.

Creemos una base de datos nueva estando dentro del ambiente proporcionado por el terminal interactivo **psql**

```
mibasededatos=# CREATE USER usuario WITH ENCRYPTED PASSWORD
'mipass';CREATE ROLE
```

Para ratificar que el usuario fue creado podemos utilizar el meta-comando de postgres **\du**

```
mibasededatos=# \du

                             List of roles
Role name |                   Attributes                | Member of
----------+---------------------------------------------+--------
usuario   |                                             | {}
```

4.2.1.7. Listar bases de dato desde el shell de postgreSQL

Como vimos anteriormente podemos ver la lista de base de datos utilizando el meta-comando de postgres **\l**. Pero si nuestra preferencia es hacerlo utilizando lenguaje SQL, podemos utilizar:

```
mibasededatos=# SELECT datname FROM pg_database;
```

Podemos ver en los resultados que la tabla **pg_database** recopila información sobre las bases de datos disponibles en PostgreSQL.

4.2.1.8. Ingresar a una base de datos desde el shell de postgreSQL

Si queremos ingresar al terminal interactivo postgresql y a nuestra base de datos inmediatamente, podemos hacer

```
$ psql -U <su_usuario> mibasededatos
```

o simplemente **$ psql -d mibasededatos** si su usuario está configurado por defecto.

Si deseamos cambiarnos a otra base, por ejemplo, a **template1** dentro de **psql** simplemente podemos hacer

```
mibasededatos=# \connect template1
You are now connected to database "template1" as user "su usuario".
template1=#
```

o aún más simple **\c** en lugar de **\connect**

4.2.2. Consultando información de una tabla

4.2.2.1. El Lenguaje Estructurado de Consultas SQL

SQL es un lenguaje de programación ampliamente utilizado que le permite definir y consultar bases de datos. Ya sea que sea un analista de marketing, un periodista o un investigador que mapea neuronas en el cerebro de una mosca de la fruta, se beneficiará del uso de SQL para administrar objetos de bases de datos, así como para crear, modificar, explorar y resumir datos.

Dado que SQL es un lenguaje maduro que existe desde hace décadas, está profundamente arraigado en muchos sistemas modernos. Un par de investigadores de IBM describieron por primera vez la sintaxis de SQL (entonces llamado SEQUEL) en un artículo de 1974, basándose en el trabajo teórico del científico informático británico Edgar F. Codd. En 1979, una precursora de la empresa de bases de datos Oracle (entonces llamada Relational Software) se convirtió en la primera en utilizar el lenguaje en un producto comercial. Hoy en día, sigue siendo uno de los lenguajes informáticos más utilizados en el mundo y es poco probable que eso cambie pronto.

SQL viene en varias variantes, que generalmente están vinculadas a sistemas de bases de datos específicos. El Instituto Nacional Estadounidense de Estándares (ANSI) y la Organización Internacional de Normalización (ISO), que establecen estándares para productos y tecnologías, proporcionan estándares para el idioma y lo revisan. La buena noticia es que las variantes no se alejan mucho del estándar, por lo que una vez que aprenda las convenciones SQL para una base de datos, podrá transferir ese conocimiento a otros sistemas.

Entonces, ¿por qué debería usar SQL? Después de todo, SQL no suele ser la primera herramienta que las personas eligen cuando están aprendiendo a analizar datos. De hecho, muchas personas comienzan con hojas de cálculo de Microsoft Excel y su variedad de funciones analíticas. Después de trabajar con Excel, es posible que se pasen a Access, el sistema de base de datos integrado en Microsoft Office, que tiene una interfaz gráfica de consulta que facilita el trabajo, lo que hace que las habilidades de SQL sean opcionales.

Pero, como ya explicamos anteriormente en esta clase, Excel y Access tienen sus límites. Excel actualmente permite un máximo de 1.048.576 filas por hoja de trabajo, y Access limita el tamaño de la base de datos a dos gigabytes y limita las columnas a 255 por tabla. No es infrecuente que los conjuntos de datos superen esos límites, especialmente cuando se trabaja con datos descargados de sistemas gubernamentales. El último obstáculo que desea descubrir al enfrentarse a una fecha límite es que su sistema de base de datos no tiene la capacidad para realizar el trabajo.

El uso de un sólido sistema de base de datos SQL le permite trabajar con terabytes de datos, múltiples tablas relacionadas y miles de columnas. Le brinda un control programático mejorado sobre la estructura de sus datos, lo que genera eficiencia, velocidad y, lo más importante, precisión.

SQL también es un excelente complemento de los lenguajes de programación utilizados en ciencias de datos, como R y Python. Si usa cualquiera de los esos lenguajes, por ejemplo, puede conectarse a bases de datos SQL y, en algunos casos, incluso incorporar la sintaxis SQL directamente en el lenguaje. Para las personas sin experiencia en lenguajes de programación, SQL a menudo sirve como una introducción fácil de entender a los conceptos relacionados con las estructuras de datos y la lógica de programación.

Además, conocer SQL puede ayudarle más allá del análisis de datos. Si profundiza en la creación de aplicaciones en línea, encontrará que las bases de datos brindan el poder de backend para muchos frameworks web comunes, mapas interactivos y sistemas de administración de contenido. Cuando necesite excavar debajo de la superficie de estas aplicaciones, la capacidad de SQL para manipular datos y bases de datos será muy útil.

4.2.2.2. Recuperando información de una tabla

Para los ejemplos continuaremos utilizando la base de datos creada anteriormente **mibasededatos** (si usted creó una base de datos con un nombre distinto o desea crear una nueva puede seguir las instrucciones de la primera parte de esta clase)

La sintaxis para crear una tabla en nuestra base de datos es la siguiente:

```
CREATE TABLE profesores (
    id bigserial,
    nombre varchar(25),
    apellido varchar(50),
    escuela varchar(50),
    fecha_de_contratacion date,
    sueldo numeric
);
```

Utilizando el terminal interactivo de postgresql, tenemos

```
mibasededatos=# CREATE TABLE profesores (
                  id bigserial,
                  nombre varchar(25),
                  apellido varchar(50),
                  escuela varchar(50),
                  fecha_de_contratacion date,
                  sueldo numeric
                );
CREATE TABLE
```

Para entender mejor las consultas (queries) se suele expandir su contenido:

```
CREATE TABLE profesores
  (
     id                    BIGSERIAL,
     nombre                VARCHAR(25),
     apellido              VARCHAR(50),
     escuela               VARCHAR(50),
     fecha_de_contratacion DATE,
     sueldo                NUMERIC
  );
```

Utilizando el terminal interactivo de postgresql, también podemos usar una sintaxis extendida obteniendo el mismo resultado **CREATE TABLE** que indica que la tabla fue creada.

```
mibasededatos=# CREATE TABLE profesores
mibasededatos-#   (
mibasededatos(#   id                    BIGSERIAL,
mibasededatos(#   nombre                VARCHAR(25),
mibasededatos(#   apellido              VARCHAR(50),
```

```
mibasededatos(#    escuela                VARCHAR(50),
mibasededatos(#    fecha_de_contratacion DATE,
mibasededatos(#    sueldo                 NUMERIC
mibasededatos(#    );
CREATE TABLE
```

Para verificar que nuestra fue creada exitosamente podemos utilizar (estando dentro del terminal interactivo de postgresql)

```
\d profesores
```

El resultado será una Descripción de la tabla.

Ahora bien, para cada columna (id, nombre, apellido) utilizamos un tipo de datos especifico.

Dado que ya tenemos la tabla (profesores vamos a llenarla con algunos datos), para tal propósito utilizaremos la instrucción **INSERT** de SQL:

```
INSERT INTO profesores2 (
  nombre, apellido, escuela, fecha_de_contratacion, sueldo)
  VALUES
('Juanita', 'Perez', 'Gabriela Mistral', '2011-10-30', 234000),
('Bruce', 'Lee', 'Republica Popular China', '1993-05-22', 780945),
('Juan Alberto', 'Valdivieso', 'Sagrada Concepcion', '2005-08-01', 3400000),
('Pablo', 'Rojas', 'E-34', '2011-10-30', 300000),
('Nicolas', 'Echenique', 'Bendito Corazón de María', '2005-08-30', 8900000),
('Jericho', 'Jorquera', 'A-18 Abrazo de Maipu', '2010-10-22', 67500);
```

También, podemos insertar filas individuales o una a una:

```
INSERT INTO profesores (
     nombre, apellido, escuela, fecha_de_contratacion, sueldo)
   VALUES
     ('Caupolicán','Catrileo','Santiago de la extremadura',
      '2000-10-26', 780000);
```

Observe que ciertos valores que estamos insertando están entre comillas simples, pero otros no. Este es un requisito estándar de SQL. El texto y las fechas requieren citas; los números, incluidos los números enteros y decimales, no requieren comillas.

Note que no se utilizó cifra alguna para la columna **id**, que es la primera columna de la tabla. Esto sucede porque cuando creó la tabla, su secuencia de comandos especificó que esa columna fuera el tipo de datos de **bigserial**. Entonces, cuando PostgreSQL inserta cada fila, automáticamente llena la columna id con un número entero que se incrementa automáticamente como veremos a continuación cuando recuperemos (o leamos) la información de nuestra tabla **profesores** utilizando la sentencia **SELECT** de SQL. Una instrucción **SELECT** puede ser simple, recuperando todo

en una sola tabla, o puede ser lo suficientemente compleja como para vincular docenas de tablas mientras maneja múltiples cálculos y filtra por criterios exactos.

Aquí hay una declaración SELECT que recupera cada fila y columna de la tabla llamada **profesores**

```
SELECT * FROM profesores;
```

```
mmibasededatos=# SELECT * FROM profesores;
```

```
Id |   nombre     |   apellido   |   escuela           |fecha_de_contratacion| sueldo
---+--------------+--------------+- -------------------+---------------------+-------
1  | Juanita      | Perez        | Gabriela Mistral    | 2011-10-30          |234000
2  | Bruce        | Lee          | Rep. Pop. China     | 1993-05-22          |78094
3  | Juan Alberto | Valdivieso   | Sagr. Concepcion    | 2005-08-01          |340000
4  | Pablo        | Rojas        | E-34                | 2011-10-30          |300000
5  | Nicolas      | Echenique    | B.Corazón de María  | 2005-08-30          |8900000
6  | Jericho      | Jorquera     | A-18 Abrazo de Maipu| 2010-10-22          |67500
7  | Caupolicán   | Catrileo     | S,de la extremadura | 2000-10-26          |780000
(7 rows)
```

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 un sustituto de un valor: no representa nada en particular y, en cambio, representa todo lo que ese valor podría ser. Aquí, es la abreviatura de "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 que desea que la consulta devuelva datos de una tabla en particular. El punto y coma después del nombre de la tabla le dice a PostgreSQL que es el final de la declaración de consulta.

También podemos consultar un subconjunto de columnas:

```
SELECT nombre, apellido FROM profesores;
```

```
mibasededatos=# SELECT nombre, apellido FROM profesores;
```

```
nombre         |  apellido
---------------+------------
Juanita        | Perez
Bruce          | Lee
Juan Alberto   | Valdivieso
Pablo          | Rojas
Nicolas        | Echenique
Jericho        | Jorquera
Caupolicán     | Catrileo
(7 rows)
```

4.2.2.3. Consultas utilizando la llave primaria

Una llave primaria o clave principal (primary key) es una columna o colección de columnas cuyos valores identifican de forma única cada fila de una tabla. Una columna de clave primaria válida impone ciertas restricciones:

- La columna o colección de columnas debe tener un valor único para cada fila.
- La columna o colección de columnas no puede tener valores faltantes.

En el caso que hemos visto hasta ahora con la tabla **profesores**, el campo **id** es una especie de llave primaria por defecto. Fíjese que cumple con las restricciones, pero podemos definir nuestras propias llaves primarias.

La llave primaria se puede definir al momento de creación de la tabla:

```
CREATE TABLE profesores
 (
    id                    BIGSERIAL,
    nombre                VARCHAR(25),
    apellido              VARCHAR(50),
    escuela               VARCHAR(50),
    fecha_de_contratacion DATE,
    sueldo                NUMERIC,
    CONSTRAINT nombre_key PRIMARY KEY (nombre)
 );
```

O también la podemos agregar una vez que la tabla ya ha sido creada:

```
ALTER TABLE profesores ADD PRIMARY KEY (nombre);
```

Si observamos la descripción de la tabla usando **\d;** notaremos que la llave primaria aparece como índice

```
mibasededatos=# \d profesores;
...
Indexes:
    "profesores_pkey" PRIMARY KEY, btree (nombre)
```

Ahora bien, dado que **nombre** es nuestra llave primaria debe cumplir con las restricciones anteriormente descritas, es decir no podemos, por ejemplo, en este caso duplicar un nombre:

```
mibasededatos=# INSERT INTO profesores (nombre, apellido, escuela,
fecha_de_contratacion, sueldo) VALUES ('Pablo', 'Lizarraga', 'Sagrado
Grial de Montegrande', '2004-08-21', 6800000);
ERROR:      duplicate   key   value   violates   unique   constraint
"profesores2_pkey"
DETAIL:  Key (nombre)=(Pablo) already exists.
```

Naturalmente nuestro ejemplo es solo ilustrativo, los nombres de personas se repiten frecuentemente en una base de datos, no obstante, no hay dos personas que tengan el mismo número de carné, tal condición es ideal para crear una llave primaria si su diseño y tabla lo requieren.

4.2.2.4. Consultas utilizando condiciones de selección

Las consultas que requieren criterios de selección son naturalmente una práctica absolutamente necesaria y frecuente. Veremos algunas típicas, existen muchos criterios de selección a la hora de ejecutar estas consultas.

Anteriormente vimos la sintaxis básica para la instrucción **SELECT**

```
SELECT * FROM profesores;
```

El uso del comodín de asterisco es útil para descubrir el contenido completo de una tabla. Pero a menudo es más práctico limitar las columnas que recupera la consulta, especialmente con grandes bases de datos.

Podemos elegir solo campos que nos interesan descartando los que no nos sirven para una consulta en particular. La regla general es

```
SELECT una_columna, otra_columna, super_columna FROM tabla;
```

Ejemplo:

```
mibasededatos=# SELECT apellido, escuela, sueldo FROM profesores;

apellido   |         escuela                  | sueldo
-----------+----------------------------------+---------
Perez      | Gabriela Mistral                 | 234000
Lee        | Republica Popular China          | 780945
Valdivieso | Sagrada Concepcion               | 3400000
Rojas      | E-34                             | 300000
Echenique  | Bendito Corazón de María         | 8900000
Jorquera   | A-18 Abrazo de Maipu             | 67500
(6 rows)
```

Con esa sintaxis, la consulta recupera todas las filas de solo esas tres columnas. El orden lo podemos elegir de acuerdo a nuestras necesidades, por ejemplo:

```
SELECT sueldo, apellido, escuela FROM profesores;
```

Aunque estos ejemplos son básicos, ilustran una buena estrategia para comenzar a consultar un conjunto de datos. Generalmente, es aconsejable comenzar su análisis verificando si sus datos están presentes y en el formato que espera.

Para este ejemplo primero insertemos una nuevo file con un apellido y una escuela repetida:

```
INSERT INTO profesores (nombre, apellido, escuela, fecha_de_contratacion, sueldo) VALUES ('Wong', 'Lee', 'Santiago de la extremadura', '2000-10-26', 780000);
```

Supongamos que necesitamos saber solo las escuelas que existen en nuestros registros, eso lo podemos lograr con **DISTINCT**

```
mibasededatos=# SELECT DISTINCT escuela FROM profesores;
         escuela
---------------------------
E-34
Santiago de la extremadura
Bendito Corazón de María
Republica Popular China
Gabriela Mistral
Sagrada Concepcion
A-18 Abrazo de Maipu
```

Pese a que los profesores Catrileo y Wong Lee trabajan en la misma escuela, solo necesitamos las escuelas y obviamente dejar de lado repeticiones innecesarias.

Podríamos ordenar la información para que nos resultara más legible, por ejemplo, alfabéticamente usando **ORDER BY** (y ascendente **ASC** o decreciente **DESC**):

```
SELECT DISTINCT escuela, sueldo FROM profesores ORDER by escuela ASC;

mibasededatos=# SELECT DISTINCT escuela, sueldo FROM profesores ORDER by escuela ASC;

escuela                    | sueldo
---------------------------+---------
A-18 Abrazo de Maipu       |   67500
Bendito Corazón de María   | 8900000
 E-34                      |  300000
Gabriela Mistral           |  234000
Republica Popular China    |  780945
Sagrada Concepcion         | 3400000
Santiago de la extremadura |  780000
(7 rows)
```

Una construcción común en las consultas, son aquellas que usan la palabra clave **WHERE** (donde). Por ejemplo, si solo queremos saber los profesores que trabajan en cierta escuela podríamos ejecutar

```
SELECT * FROM profesores
```

```
  WHERE escuela = 'Santiago de la extremadura';
```

Resultando en:

```
mibasededatos=# SELECT * FROM profesores
  WHERE escuela = 'Santiago de la extremadura';
```

```
id| nombre    | apellido |  escuela                 |fecha_de_contratacion |  sueldo
--+-----------+----------+--------------------------+----------------------+------
7 |Caupolicán |Catrileo  |Santiago de la extremadura|   2000-10-26         | 780000
9 | Wong      |Lee       |Santiago de la extremadura|   2000-10-26         | 780000
```

Existen muchos Operadores que se pueden usar en conjunto con **WHERE**

- **=** Igual a - ej. **WHERE escuela = 'E-34'**
- **<>** o **!=** Distinto de - ej. ``WHERE escuela <> 'E-34'``
- **>** Mayor a - ej. **WHERE sueldo > 100000**
- **<** Menor que - ej. **WHERE sueldo < 1000000**
- **>=** Mayor o igual que - ej. **`WHERE sueldo >= 100000`**
- **<=** Menor o igual que - ej. **WHERE sueldo <= 1000000**
- **BETWEEN** Dentro de un rango - ej. **WHERE sueldo BETWEEN 100000 AND 600000**
- **IN** Coincidir con un grupo de valores - ej. `WHERE apellido IN ('Catrileo', 'Perez')`
- **LIKE** Coincidir con un patrón (distingue mayúsculas y minúsculas) - ej. **WHERE apellido LIKE 'Catri%'**
- **ILIKE** Coincidir con un patrón (No distingue mayúsculas y minúsculas) - ej. **WHERE apellido LIKE 'catri%'**
- **NOT** Niega una condición - ej. **WHERE apellido NOT LIKE 'catri%'**

Para ilustrar vamos a ejecutar una de estas instrucciones, el resto se pueden ratificar de forma similar.

```
SELECT * FROM profesores
  WHERE apellido NOT ILIKE 'catri%';
```

Resulta en

```
mibasededatos=# SELECT * FROM profesores WHERE apellido NOT ILIKE 'catri%';
```

```
id| nombre     | apellido |  escuela                 |fecha_de_contratacion| sueldo
--+------------+----------+--------------------------+---------------------+-------
1 |Juanita     | Perez    |Gabriela Mistral          | 2011-10-30          | 234000
2 | Bruce      | Lee      |Republica Popular China   | 1993-05-22          | 780945
3 |Juan Alberto|Valdivieso|Sagrada Concepcion        | 2005-08-01          | 3400000
4 | Pablo      | Rojas    |E-34                      | 2011-10-30          | 300000
5 | Nicolas    | Echenique|Bendito Corazón de María  | 2005-08-30          | 8900000
6 | Jericho    | Jorquera | A-18 Abrazo de Maipu     | 2010-10-22          | 67500
9 | Wong       | Lee      |Santiago de la extremadura| 2000-10-26          | 780000
```

4.2.2.5. Utilización de funciones en las consultas

Al igual que con el número de operadores, el número de funciones integradas de PostgreSQL es enorme. También vamos a explorar solo algunas de las más comunes, la documentación en línea tiene todo lo que usted pueda necesitar para tareas más específicas.

- **current_date**: Devuelve la fecha de hoy.
- **current_time**: Devuelve la hora actual (no se devuelve información de fecha).
- **current_timestamp**: Devuelve una marca de tiempo (fecha y hora) de la hora actual.
- **`date_part(text,timestamp)**: Devuelve un campo especificado en el texto de una marca de tiempo determinada.
- **now()**: Devuelve un campo especificado en el texto de una marca de tiempo determinada.
- **timeofday()**: Devuelve la salida de texto de la hora actual. Este valor aumenta durante las transacciones.

Las funciones antes descritas se pueden probar fácilmente usando SELECT

```
SELECT current_time;

mibasededatos=# SELECT current_time;
    current_time
--------------------
16:07:22.289126+02
(1 row)
```

Las funciones de cadena (string) se pueden usar para manipular valores de cadena de varias formas. Para estas funciones, cualquier entrada de cadena, incluidas las cadenas de texto, varchar y char, se considerará una entrada válida para la función.

- **lower(string)**: Devuelve la cadena en minúsculas.
- **position(substring in string)**: Devuelve la posición entera de una subcadena dentro de la cadena.
- **split_part(string,delimiter,field)**: Divide la cadena usando un delimitador y devuelve un campo especificado.
- **substring(string,from,[for])**: Extrae una subcadena de la cadena a partir de los dígitos especificados.
- **replace(string,from,to)**: Reemplaza texto por texto en una cadena determinada.
- **upper(string)**: Devuelve la cadena en mayúsculas.

Ejemplo

```
SELECT replace('Monito', 'ito', 'oto');


# SELECT replace('Monito', 'ito', 'oto');
replace
---------
Monoto
(1 row)
```

4.2.2.6. Consultas de selección con funciones de agrupación (COUNT y SUM)

SQL proporciona una serie de funciones que le permiten realizar recuentos, promedios y otras operaciones agregadas. Algunas de las funciones agregadas más comunes son:

- **avg(expression)**: El promedio de todos los valores de entrada. Los valores de entrada deben ser uno de los tipos enteros.
- **count(*)**: El número de valores de entrada.
- **count(expression)**: El número de valores de entrada no nulos.
- **max(expression)**: El valor máximo de expresión para todos los valores de entrada.
- **min(expression)**: Los valores mínimos de expresión para todos los valores de entrada.
- **sum(expression)**: La suma de la expresión para todos los valores de entrada no nulos.

Utilizando nuestra base datos de ejemplo de apartados anteriores, busquemos la cantidad de profesores que fueron contratados a partir del año 2010.

```
SELECT
   count(DISTINCT escuela)
FROM
  profesores
WHERE
 fecha_de_contratacion > 01-01-2010;


mibasededatos=# SELECT
  count(DISTINCT escuela)
FROM
  profesores
WHERE
  fecha_de_contratacion > '01-01-2010';
count
-------
     3
(1 row)
```

O sumemos la cantidad de dinero mensual gastada en todos los profesores registrados

```
SELECT sum(sueldo) AS total FROM profesores;
```

```
mibasededatos=# SELECT sum(sueldo) AS total FROM profesores;
  total
----------
 15242445
(1 row)
```

4.2.3. Consultando información relacionada en varias tablas

4.2.3.1. Qué es un modelo de datos y cómo leerlo

Modelos de Datos

Durante las décadas de 1960 y 1970, los desarrolladores crearon bases de datos que resolvieron el problema de los grupos repetidos de varias formas diferentes. Estos métodos dan como resultado lo que se denominan modelos para sistemas de bases de datos. La investigación realizada en IBM proporcionó gran parte de la base para estos modelos, que todavía se utilizan en la actualidad.

Un factor principal en los primeros diseños de sistemas de bases de datos fue la eficiencia. Una de las formas comunes de hacer que los sistemas sean más eficientes era imponer una longitud fija para los registros de la base de datos, o al menos tener un número fijo de elementos por registro (columnas por fila). Esto esencialmente evita el problema del grupo que se repite. Si es un programador en casi cualquier lenguaje de procedimiento, verá fácilmente que, en este caso, puede leer cada registro de una base de datos en una estructura C simple. La vida real rara vez es tan complaciente, por lo que debemos encontrar formas de lidiar con datos estructurados de manera inconveniente. Los diseñadores de sistemas de bases de datos hicieron esto mediante la introducción de diferentes tipos de bases de datos.

- Modelo de base de datos jerárquica.
- Modelo de base de datos de red.
- Modelo de base de datos relacional.

Los primeros dos escapan al alcance de este curso y requerirían probablemente cursos apartes especializados. Nuestro foco será, como hasta acá, el modelo relacional.

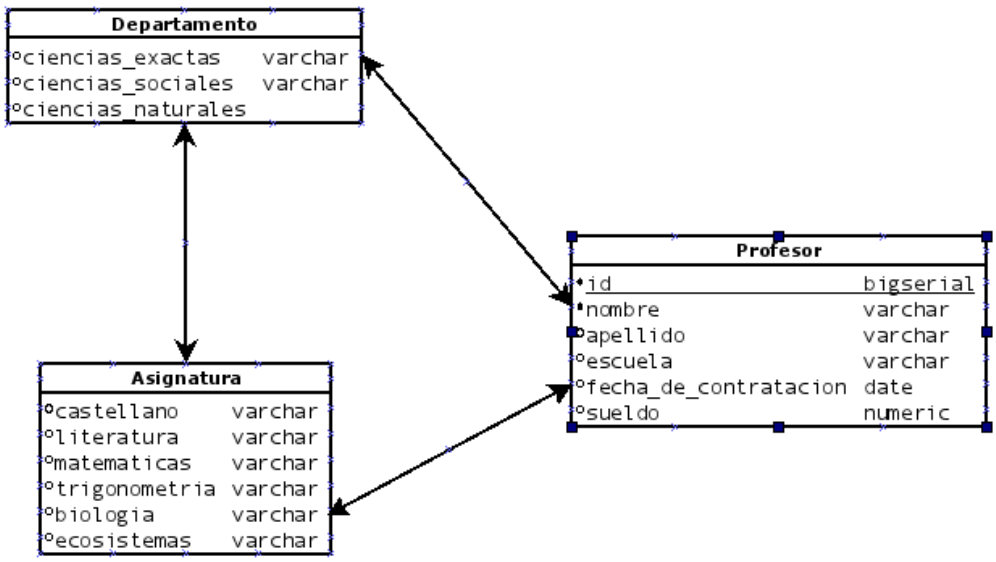
No es ningún misterio que una base de datos sea algo que almacena datos. Sin embargo, los modernos sistemas de administración de bases de datos relacionales

(RDBMS) de hoy, como MySQL, PostgreSQL, SQL Server, Oracle, DB2 y otros, han ampliado esta función básica al agregar la capacidad de almacenar y administrar datos relacionales. Este es un concepto que merece cierta atención.

Entonces, ¿qué significan los datos relacionales? Es fácil ver que cada dato escrito en una base de datos del mundo real está relacionado de alguna manera con información ya existente. Los productos están relacionados con categorías y departamentos; los pedidos están relacionados con productos y clientes, etc. Una base de datos relacional mantiene su información almacenada en tablas de datos, pero también es consciente de las relaciones entre las tablas.

Estas tablas relacionadas forman la base de datos relacional, que se convierte en un objeto con un significado propio, en lugar de ser simplemente un grupo de tablas de datos no relacionadas. Se dice que los datos se convierten en información solo cuando les damos significado, y establecer relaciones con otros datos es un medio ideal para hacerlo.

La Figura muestra una representación simple de tres tablas de datos.



Cuando se dice que dos tablas están relacionadas, esto significa más específicamente que los registros de esas tablas están relacionados. Entonces, si la tabla de productos está relacionada con la tabla de asignatura, esto se traduce en que cada registro de profesores está relacionado de alguna manera con uno de los registros de la tabla de asignatura.

Los diagramas como este se utilizan para decidir qué se debe almacenar en la base de datos. Una vez que sepa qué almacenar, el siguiente paso es decidir cómo se relacionan los datos enumerados, lo que conduce a la estructura física de la base de

datos. El diagrama de la figura es solo un boceto que sirve para ir estructurando un modelo (que por supuesto puede sufrir modificaciones).

Entonces, ahora que conoce los datos que desea almacenar, pensemos en cómo se relacionan las tres partes entre sí. Además de saber que los registros de dos tablas están relacionados de alguna manera, también necesita saber el tipo de relación entre ellos. Veamos ahora más de cerca las diferentes formas en que se pueden relacionar dos tablas.

Datos relacionales y relaciones de tablas

Para continuar explorando el mundo de las bases de datos relacionales, analicemos más a fondo las tres tablas lógicas que hemos estado viendo hasta ahora. Para hacer la vida más fácil, démosles nombres ahora: la tabla que contiene profesores es profesor (necesitamos renombrarla); la tabla que contiene departamentos es departamento; y el último es asignatura. ¡No hay sorpresas aquí! Afortunadamente, estas tablas implementan los tipos más comunes de relaciones que existen entre tablas, las relaciones de uno a muchos y de muchos a muchos, por lo que tiene la oportunidad de aprender sobre ellas.

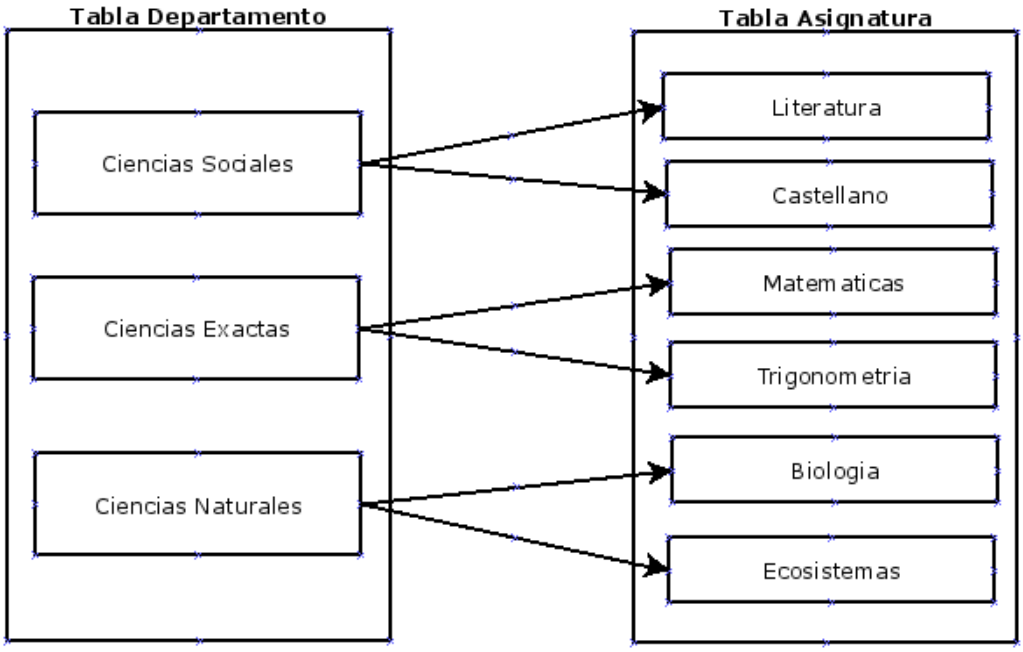
Renombremos nuestra tabla **profesores** a **profesor** para seguir las convenciones.

```
ALTER TABLE profesores RENAME TO profesor;
```

Nota: Existen algunas variaciones de estos dos tipos de relación que explicaremos, así como la relación uno a uno que es menos popular. En la relación uno a uno, cada fila de una tabla coincide exactamente con una fila de la otra. Por ejemplo, en una base de datos que permitiera que los pacientes fueran asignados a camas, ¡sería de esperar que hubiera una relación uno a uno entre los pacientes y las camas! Los sistemas de bases de datos no admiten la aplicación de este tipo de relación, porque tendría que agregar registros coincidentes en ambas tablas al mismo tiempo. Además, se pueden unir dos tablas con una relación uno a uno para formar una sola tabla.

Relaciones uno a muchos (one-to-many)

La relación de uno a varios ocurre cuando un registro de una tabla se puede asociar con varios registros de la tabla relacionada, pero no al revés. En nuestro caso, esto sucede para la relación departamento-asignatura. Un departamento específico puede contener cualquier número de categorías, pero cada categoría pertenece exactamente a un departamento. La figura siguiente representa mejor la relación de uno a varios entre departamentos y categorías.



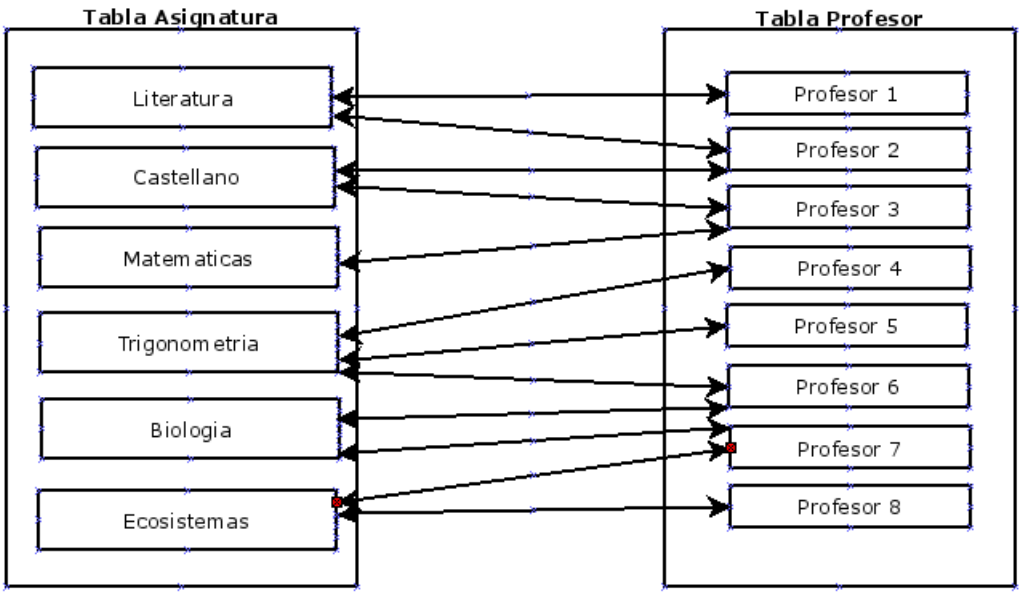
La relación de uno a muchos se implementa en la base de datos agregando una columna adicional en la tabla en el "lado muchos" de la relación, que hace referencia a la columna de ID de la tabla en el lado de la relación. En pocas palabras, en la tabla de asignaturas, tendrá una columna adicional (llamada **departmento_id)** que contendrá el ID del departamento al que pertenece la asignatura.

Relaciones de muchos a muchos (many-to-many)

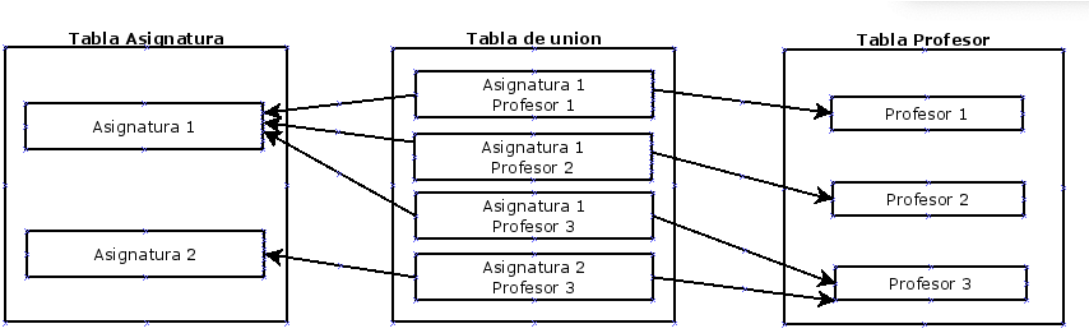
El otro tipo común de relación es la relación de muchos a muchos o varios a varios. Este tipo de relación se implementa cuando los registros en ambas tablas de la relación pueden tener múltiples registros coincidentes en la otra. En nuestro escenario, esto sucede para las tablas de profesores y asignaturas, porque sabemos que un profesor puede existir en más de una asignatura (un profesor con muchas asignaturas) y una asignatura puede tener más de un profesor (una asignatura con muchos profesoresductos).

Esto sucede porque decidimos antes que un profesor podría estar en más de una asignatura. Si un profesor solo pudiera pertenecer a una asignatura, tendría otra relación de uno a varios, como la que existe entre departamentos y asignaturas (donde una asignatura no puede pertenecer a más de un departamento).

Representaremos esta relación con una imagen, al igual que como lo hicimos para la relación anterior:



Aunque lógicamente la relación de muchos a muchos ocurre entre dos tablas, las bases de datos no tienen los medios para implementar físicamente este tipo de relación usando solo dos tablas, por lo que hacemos trampa agregando una tercera tabla a la mezcla. Esta tercera tabla, llamada tabla de unión (también conocida como tabla de vinculación o tabla asociada) y dos relaciones de uno a muchos ayudarán a lograr la relación de muchos a muchos. La tabla de unión se utiliza para asociar profesores y asignaturas, sin restricciones sobre cuántos profesores pueden existir para una asignatura o cuántas asignaturas se pueden agregar a un profesor. La figura muestra el papel de la tabla de unión.



Tenga en cuenta que cada registro de la tabla de unión vincula una categoría con un profesor. Puede tener tantos registros como desee en la tabla de unión, vinculando cualquier asignatura a cualquier profesor. La tabla de unión contiene dos campos, cada uno de los cuales hace referencia a la llave primaria de una de las dos tablas vinculadas. En nuestro caso, la tabla de unión contendrá dos campos: un campo **asignatura_id** y un campo **profesor_id**.

Cada registro de la tabla de unión constará de un par de ID de profesor y asignatura (**profesor_id, asignatura_id**), que se utilizará para asociar un profesor en particular

con una asignatura en particular. Al agregar más registros a la tabla **profesor_asignatura**, puede asociar un profesor con más asignaturas o una asignatura con más profesores, implementando eficazmente la relación de muchos a muchos.

Debido a que la relación de muchos a muchos se implementa utilizando una tercera tabla que hace la conexión entre las tablas vinculadas, no es necesario agregar campos adicionales a las tablas relacionadas de la forma en que agregamos el **departamento_id** a la tabla de asignatura para implementar la relación de uno a muchos.

No existe una convención de nomenclatura definitiva para usar en la tabla de unión. La mayoría de las veces está bien unir los nombres de las dos tablas vinculadas; en este caso, la tabla de unión se llama asignatura_profesor.

Aplicación de relaciones de tabla mediante llaves externas

Las relaciones entre tablas se pueden hacer cumplir físicamente en la base de datos utilizando restricciones **FOREIGN KEY**, o simplemente llaves externas o claves foráneas.

Ya aprendimos acerca de la restricción **PRIMARY KEY** o llave principal. Las llaves externas, por otro lado, ocurren entre dos tablas: la tabla en la que se define la llave externa (la tabla de referencia) y la tabla a la que hace referencia la clave externa (la tabla referenciada).

Una llave externa es una columna o combinación de columnas que se utiliza para imponer un vínculo entre los datos de dos tablas (generalmente representa una relación de uno a varios). Las llaves externas se utilizan como método para garantizar la integridad de los datos y para establecer una relación entre tablas.

Para hacer cumplir la integridad de la base de datos, las llaves externas, aplican ciertas restricciones. A diferencia de las restricciones **PRIMARY KEY** y **UNIQUE** que aplican restricciones a una sola tabla, la restricción **FOREIGN KEY** aplica restricciones tanto en las tablas de referencia como en las referenciadas. Por ejemplo, si aplica la relación de uno a varios entre las tablas de departamento y de categoría mediante una restricción **FOREIGN KEY**, la base de datos incluirá esta relación como parte de su integridad. No le permitirá agregar una categoría a un departamento inexistente, ni le permitirá eliminar un departamento si hay categorías que pertenecen a él.

Creación y llenado de nuevas tablas de datos

Ya renombramos la tabla profesores como **profesor** para seguir las convenciones (no usar plurales). Es hora de completar el resto de nuestras tablas.

```
CREATE TABLE departamento (
  departamento_idBIGSERIAL,
  nombre        VARCHAR(100),
  descripcion   VARCHAR(1000),
  PRIMARY KEY (departamento_id)
);

INSERT INTO departamento (departamento_id, nombre, descripcion)
  VALUES
(1, 'Ciencias Sociales', 'Ramas de la ciencia relacionadas con la
sociedad y el comportamiento humano.'),
(2, 'Ciencias Exactas', 'Disciplinas que se basan en la observación y
experimentación para crear conocimientos y cuyos contenidos pueden
sistematizarse a partir del lenguaje matemático.'),
(3, 'Ciencias Naturales', 'Ciencias que tienen por objeto el estudio
de la naturaleza, siguiendo la modalidad del método científico conocida
como método empírico-analítico.');


CREATE TABLE asignatura
(
       asignatura_id      BIGSERIAL NOT NULL,
       departamento_id    BIGSERIAL NOT NULL,
       nombre             VARCHAR(100) NOT NULL,
       descripcion        VARCHAR(1000),
PRIMARY KEY (asignatura_id)
);
```

Nota: PostgreSQL tiene los tipos de datos smallserial, serial y bigserial; estos no son tipos verdaderos, sino simplemente una conveniencia de notación para crear columnas de identificador único. Son similares a la propiedad **AUTO_INCREMENT** admitida por algunas otras bases de datos.

Si por alguna razón desea borrar alguna tabla, por ejemplo, con el fin de crearla de nuevo para experimentar, puede ejecutar

```
DROP TABLE <nombre de la tabla>;
```

Ahora llenaremos algunas filas de nuestra tabla **asignatura**

```
INSERT INTO asignatura (
      asignatura_id, departamento_id, nombre, descripcion)
VALUES
(1, 1, 'Literatura', 'Arte de la expresión verbal'),
(2, 1, 'Castellano', 'Lengua romance procedente del latín hablado'),
(3, 2, 'Matemáticas', 'Ciencia formal que, partiendo de axiomas y siguiendo
el razonamiento lógico, estudia las propiedades y relaciones entre entidades
abstractas como números, figuras geométricas, iconos, glifos, o símbolos en
general'),
(4, 2, 'Trigonometría', 'Rama de la matemática, cuyo significado etimológico
es la medición de los triángulos'),
(5, 3, 'Biología', 'Rama de la ciencia que estudia los procesos naturales de
los organismos vivos, considerando su anatomía, fisiología, evolución,
desarrollo, distribución y relaciones'),
(6, 3, 'Ecosistema', 'Sistema biológico constituido por una comunidad de
organismos vivos (biocenosis) y el medio físico donde se relacionan
(biotopo)');
```

```
CREATE TABLE profesor_asignatura (
  profesor_id INT NOT NULL,
  asignatura_id INT NOT NULL,
  PRIMARY KEY (profesor_id, asignatura_id)
);


INSERT INTO profesor_asignatura (profesor_id, asignatura_id) VALUES
(1, 1), (1, 2), (2, 3), (2, 4), (3, 5), (3, 6), (4, 6), (5, 4), (6,
1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (7, 3), (7, 4), (8, 1);
```

Ahora realicemos cambios en nuestra tabla **profesor** para continuar con las convenciones de nombres que hemos adoptado.

```
ALTER TABLE profesor RENAME COLUMN id TO profesor_id;
```

Agregamos la llave primaria

```
ALTER TABLE profesor ADD PRIMARY KEY (profesor_id);
```

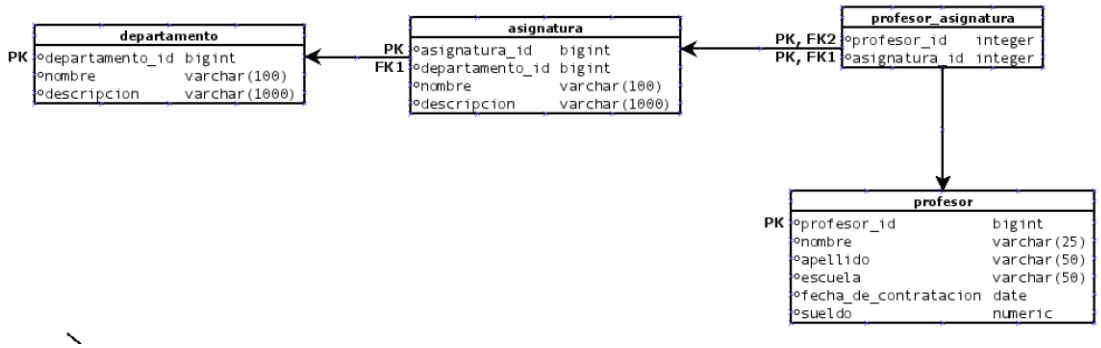
4.2.3.2. Consultas de selección con tablas relacionadas

Ahora tenemos un esquema de tablas que podemos relacionar y consultar según nuestras necesidades, por ejemplo, si queremos saber a qué departamento pertenece la asignatura 'Castellano'.

```
SELECT departamento.nombre FROM departamento
INNER JOIN asignatura
       ON asignatura.departamento_id = departamento.departamento_id
WHERE asignatura.nombre = 'Castellano';
```

Esta consulta entrega

```
     nombre
-------------------
  Ciencias Sociales
(1 row)
```

O si queremos saber en qué departamento(s) trabaja el profesor Pablo Rojas

```
SELECT d.nombre
        FROM departamento d
        INNER JOIN asignatura a
                ON a.departamento_id = d.departamento_id
    INNER JOIN profesor_asignatura pa
        ON pa.asignatura_id = a.asignatura_id
    INNER JOIN profesor p
        ON p.profesor_id = pa.profesor_id
WHERE p.nombre = 'Pablo' AND p.apellido = 'Rojas';
```

Resultado

```
      nombre
--------------------
 Ciencias Naturales
(1 row)
```

Puede comprobar el resultado siguiendo las queries y las tablas de forma visual/manual.

4.2.3.3. Integridad referencial

La integridad referencial se refiere a la precisión y consistencia de los datos dentro de una relación.

En las relaciones, los datos están vinculados entre dos o más tablas. Esto se logra haciendo que la llave externa (en la tabla asociada) haga referencia a un valor de llave principal (en la tabla principal). Debido a esto, debemos asegurarnos de que los datos de ambos lados de la relación permanezcan intactos.

Por lo tanto, la integridad referencial requiere que, siempre que se use un valor de llave externa, debe hacer referencia a una llave primaria válida existente en la tabla principal.

En nuestras tablas no hemos observado que, por ejemplo, exista una asignatura que haga referencia al **departamento_id = 4**, solo tenemos 3 departamentos Ciencias Sociales, Ciencias Exactas y Ciencias Naturales. Referenciar un departamento con **id = 4** generaría un record huérfano.

La falta de integridad referencial en una base de datos puede llevar a que se devuelvan datos incompletos, normalmente sin indicación de error. Esto podría resultar en la "pérdida" de registros en la base de datos, porque nunca se devuelven en consultas o informes.

También podría dar lugar a que aparezcan resultados extraños en informes (como productos sin una empresa asociada). O peor aún, podría resultar en que los clientes no reciban los productos por los que pagaron o que un paciente del hospital no reciba el tratamiento correcto, o un equipo de socorro en casos de desastre que no reciba los suministros o la información correctos.

4.2.3.4. Querys anidadas

Una subconsulta a veces se anida dentro de otra consulta. Normalmente, se utiliza para un cálculo o una prueba lógica que proporciona un valor o un conjunto de datos que se pasarán a la parte principal de la consulta. Su sintaxis no es inusual: simplemente encerramos la subconsulta entre paréntesis y la usamos donde sea necesario. Por ejemplo, podemos escribir una subconsulta que devuelve varias filas y trata los resultados como una tabla en la cláusula **FROM** de la consulta principal. O podemos crear una subconsulta escalar que devuelva un solo valor y usarlo como parte de una expresión para filtrar filas a través de las cláusulas **WHERE, IN** y **HAVING**. Esos son los usos más comunes de las subconsultas.

Como ejemplo una vez más buscaremos en qué departamento(s) trabaja el profesor Pablo Rojas

```
SELECT d.nombre
    FROM departamento d
    WHERE d.departamento_id = (
       SELECT departamento_id FROM asignatura a
              WHERE a.asignatura_id = (
              SELECT asignatura_id FROM profesor_asignatura pa
                     WHERE pa.profesor_id = (
                     SELECT p.profesor_id FROM profesor p
                            WHERE p.nombre = 'Pablo' AND p.apellido = 'Rojas'
                )
          )
  );
```

Resultado:

```
      nombre
--------------------
  Ciencias Naturales
(1 row)
```

4.2.3.5. Queries con distintos tipos de JOIN (INNER, LEFT, OUTER)

En lo que se refiere a **JOIN(s)** Hasta ahora hemos utilizado **INNER JOIN** en nuestras consultas. Ahora veremos las distintas opciones de esta cláusula.

Hay más de una forma de unir tablas en SQL, y el tipo de unión que usará depende de cómo desee recuperar los datos. La siguiente lista describe los diferentes tipos de combinaciones. Al revisar cada uno, es útil pensar en dos tablas una al lado de la otra, una a la izquierda de la palabra clave **JOIN** y la otra a la derecha. A continuación de la lista, se muestra un ejemplo basado en datos de cada combinación:

- **JOIN** Devuelve filas de ambas tablas donde se encuentran valores coincidentes en las columnas unidas de ambas tablas. La sintaxis alternativa es **INNER JOIN**.
- **LEFT JOIN** Devuelve todas las filas de la tabla de la izquierda más las filas que coinciden con los valores de la columna unida de la tabla de la derecha. Cuando una fila de la tabla de la izquierda no tiene una coincidencia en la tabla de la derecha, el resultado no muestra valores de la tabla de la derecha.
- **RIGHT JOIN** Devuelve todas las filas de la tabla derecha más las filas que coinciden con los valores clave en la columna clave de la tabla izquierda. Cuando una fila de la tabla de la derecha no tiene una coincidencia en la tabla de la izquierda, el resultado no muestra valores de la tabla de la izquierda.
- **FULL OUTER JOIN** Devuelve cada fila de ambas tablas y coincide con las filas; luego une las filas donde coinciden los valores de las columnas unidas. Si no hay ninguna coincidencia para un valor en la tabla izquierda o derecha, el resultado de la consulta contiene una fila vacía para la otra tabla.

Para explicar de forma más clara vamos a "ensuciar" un poco nuestra base de datos e insertar una fila adicional y sin relaciones en la tabla departamento y otra en la tabla asignatura (violando la integridad referencial solo para ejemplificar).

```
INSERT INTO departamento (departamento_id, nombre, descripcion) VALUES
(4, 'Ciencias Especiales', 'Ramas de la ciencia que son especiales.');

INSERT INTO asignatura (asignatura_id, departamento_id, nombre,
descripcion) VALUES (7, 10, 'Especial', 'Asignatura Especial');
```

Usaremos un par de tablas de nuestra base de datos para demostrar alternativas

Hasta ahora hemos usado **JOIN (INNER JOIN)**; este tipo puede ser pensado como "Intersección" de ambas tablas. Ejemplo para recordar que asignaturas corresponden a que departamento podemos ejecutar:

```
SELECT d.nombre, a.nombre FROM
departamento d JOIN asignatura a
ON d.departamento_id = a.departamento_id;
```

```
mibasededatos-# ON d.departamento_id = a.departamento_id;

    nombre          |    nombre
--------------------+---------------
Ciencias Sociales   | Literatura
Ciencias Sociales   | Castellano
Ciencias Exactas    | Matemáticas
Ciencias Exactas    | Trigonometría
Ciencias Naturales  | Biología
Ciencias Naturales  | Ecosistema
(6 rows)
```

- **LEFT JOIN y RIGHT JOIN**: A diferencia de **JOIN**, las palabras clave **LEFT JOIN** y **RIGHT JOIN** devuelven todas las filas de una tabla y muestran filas en blanco de la otra tabla si no se encuentran valores coincidentes en las columnas unidas. Primero veamos **LEFT JOIN** en acción, esta puede ser pensada como la "Intersección más lo que está a la izquierda"

```
SELECT d.nombre, a.nombre FROM
departamento d LEFT JOIN asignatura a
ON d.departamento_id = a.departamento_id;
```

Resultado:

```
      nombre        |    nombre
--------------------+---------------
Ciencias Sociales   | Literatura
Ciencias Sociales   | Castellano
Ciencias Exactas    | Matemáticas
Ciencias Exactas    | Trigonometría
Ciencias Naturales  | Biología
Ciencias Naturales  | Ecosistema
Ciencias Especiales|
(7 rows)
```

```
- Vemos que el departamento dummy Ciencias Especiales tambien aparece
en la consulta.
```

- **RIGHT JOIN** es similar solo que este puede pensado como la "Intersección más lo que está a la derecha"

```
SELECT d.nombre, a.nombre FROM
departamento d RIGHT JOIN asignatura a
ON d.departamento_id = a.departamento_id;
```

Obtenemos

```
    nombre          |    nombre
--------------------+---------------
Ciencias Sociales   | Literatura
Ciencias Sociales   | Castellano
Ciencias Exactas    | Matemáticas
Ciencias Exactas    | Trigonometría
Ciencias Naturales  | Biología
Ciencias Naturales  | Ecosistema
                    | Especial
Ciencias Especiales|
(8 rows)
```

Como puede verse esto incluye "todo" de ambas tablas.

4.2.4 Referencias

[1] Rahul Batra, SQL Primer An Accelerated Introduction to SQL Basics, 2018.

[2] PostgreSQL Tutorial
https://www.tutorialspoint.com/postgresql/

[3] Manpreet Kaur, PostgreSQL Development Essentials, 2016.

[4] PostgreSQL Foreign Key
https://www.postgresqltutorial.com/postgresql-foreign-key/

[5] PostgreSQL Documentation
https://www.postgresql.org/docs/12/index.html

[6] Allen G. Taylor, SQL, 2019.