Taller de Química Computacional Aplicada: Día 2

Rony J. Letona

9 de octubre de 2014

1. Ejercicios con una Base de Datos Relacional - SQLite

Una tabla es, generalmente, la mejor forma de ordenar datos cuando queremos almacenarlos o representarlos. Sin embargo, a todos los que hemos trabajado con muchos datos, nos suele suceder que queremos aislarlos o filtrarlos de alguna manera para analizar alguna tendencia particular. Microsoft Excel nos deja ordenar los datos, pero no nos deja aislarlos, ni mucho menos filtrarlos en base a alguna condición. Y hay otro problema aún! Una tabla de Excel o de Calc tiene límites bajos de almacenamiento de datos. Qué hacer ahora? Allí entramos al reino de las bases de datos.

Una base de datos (o DB por sus siglas en inglés) es simplemente un lugar donde meter información en gran cantidad. Existen de varios tipos, pero en particular hablaremos de las bases de datos relacionales. En estas, los datos se almacenan de forma ordenada en tablas gigantes. Para accesar los datos se utiliza un lenguaje muy particular llamado lenguaje de consulta estructurado, o SQL por sus siglas en inglés. Este permite hacer una serie de operaciones que van desde insertar o eliminar un dato en específico, hasta filtrar todos los datos almacenados y calcular propiedades de los mismos. Guardar estos datos para representarlos en gráficas o para posteriores análisis también se puede hacer, logrando así que muchos datos no se queden en datos, sino que se pueda hacer inferencias sobre ellos.

Para comprender mejor cómo es que funciona esto debemos de considerar que una DB no es solo el lugar de almacenamiento, sino que también hay un *administrador* de la DB que es el programa que interpreta el SQL y hace las consultas y los cálculos. Por eso es que generalmente las DBs no son un archivo con toda la información nada más. Una DB es más compleja que eso. Sin embargo, por diferentes razones y circunstancias existen ahora varias formas de bases de datos que pueden almacenarse en un archivo, que pueden accesarse de otras formas, etc. Existen 6 "sabores" de DBs con las que puede que nos topemos seguido: Microsoft Access, LibreOffice Base, Microsoft SQL Server, MySQL, PostgreSQL y SQLite. Para nuestro caso en particular que solemos llevar y traer datos, que no tenemos conecciones del todo estables de internet, que buscamos universalidad evitando software caro y que deseamos algo prequeño, rápido de usar y que no involucre tener que montar todo un sistema para ello, la portabilidad, el costo y la sencillez son importantes. Por eso haremos uso de SQLite.

1.1. Seleccionando Datos

Para explicar cómo funcionan las bases de datos se comienza generalmente por el final. Lo primero en describirse es cómo se ven los datos ya ingresados en una y luego se pasa a explicar cómo se crean, eliminan, cómo se le agregan datos o se le quitan, etc. Para entender el cómo y el por qué de esto, se propone una analogía:

Cuando vamos a un laboratorio, generalmente hay una persona que es la encargada de los reactivos y el equipo (o en cargado de bodega). A esta persona debemos decirle lo que necesitamos para que nos lo prepare y lo tenga listo a la hora de realizar una práctica o un procedimiento. Si repasamos detenidamente lo que tenemos qué hacer para ello, hallaremos las similitudes con una DB.

1. Llegamos al laboratorio.
2. Nos presentamos con el encargado del laboratorio.
3. Le pedimos los listados de equipo o reactivos si no sabemos cuáles son todavía.
4. Seleccionamos lo que deseamos de los listados.
5. Recibimos el equipo y los reactivos como el resultado de nuestra consulta.

En el caso de una DB, los pasos para obtener información son muy similares.

1. Localizamos la base de datos.
2. Iniciamos con el administrador.
3. Solicitamos las tablas en la base de datos, si no las conocemos todavía.
4. Seleccionamos los datos que deseamos de una tabla.
5. El administrador nos despliega el resultado de nuestra consulta.

Como podemos ver, los pasos son similares. Otra cosa que debemos tomar en cuenta es que a nosotros no se nos dejaba usar reactivos o equipo (más adelante pedir reactivos o equipo) hasta que ya hubiéramos pasado algunos cursos, tenido un poco de experiencia y estado concientes de lo que estábamos haciendo. También debemos recordar que aunque ya se nos deje solicitar estas cosas, siempre lo debemos hacer a través del encargado del laboratorio o bodega. Con esto en mente ya procederemos a ir paso a paso aprendiendo cómo se usa una DB.

Para comenzar vamos a vamos a abrir una línea de comando y en ella iremos hasta nuestro directorio de documentos. Allí ingresaremos el comando `sqlite3 monitoreo.db` Aquí vamos a ver algunas líneas que describen la versión de SQLite que tenemos y cómo obtener ayuda. Considerando que ya estamos en una DB, piensa por un momento: qué es lo que nos gustaría saber de esta base de datos?

Si "el contenido" fue lo primero que apareció en nuestra mente, nos estamos adelantando un poco y estamos perdiendo la noción de la idea de una base de datos. Por qué será que ver *todo* el contenido es una mala idea? (Pensemos en bases de datos grandes.) Antes de apresurarnos a ver qué hay dentro de la DB, debemos entender cómo se divide; qué listas de reactivos o equipos hay. Para ello vamos a ingresar el comando `.tables` Qué te mostró el administrador? Qué crees que son esos nombres? Piénsalo y discútelo un momento.

Vamos a ver el contenido de cada una de las tablas en la base de datos, pero antes, vamos a configurar a SQLite para que los resultados nos sean fáciles de interpretar. Para ello vamos a ingresar los siguientes dos comandos:

```
.headers ON
.mode columns
```

Estos nos permitirán visualizar los datos como columnas ordenadas y con sus respectivos encabezados. Ahora sí procederemos a ver qué hay en las tablas.

1.1.1. Una Consulta Sencilla

La base de datos con la que estamos trabajando contiene datos *falsos* del laboratorio de Monitoreo del Aire. Después de haber visto qué tablas había en ellos y sabiendo cómo es que funcionan los laboratorios, es seguro pensar que las tablas de los lugares y las personas van a ser más pequeñas que las de los muestreos. Por ello, para nuestra primera consulta vamos a ver quiénes han hecho análisis en el laboratorio de Monitoreo del Aire.

Comencemos! La idea será seleccionar toda la tabla para conocer qué *campos* contiene esta, y además, qué *entradas* hay almacenadas. Recapitulando: vamos a seleccionar toda la información de la tabla de **Personas**. El comando para esto es el siguiente: `SELECT * FROM Personas;` Analicemos un momento el comando e intentemos entender qué es lo que hace cada parte. Además, anotemos los encabezados de cada columna; nos van a servir después.

Hay dos partes que merecen atención de nuestra primera consulta. La primera, y muy importante, es que **toda** consulta hecha en SQL debe terminar con un punto y coma. La segunda cosa a notar es el asterisco. Como ya habíamos visto ayer en la línea de comando, el asterisco se utiliza como un comodín; un caracter que puede representar lo que sea. En este caso eso nos permite extraer toda la información de la tabla. Pero, qué puede usarse si no es un comodín? Para ello vamos a volver a hacer la consulta, haciendo una variación: `SELECT nombre FROM Personas;` Observa qué fue lo que pasó e intenta explicar por qué ocurrió el cambio.

Ahora que ya vimos cómo realizar consultas de un *campo* en particular, vamos a intentar realizar una con varios campos. Para ello vamos a intentar incluir varios campos dentro de la consulta y sin un orden en particular. `SELECT apellido, pid, nombre FROM Personas;` Como vemos, podemos accesar los campos que querramos y en el orden que querramos. Esto nos será útil cuando solo necesitemos de alguna información de la base de datos, y no todo el contenido de las tablas.

Como ejercicio rápido, revisa el contenido de las tablas **Lugares** y **Muestreo**; nos va a servir después. Ten cuidado con la tabla **Muestreo**: serán bastantes resultados. Toma nota de los campos de cada una.

Hemos visto cómo controlar los campos en cada tabla. Después vamos a ver cómo visualizar solo algunas entradas dependiendo de alguna condición. Por ahora, es importante que sepamos que casi siempre existe algún número, código o identificador que servirá para representar a cada entrada. No existen dos identificadores iguales, y generalmente a este campo se le llama *llave primaria*. En los ejemplos anteriores, este ha sido el campo llamado `pid` .

1.1.2. Ordenando y Filtrando *donde* se cumpla una Condición

Una de las tareas complicadas al trabajar con hojas de cálculo (como en Microsoft Excel) es el filtrar los datos para poder visualizar alguna tendencia o patrón en particular. Utilizando SQL, esto se vuelve una tarea sencilla. Es común, por ejemplo, que necesitemos ver la cantidad de algún compuesto en un lugar en particular a lo largo del tiempo, pero en la base de datos tenemos todo junto. Para ello solo tenemos que aprender a filtrar los resultados que estábamos obteniendo antes.

Vamos a consultar por todas las personas con apellido 'García' que se encuentren en la tabla **Personas**. Refrescando un poco, la consulta que teníamos antes se veía así: `SELECT * FROM Personas;` Ahora debemos seleccionar *todas* las entradas de la tabla **Personas**, en *donde* el apellido sea igual a García. Para ello, la consulta se verá así: `SELECT * FROM Personas WHERE apellido='Garcia';`

El resultado se explica por sí solo. Intentemos ahora consultar a la base de datos, a modo de que se distinga si hay entradas con un campo repetido y eliminar los duplicados. Para ello, intentemos con la siguiente consulta: `SELECT DISTINCT apellido FROM Personas;` El resultado nos revela solo las entradas *distintas*! Eliminar duplicados fue fácil. Intentemos ordenar los resultados basándonos en el nombre (por orden alfabético, claro). La consulta ha de verse así: `SELECT * FROM Personas ORDER BY nombre;` Observemos los resultados y notemos cómo aparece la columna de nombres y la de identificadores. Ahora, para terminar con la parte de ordenado, vamos a ordenar los apellidos de esta tabla de forma descendente: `SELECT * FROM Personas ORDER BY apellido DESC;`

Eso fue sencillo y nos dio los resultados esperados. Vamos a intentar algo un poco más complicado. El trébol en la ciudad de guatemala tiene las coordenadas: 14,613309^oN y 90,535149^oO. Esto se traduce en las cifras: 14,613309

y −90,535149. Cuando la longitud es mayor, nos movemos más al este (a la derecha en el mapa). Cuando la latitud es mayor, nos movemos más al norte (hacia arriba en el mapa). Vamos a intentar hallar todos aquellos lugares en el extremo superior derecho con respecto al trébol (NE). Para ello necesitmos que la longitud y la latitud sean mayores a las coordenadas el trébol. En particular, queremos los nombres de los lugares. Intentemos hacer esto con una nueva consulta: `SELECT nombre FROM Lugares WHERE (longitud>-90.535149) AND (latitud>14.613309);`

Esto nos debió haber mostrado solo un resultado: MUSAC. Tengamos cuidado con los signos de mayor y menor! Ahora, como ejercicio, intentemos consultar por las otras 3 direcciones: NO, SO, SE. Para terminar con esta sección, vamos a realizar una consulta sencilla de la tabla **Lugares**. Vamos a extraer todos los nombres de lugares que comiencen con *i*. Para ello vamos a hacer la siguiente consulta:

```
SELECT nombre FROM Lugares WHERE (nombre LIKE 'I%');
```

Con esto último ya podemos hallar cosas semejantes. Si nos damos cuenta atentamente, el comodín en este caso no es el asterisco `*` como normalmente había sido, sino que ahora es el signo de porcentaje `%`

1.1.3. Juntando Tablas

Esta es una de las partes más importantes de las bases de datos y es lo que les da **toda** la magia. Por ahora solo habíamos notado que en la tabla de **Muestreo** teníamos un campo para personas y uno para lugares, pero estaban llenos de números. Si habíamos puesto atención a los detalles, nos dimos cuenta de que esos números son realmente los identificadores *pid* y *lid* de las tablas **Personas** y **Lugares** respectivamente. Y es que de aquí es de donde viene el nombre de las bases de datos relacionales: relacionan los datos entre tablas para no tener que guardar cierto dato grande repetidas veces en una sola tabla. Y ahora vamos a ver cómo es que estas se relacionan realmente.

Vamos a intentar mostrar a las personas que trabajaron en análisis en el laboratorio de Monitoreo del Aire durante el mes de enero del año 2013. Para ello deberemos filtrar por fecha, y juntar a la tabla de **Muestreo** con la de **Personas**. Hagamos la consulta:

```
SELECT Muestreo.mid, Personas.nombre, Muestreo.fecha FROM Muestreo JOIN Personas
ON Muestreo.persona=Personas.pid WHERE (Muestreo.fecha>='2013-01-01') AND
(Muestreo.fecha<='2013-01-31');
```

La consulta se ve algo monstruosa, pero la verdad solo son muchos pedacitos unidos en una gran línea. Vamos paso a paso: Algo nuevo es que estamos llamando a cada campo con su nombre y apellido. Ya no estamos solo pidiendo `fecha` por ejemplo, sino que estamos pidiendo `Muestreo.fecha` Esto se debe a que estamos trabajando con *dos* tablas en vez de una y debemos ser claros con los campos de cada una (habrán veces en donde dos tablas tienen campos con los mismos nombres y la forma de diferenciarlos es con el nombre de la tabla). Luego, utilizamos una nueva palabra clave: `JOIN` Esta es la que va a traer la *otra* tabla, en este caso la de **Personas** y la deja a disposición nuestra. Finalmente vemos que hay una última palabrita nueva en nuestra consulta: `ON` Esta señala en dónde estamos juntando las dos tablas, o en otras palabras, en dónde existe la relación entre las dos.

Recapitulando, la consulta que hicimos dice así: seleccionar el identificador de la tabla de Muestreo, el nombre de la tabla de Personas y la fecha de la tabla de Muestreo desde la tabla de Muestreo unida a la tabla de Personas entre el campo personas de Muestreo y el identificador de Personas, cuando la fecha sea mayor o igual al primero de enero del 2013 y menor o igual al 30 de enero del 2013. Ahora nos parece que verla en SQL es más fácil. Es de agradecer que generalmente esto solo se programa en un lenguaje de programación *una vez* y luego ya no se tiene que volver a ver. Sin embargo, la idea se mantiene. Y lo importante es que las bases de datos relacionales pueden crear este tipo de relaciones entre tablas. Ahora pasaremos a ver qué se puede hacer con los resultados que hemos estado obteniendo de tantas maneras.

1.2. Operaciones Matemáticas

Además de observar datos ordenados de diferentes maneras y filtrados, muchas veces queremos extraer información colectiva de todos los datos o ver los datos de diferente manera. Para esto nos hace falta poder realizar operaciones a los datos y así obtener resultados diferentes. Para comenzar debemos de tomar en cuenta ciertas cosas. Las operaciones básicas son suma, resta multiplicación, división y módulo. Los signos que las representan en SQL son `+` , `-` , `*` , `/` y `%`

Por si nunca habíamos escuchado de la operación módulo, la aclaramos rápidamente: es la operación que nos devuelve el residuo de una división (e.g. `5%2 = 1` , `7%5 = 2` y `7%4 = 3`). A pesar de que ahorita no le veamos mucho uso, esta operación hace de la programación algo mucho más cómodo en muchos casos.

1.2.1. Alterando el Resultado

Vamos a trabajar finalmente con la tabla **Muestreo**. En ella hay muchos datos (falsos) sobre mediciones de varios gases y material particulado en el aire. Resulta que los datos de PM10 están dados en gramos y para un estudio en Estados Unidos los necesitan en onzas. También necesitan que estos datos sean solo los del año 2012. Para ello necesitamos pasar todos los datos de PM10 a onzas y filtrar. Cómo hacemos eso? Pues es sencillo. El factor de conversión es el siguiente: $1g = 0{,}035274oz$, así que lo que debemos hacer es multiplicar todos nuestros datos de PM10 por ese factor. Intentémoslo.

Lo primero que debemos pensar es en hacer una consulta a la tabla de **Muestreo**. De ella vamos a tomar solo los datos de CO_2, estos los vamos a multiplicar por nuestro factor y luego vamos a filtrar las fechas. Veamos la consulta:
`SELECT PM10 * 0.035274, fecha FROM Muestreo WHERE (fecha>'2012-01-01') AND (fecha<'2012-12-31');`
Los resultados son exactamente lo que buscábamos! Todos los resultados transformados a otra unidad. Y así se podría hacer con cualquier columna o cualquier operación. Todos nuestros datos los podemos representar de maneras distintas. Pero, será que podemos utilizar esto también dentro de las condiciones?

Como un ejercicio pequeño, intenta descifrar qué es lo que hace las siguientes consultas:

```
SELECT * FROM Personas WHERE pid%2=0;
SELECT round(pH,4), fecha FROM Muestreo WHERE round(pH,1)>6.5;
```

1.2.2. Obteniendo Descriptores

Cuando ya logramos manipular nuestros datos, solo nos queda una cosa por hacer. Comenzar a calcular valores nuevos a partir de ellos. En estadística, estos se llaman descriptores. SQLite permite calcular algunos descriptores sencillos, como los que vamos a ver a continuación. Para hacer análisis más extensos de la información ya se utilizan Mapeos Objeto-Relacional (u ORMs por sus siglas en inglés) y paquetes estadísticos, los cuales tocaremos brevemente más adelante. Por ahora, comencemos con lo básico.

Los descriptores que podemos calcular con facilidad en SQLite son la media, el mínimo, el máximo, contar las entradas y la suma o total de todas. Veremos que una vez aprendemos una, las demás son sencillas. Existen algunos métodos sucios de calcular la varianza haciendo uso de estas funciones y operaciones matemáticas, pero siempre es más conveniente utilizar un ORM en vez de SQL para esto.

De los datos de CO_2 en la tabla de **Muestreo** vamos a calcular todos estos descriptores para el año 2013. Para ello vamos a realizar las siguientes consultas:

```
SELECT min(CO2) FROM Muestreo WHERE (fecha>='2013-01-01') AND (fecha<='2013-12-31');
SELECT max(CO2) FROM Muestreo WHERE (fecha>='2013-01-01') AND (fecha<='2013-12-31');
SELECT count(CO2) FROM Muestreo WHERE (fecha>='2013-01-01') AND (fecha<='2013-12-31');
SELECT avg(CO2) FROM Muestreo WHERE (fecha>='2013-01-01') AND (fecha<='2013-12-31');
SELECT sum(CO2) FROM Muestreo WHERE (fecha>='2013-01-01') AND (fecha<='2013-12-31');
```

1.3. Últimas Consultas

El último ejercicio que haremos con respecto a lo que hemos visto hasta ahora será una extraña combinación de todo. La idea será obtener como resultado el promedio de los datos de SO_2 del INCAP para todo el año 2012. Y luego, siguiendo la misma idea, desplegar los datos de NO_2, la fecha y los analistas para MUSAC en el año 2013. Antes de comenzar a probar, pensemos un momento lo que queremos hacer y propongamos la consulta sin revisar cómo se hace.

1.3.1. Primera Consulta Final

Queremos una consulta de los muestreos: `SELECT * FROM Muestreo;`
En particular, queremos el promedio de los datos de SO_2: `SELECT avg(SO2) FROM Muestreo;`
Pero los deseamos solo de un lugar en particular y ese lugar viene de otra tabla:

```
SELECT avg(Muestreo.SO2) FROM Muestreo JOIN Lugares ON Muestreo.lugar=Lugares.lid;
```

En particular, lo queremos del INCAP:

```
SELECT avg(Muestreo.SO2) FROM Muestreo JOIN Lugares ON Muestreo.lugar=Lugares.lid WHERE
(Lugares.nombre='INCAP');
```

Y el resultado lo deseamos entre el primero de enero del 2012 y el 31 de diciembre del mismo año:

```
SELECT avg(Muestreo.SO2) FROM Muestreo JOIN Lugares ON Muestreo.lugar=Lugares.lid WHERE
(Lugares.nombre='INCAP') AND (Muestreo.fecha>='2012-01-01') AND (Muestreo.fecha<='2012-12-31');
```

Y así logramos el resultado de la primera consulta que queríamos realizar. Ahora intentémoslo con la segunda.

1.3.2. Segunda Consulta Final

Queremos una consulta de los muestreos nuevamente: `SELECT * FROM Muestreo;`
En particular, queremos los datos de NO_2 y la fecha: `SELECT NO2, fecha FROM Muestreo;`
Pero los deseamos solo de un lugar en particular y ese lugar viene de otra tabla:

```
SELECT Muestreo.NO2, Muestreo.fecha FROM Muestreo JOIN Lugares ON Muestreo.lugar=Lugares.lid;
```

En particular, lo queremos del MUSAC:

```
SELECT Muestreo.NO2, Muestreo.fecha FROM Muestreo JOIN Lugares ON Muestreo.lugar=Lugares.lid
WHERE (Lugares.nombre='MUSAC');
```

También deseamos los nombres de los analistas, y para ello necesitamos a la tercera tabla:

```
SELECT Muestreo.NO2, Muestreo.fecha FROM Muestreo JOIN Lugares ON Muestreo.lugar=Lugares.lid
JOIN Personas ON Muestreo.persona=Personas.pid WHERE (Lugares.nombre='MUSAC');
```

Los nombres deben de aparecer como resultado de la consulta, así que debemos incluirlos:

```
SELECT Muestreo.NO2, Muestreo.fecha, Personas.nombre, Personas.apellido FROM Muestreo JOIN
Lugares ON Muestreo.lugar=Lugares.lid JOIN Personas ON Muestreo.persona=Personas.pid WHERE
(Lugares.nombre='MUSAC');
```

Y el resultado lo deseamos entre el primero de enero del 2013 y el 31 de diciembre del mismo año:

```
SELECT Muestreo.NO2, Muestreo.fecha, Personas.nombre, Personas.apellido FROM Muestreo JOIN
Lugares ON Muestreo.lugar=Lugares.lid JOIN Personas ON Muestreo.persona=Personas.pid WHERE
(Lugares.nombre='MUSAC') AND (Muestreo.fecha>='2013-01-01') AND (Muestreo.fecha<='2013-12-31');
```

Por estética decidimos incluir el lugar como una de los columnas a desplegarse y ordenar los resultados por apellidos de manera descendente.

```
SELECT Lugares.nombre, Muestreo.NO2, Muestreo.fecha, Personas.nombre, Personas.apellido
FROM Muestreo JOIN Lugares ON Muestreo.lugar=Lugares.lid JOIN Personas ON
Muestreo.persona=Personas.pid WHERE (Lugares.nombre='MUSAC') AND (Muestreo.fecha>='2013-01-01')
AND (Muestreo.fecha<='2013-12-31') ORDER BY Personas.apellido DESC;
```

Como siempre, nuestra consulta al final de cuentas se ve algo monstruosa, pero logró lo que deseábamos hacer. Ahora que ya tenemos una idea sobre cómo trabajar con SQL, vamos a pasar a una parte que requiere de más seriedad y delicadeza de nosotros.

1.4. Creando, Actualizando y Eliminando

En este segmento vamos a pasar a tomar más responsabilidad de nuestras acciones y vamos a ir con el encargado de bodega a sacar y meter el equipo y los reactivos que necesitamos. Incluso vamos a ayudarle a mover muebles de la bodega! La idea ahora será llegar al principio: cómo crear una base de datos y agregarle datos. Esta parte, contrario a lo que cualquiera esperaría, es bastante sencilla, pero mucho más peligrosa. Si llegamos a agregar algo que no es real o a eliminar un dato, no hay vuelta atrás: hicimos algo mal y corregirlo cuesta.

1.4.1. Bases de Datos

Crear una base de datos en SQLite no es nada complicado. Hay otras DBs como MySQL o PostgreSQL que requieren que entremos al administrador de DB antes de crearla en sí. En el caso de SQLite, lo único que debemos hacer es lo mismo que hacemos para abrirla. En una línea de comando linux escribimos `sqlite3 nombre_de_mi_nueva.db` Y ya, con eso estamos dentro de nuestra nueva base de datos. Por supuesto, debemos utilizar nombres significativos en nuestros archivos.

Para eliminar una base de datos, la operación es aún más fácil. Puesto que una base de datos en SQLite es un archivo, solo necesitamos eliminarlo; eso es todo. Sin embargo, recordemos que borrar datos es algo **MUY** peligroso. Solo se debe de hacer si esos datos no nos van a servir a nosotros ni a nadie nunca.

1.4.2. Tablas

Crear una tabla ya es un proceso un poco más complicado. Para ello hay una serie de palabras y conceptos que debemos saber. De hecho, crear una tabla ya es algo que se hace con SQL. Para realizar este ejercicio, vamos a replicar la base de datos con la que estábamos trabajando. Para eso vamos a crear una nueva DB que se llamará *mi_monitoreo.db* Una vez estemos adentro, la vamos a configurar con `.headers ON` y `.mode columns` Ahora procederemos a crear la primera tabla.

```
CREATE TABLE Personas (
   pid INTEGER NOT NULL,
   nombre VARCHAR(15),
   apellido VARCHAR(15),
   PRIMARY KEY(pid)
);
```

La parte de crear la tabla llamada **Personas** es fácil de entender. Lo que no es tan sencillo son las líneas que siguen. Pero qué dicen realmente? En este caso, cada línea es un campo de la nueva tabla (excepto por la última línea, pero ya llegaremos a ella). En cada línea necesitamos especificar qué tipo de dato es el que vamos a almacenar. Por ello, para *pid* decimos que almacenará enteros `INTEGER` , y que jamás puede estar vacía `NOT NULL` El campo *nombre* almacenará caracteres de largo variable `VARCHAR` hasta un máximo de 15 caracteres `VARCHAR(15)` Lo mismo sucederá con el campo *apellido*. Finalmente tenemos una línea en la que especificamos que *pid* será nuestra llave primaria `PRIMARY KEY(pid)` : el identificador del que tanto hablábamos antes que existía en cada tabla. Veamos otro caso!

```
CREATE TABLE Lugares (
   lid INTEGER NOT NULL,
   nombre TEXT,
   latitud REAL,
   longitud REAL,
   PRIMARY KEY(lid)
);
```

Las diferencias no son muchas. Vale la pena mencionar que con `VARCHAR` le podemos colocar un límite a la cantidad de caracteres a almacenarse para que la base de datos no ocupe espacio de más. `TEXT` por otra parte, no pone límite y a este se le puede ingresar cantidades muy grandes de texto. El tipo de dato `REAL` especifica que los datos en ese campo son números reales. Y fuera de eso, no hay muchas diferencias con la tabla anterior. Intentemos crear la última tabla.

```
CREATE TABLE Lugares (
   mid INTEGER NOT NULL,
   perona INTEGER NOT NULL,
   lugar INTEGER NOT NULL,
   CO2 REAL,
   NO2 REAL,
   SO2 REAL,
   PM10 REAL,
   [PM2.5] REAL,
   pH REAL,
   fecha DATE,
   PRIMARY KEY(mid),
   FOREIGN KEY(persona) REFERENCE Personas(pid),
   FOREIGN KEY(lugar) REFERENCE Lugares(lid)
);
```

Aquí nos damos cuenta de que hay un campo nuevo: `DATE` Este almacena fechas nada más. Hay otro formato similar que puede almacenar fechas y horas. Sería un buen ejercicio averiguar cuál es. Por el momento, nos interesa ver qué es lo que agregamos al final. Esas últimas dos líneas son las que indican que un campo de nuestra tabla debe de referirse al campo de *otra* tabla. Esto tiene varias ventajas realmente. En la tabla de **Muestreo**, los campos *persona* o *lugar* no van a aceptar ningún valor que no exista en las otras tablas (**Personas** y **Lugares**). Además, no podremos eliminar una entrada de estas últimas dos tablas, si existe alguna entrada en **Muestreo** que se refiera a ellas. Esto es una gran ayuda!

Ahora mencionaremos nada más cómo es que las tablas se pueden eliminar. Esto último es bastante fácil en SQL. El comando para esto es `DROP TABLE Muestreo;` A veces la creación o eliminación de una tabla toma tiempo; seamos pacientes si así es el caso. Por ahora no eliminemos ninguna de nuestras dos nuevas tablas; nos servirán. Pero para proseguir, debemos ingresar algunos datos en ellas.

1.4.3. Entradas

Insertar nuevas entradas a una tabla es una tarea que, como todo en SQL, no es tan complicado. Sin embargo, se requiere de escribir bastante. Para insertar una nueva entrada, debemos especificar en qué tabla vamos a ingresar la entrada, qué columnas tiene, y qué valores vamos a ingresar. Agreguemos algunas entradas a la tabla de **Personas**.

```
INSERT INTO Personas(pid, nombre, apellido) VALUES(NULL, 'Alvaro', 'Garcia');
INSERT INTO Personas(pid, nombre, apellido) VALUES(NULL, 'Eduardo', 'Saquilmer');
INSERT INTO Personas(pid, nombre, apellido) VALUES(NULL, 'Francisco', 'Garcia');
INSERT INTO Personas(pid, nombre, apellido) VALUES(NULL, 'Elisandra', 'Hernandez');
```

Aquí hay algo peculiar. Notamos que para *pid* estamos ingresando el valor `NULL` . Si recordamos bien cuando creamos la tabla, especificamos que ese campo **no** podía estar vacío; no podía ser `NULL` Entonces qué pasó? Como especificamos que *pid* es una llave primaria, SQLite inserta un número en vez de dejar el campo vacío. Como también especificamos que ese campo solo podía almacenar números enteros, SQLite comienza numerando desde 1 en adelante. A cada nueva entrada le toca el número siguiente, a menos de que seamos nosotros los que asignemos un número en particular. Eso nos lleva a otra cosa que se puede hacer con las entradas.

Después de ingresar toda la información en la tabla de **Lugares**, nos damos cuenta de que nos equivocamos en algo: ingresamos INCAP una segunda vez, en vez de ingresar MUSAC. Clásico error de estar ingresando datos solo copiando y pegando la misma línea y haciéndole cambios cada vez. Ahora la pregunta es: qué hacemos?

```
INSERT INTO Lugares(lid, nombre, longitud, latitud) VALUES(1,'INSIVUMEH',-90.532677,14.487356);
INSERT INTO Lugares(lid, nombre, longitud, latitud) VALUES(2,'T10',-90.554700,14.585181);
INSERT INTO Lugares(lid, nombre, longitud, latitud) VALUES(3,'EFPEM',-90.545535,14.588231);
INSERT INTO Lugares(lid, nombre, longitud, latitud) VALUES(4,'San Juan',-90.547528,14.622473);
INSERT INTO Lugares(lid, nombre, longitud, latitud) VALUES(5,'INCAP',-90.539983,14.616133);
INSERT INTO Lugares(lid, nombre, longitud, latitud) VALUES(6,'INCAP',-90.510983,14.638987);
```

Para enmendar este tipo de errores, o para hacer cambios puntuales en una entrada en particular, existe una consulta de actualización de datos. Vamos a actualizar de una tabla, el valor de un campo establecido, en donde se cumpla alguna condición. Veamos esta consulta para entender mejor la situación:

```
UPDATE Lugares SET nombre='MUSAC' WHERE lid=6;
```

De esta forma corregimos el error que habíamos hecho. Si no ponemos una condición, se le pondrá de nombre MUSAC a todas las entradas. La condición es la que especifica en qué entrada se debe de hacer el cambio realmente. Finalmente, qué hubiera pasado si en vez de habernos equivocado en un nombre, hubieramos ingresado el mismo dato dos veces? La respuesta es evidente: borrar una de las dos entradas.

Supongamos que de casualidad agregamos otra vez la información del INCAP:
`INSERT INTO Lugares(lid, nombre, longitud, latitud) VALUES(NULL,'INCAP',-90.510983,14.638987);`
Lo que nos queda ahora es eliminar la última, para ello debemos saber seleccionar esa entrada únicamente. Pensemos por un momento, cómo seleccionamos solamente esa entrada? La única alternativa que nos queda es utilizar el identificador. Por qué? Porque es el único que declaramos como llave principal. Eso significa que es único en toda la tabla! Por eso diseñamos nuestra consulta de la siguiente manera:

```
SELECT * FROM Lugares WHERE lid=7;
```

Perfecto! La podemos mostrar de manera aislada. Pero esto de qué nos sirve? Pues que este es el primer paso para eliminar una entrada: saber cómo hallarla y aislarla. Eliminarla solo requiere de un pequeño cambio en la consulta que acabamos de hacer: `SELECT *` lo reemplazamos con `DELETE` . Entonces la consulta se ve de la siguiente manera:

```
DELETE FROM Lugares WHERE lid=7;
```

Y así de sencillo se puede eliminar una entrada.

1.4.4. Construyendo y Exportando una Base de Datos

Si seguimos las indicaciones del taller hasta ahora, tenemos una base de datos nueva con dos tablas: la de **Personas** y la de **Lugares**. La de **Muestreo** la habíamos creado, pero la eliminamos. Por otra parte, todas estas consultas que hemos hecho las hemos tenido que ir ingresando una a una para acceder a los datos y editarlos. No sería más eficiente que nuestro ordenador pudiera solo leer todas las instrucciones de un solo y armar la base de datos o realizar una consulta complicada? Pues sí la hay! Vamos a crear la tabla de **Muestreo** y llenarla con todos los datos que esta lleva sin escribir más que una línea. En nuestro directorio de documentos se halla un archivo de texto llamado *monitoreo.txt*. Abrámoslo por un momento y, sin alterar el contenido, entendamos lo que hay en él.

Posteriormente, después de entender eso, vamos a volver a nuestra línea de comando con nuestra base de datos nueva y en ella vamos a escribir `.read monitoreo.txt` Esto, en SQLite, lee todo el contenido del archivo y lo ejecuta. Puede que se tarde un momento, pero al final, la tabla de **Muestreo** no solo estará creada, sino llena con toda la información también. Esto resulta más cómodo, especialmente cuando tenemos datos en hojas de cálculo (Miscrosoft Excel) y las deseamos pasar a una base de datos. El procedimiento para hacerlo es similar. Esto es otra cosa que sería conveniente investigar después.

Finalmente vamos a hacer el mismo procedimiento que acabamos de hacer, pero al revés. Vamos extraer las instrucciones de creación y llenado de una base de datos. Esto resulta útil cuando no estamos trabajando con una base de datos SQLite (que es portátil). Además, también resulta útil si se desea migrar de una base de datos a otra (e.g. MySQL a SQLite), el SQL suele ser el mismo o tener pocas variaciones. Veamos cómo se hace.

Hacer una copia de seguridad de las instrucciones de la base de datos es algo que, si bien se puede hacer desde dentro de la DB, es más sencillo hacerlo desde afuera de ella en el caso de SQLite. Lo que debemos hacer es colocarnos en el directorio con la DB e ingresar: `sqlite3 mi_base_de_datos.db '.dump' > mi_dump.sql` El resultado será un nuevo archivo en nuestra carpeta con todas las instrucciones para ensamblar y llenar una base de datos.

La alternativa a hacer esto es hacerlo desde dentro de la DB. Para ello vamos a ingresar a la base de datos mediante: `sqlite3 monitoreo.db` y luego vamos a decirle 2 cosas al administrador: que queremos que los resultados los guarde en un archivo `.output mi_dump2.sql` y que deseamos que ese archivo se llene con todas las

instrucciones de creación de la base de datos `.dump` Eso es todo.

Antes de terminar con esta sección, es recomendable que abramos uno de los archivos que acabamos de crear y los veamos completamente. Vamos a notar que las primeras 2 líneas y la última no son algo que nosotros entendamos. Esto es solo algo que agregó el administrador al hacer el *dump*. Borremos esas líneas y nuestro archivo queda listo para armar una base de datos en cualquier momento.

1.5. Biblioteca y Toxicidades

La mayoría de bibliotecas en el mundo se manejan ya con ficheros electrónicos. Utilizan el código ISBN de los libros como identificador y además permiten listar la información de cada libro, además de su disponibilidad, en línea. Esto es una clara ventaja para cuando se desea buscar dentro de los contenidos de una biblioteca. Sin embargo, algo que no se ha implementado en la mayoría de ellas es que de *esa* base de datos se puedan crear, de manera inmediata, las bibliografías que a veces tanto tiempo nos cuesta crear. Más adelante veremos que esto se podría programar y se podría ligar a documentos de Microsoft Word o LaTeX.

Otro proyecto similar sería la creación de una base de datos con todas las toxicidades de todos los reactivos con los que se vaya trabajando. De crearla así, se podría obtener la tabla de toxicidades para cada laboratorio solo buscando los reactivos a utilizar, y sin tener que leer la hoja de seguridad entera. Es cierto que al ver la hoja de seguridad uno aprende cómo es que funcionan ellas, pero repetir lo mismo durante todos los laboratorios de una carrera resulta tedioso. Esto, como se verá adelante, también se podría programar y dejar creado para futuras generaciones.

Ambos proyectos serían interesantes para quien los quisiera llevar a cabo como ejercicio. No son nada complicados, aunque sí algo tediosos, puesto que las bases de datos se tendrían que ir llenando al principio. Para quien se desee aventurar, esto podría ser un proyecto muy interesante para estudiantes de pregrado.

1.6. Comentarios Finales

Felicidades, has completado el segundo día del taller de QCA. Ahora ya tienes idea de qué es una base de datos, qué se puede hacer con ella y cómo hacerlo. Como dijimos en algún momento durante el desarrollo, el uso de SQL es menor en estos días gracias a los ORMs que facilitan mucho del trabajo. Sin embargo, conocer cómo manejar una base de datos es de vital importancia en un mundo en el que todo se almacena en ellas.

Si deseas continuar aprendiendo sobre el tema, sería recomendable que buscaras en internet tutoriales, ejemplos y las instrucciones detalladas de todos los comandos que existen en cada base de datos. También es conveniente probar otras bases de datos como MySQL o PostgreSQL. Con eso sería suficiente para decir que ya puedes hacer uso de ellas.

Felicidades nuevamente, nos vemos mañana para aprender sobre control de revisiones. Ánimo!

2. Glosario de comandos sencillos

- **.help**: Solicita la página de ayuda de SQLite.
- **.tables**: Muestra qué tablas hay en la base de datos actualmente.
- **.headers ON/OFF**: Activa o desactiva que se muestren los encabezados al desplegar el resultado de una consulta.
- **.mode**: Cambia el formato para el despliegue del resultado de una consulta. Opciones: *csv*, *column*, *html*, *insert*, *line*, *list*, *tabs*, *tcl*.
- **.output FILE**: Especifica el archivo en el que se van a almacenar los resultados desplegados por el administrador.
- **.dump TABLE***: Generar SQL de creación y llenado, desde una base de datos ya hecha. Puede tratarse de una tabla individual o de toda la base de datos.
- **.schema TABLE**: Generar SQL para la creación de una tabla dada.
- **.read FILE**: Leer SQL de un archivo dado.
- **.exit**: Salir del programa.
- **.quit**: Salir del programa.

- **.import FILE TABLE**: Importar datos desde un arhivo dado, hacia una tabla específica.
- **SELECT**: Seleccionar y mostrar campos.
- **FROM**: Especifica una tabla a la cual se debe hacer referencia.
- **WHERE**: Abre la posibilidad a filtrar los resultados mediante condiciones.
- **DISTINCT**: Elimina duplicados de una lista de entradas.
- **ORDER BY**: Ordena las entradas de un campo en particular.
- **DESC**: Modifica un ordenamiento para que sea al revés.
- **AND**: Operador lógico que permite otra condición.
- **LIKE**: Compara entradas y revisa que alguna se parezca a una expresión dada.
- **JOIN**: Junta una tabla con otra.
- **ON**: Especifica mediante qué campos es que se juntan dos tablas.
- **CREATE TABLE**: Crea una tabla.
- **INTEGER**: Tipo de dato - *número entero*.
- **VARCHAR()**: Tipo de dato - *caracteres variables*. Puede colocársele límite.
- **DATE**: Tipo de dato - *fecha*.
- **TEXT**: Tipo de dato - *texto*.
- **REAL**: Tipo de dato - *número real*.
- **NOT NULL**: Especifica que el dato no puede ser vacío; no puede ser `NULL` .
- **PRIMARY KEY**: Identificador principal de las entradas en una tabla - llave principal.
- **FOREIGN KEY**: Identificador principal de las llaves de una tabla que no es con la que se está trabajando - llave externa.
- **REFERENCE**: Especifica una tabla y un campo a los que se debe de hacer referencia si se está creando una tabla.
- **INSERT INTO**: Crea y almacena una nueva entrada en una tabla dada.
- **VALUES()**: Valores a ser insertados como entrada en una tabla.
- **UPDATE**: Actualizar los valores en una tabla.
- **SET**: Establece qué campo debe de ser actualizado y cómo.
- **DELETE**: Elimina entradas de una tabla.
- **min()**: Busca el valor mínimo en un campo.
- **max()**: Busca el valor máximo en un campo.
- **avg()**: Calcula la media de los datos en un campo.
- **sum()**: Suma todos los datos de un campo.
- **count()**: Cuenta cuántos datos hay en un campo.

Licencia