# Bases de Datos

TB022 - Esteban - Riesgo - Kristal

# Qué es una base de datos?

Como el nombre lo indica, una BD es cualquier sistema de almacenamiento que permite persistir información de manera "permanente" que no dependa de la ejecución de un programa. Esta información se guarda en discos de almacenamiento (disco duro).

La forma más simple que puede tomar una base de datos es la de un archivo de texto.

# Clasificación

Si bien cualquier sistema puede desarrollar su propia base de datos con el formato y lógica que se les ocurra, las bases de datos se suelen clasificar en dos categorías:

- Bases de Datos relacionales
- Bases de Datos NO relacionales

# Bases de Datos Relacionales (RDBS)

El concepto fue ideado inicialmente en los 70. Si bien la idea de organizar la información en tablas es mucho anterior (de toda la historia de la humanidad más o menos) los primeros sistemas informáticos dedicados exclusivamente a este propósito se empezaron a desarrollar en esta época.

La información se presenta en tablas, donde cada tabla representa un tipo de entidad; cada columna de la tabla los campos de la misma; y cada fila representa una instancia de esa entidad.

# Bases de Datos Relacionales (RDBS)

Customer

| CustomerID | CustomerName | LastName | Country | Age | Phone |
|---|---|---|---|---|---|
| 1 | Shubham | Thakur | India | 23 | xxxxxxxxxx |
| 2 | Aman | Chopra | Australia | 21 | xxxxxxxxxx |
| 3 | Naveen | Tulasi | Sri lanka | 24 | xxxxxxxxxx |
| 4 | Aditya | Arpan | Austria | 21 | xxxxxxxxxx |
| 5 | Nishant. Salchichas S.A. | Jain | Spain | 22 | xxxxxxxxxx |

Cada *tabla* define una entidad

Cada *columna* representa los campos de la tabla. Qué propiedades tienen las entidades de tipo Customer.

Cada *fila* representa una instancia de una entidad, un Costumer en este caso. Cada uno con sus propios valores para cada uno de los campos. Cada Consumer tiene un identificador único.

# BDD No Relacionales

Tambien llamadas NoSQL. Incluyen a cualquier tipo de base de datos que no sea una RDB.

El concepto es muy amplio, existen categorías para este tipo de DB también. Las RDB son de uso general, se pueden usar para afrontar una gran cantidad de problemas distintos. Las bases de datos NoSQL tienen cada una un uso muy específico. Justamente están hechas para esos pocos casos en los cuales las RDB no son buenas.

Algunos tipos y las implementaciones más comunes

**Documents** - MongoDB (open source), CouchDB (Amazon), CosmosDB (Azure)

**Key Value** - Redis (era open source), Memcached (open source), DynamoDB (Amazon)

**Graph** - neo4j (open source), Neptune (Amazon)

**No** vamos a adentrarnos en este tipo de base datos. Nos vamos a enfocar únicamente en RDBs

# Structured Query Language : SQL

Desde su concepción, las RDBs utilizan un lenguaje propio llamado SQL que sirve de interfaz para manejar las operaciones (consulta, creación, actualización, etc.) contra la información guardada en disco.

Si bien se habla de SQL en general, no existe una única implementación de este lenguaje. Hay muchas bases de datos relacionales que tienen cada una su propia implementación de SQL. Si bien hay una base común para todas, hay operaciones y ciertas estructuras que cambian según qué RDBM se use.

# Relational Database Management System (RDBMS)

Sistemas para manejar RDBs.

Como mencionamos, hay muchos, pero los más conocidos son:

MySQL

PostgreSQL

SQLite

Nosotros vamos a usar SQLite por sus simplicidad. Es el que menos features tiene de éstos 3, pero es el que más fácil nos va a ser de usar.
Todo el lenguaje SQL que veamos va a ser exclusivo de SQLite.

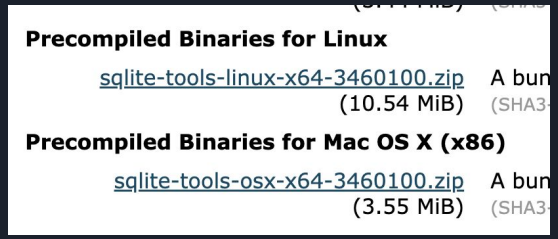
# SQLite

Como su nombre lo dice, es una versión "lite" de un RDBM que está pensado para ser usado en sistemas simples y que no necesiten escalar infinitamente (es decir, no está pensado para usar en sistemas con miles de millones de registros).

Es tan simple en su funcionamiento que va a guardar un único archivo con toda la información de la base de datos (un archivo binario, no se puede leer con otros sistemas que no sean SQLite).

Instalar

Ir a https://www.sqlite.org/download.html y descargar lo siguiente según su sistema:




Este zip contendrá 3 archivos. Recomendamos hacer lo siguiente:

1. Descomprimir el .zip y guardar los archivos un directorio, por ejemplo `~/Downloads/sqlite`
2. Agregar al PATH el path del ejecutable `sqlite3` que colocamos en el directorio anterior (para poder ejecutarlo desde cualquier parte de nuestro sistema).

# SQLite

El ejecutable SQLite3 nos provee una CLI para hacer queries contra cualquier base de datos (creada por SQLite) que tengamos guardada en nuestro sistema.

Si lo ejecutamos sin argumentos trabajaremos sobre una DB en memoria, al cerrar el programa (`.quit`) se borrará todo lo que hicimos.

Le podemos pasar como argumento el nombre de la DB persistida en disco queremos acceder. Si no existiera todavía, la crea. Recordemos que crear una DB va a ser simplemente crear un archivo.

# SQL: tipos de datos

**NULL** - Valor nulo.

**INTEGER** - Número entero con signo, se guarda en 0, 1, 2, 3, 4, 6, o 8 bytes según su valor.

**REAL** - Número decimal, un float. Se guarda como un 8-byte IEEE floating point number.

**TEXT** - String de tamaño variable.

**BLOB** - Tipo de dato genérico. Se guarda así como viene. Puede ser cualquier cosa. Se utiliza para representar cualquier otra cosa no contemplada por los demás tipos (por ejemplo, listas).

NO hay un tipo BOOL. Se representa con un 0 o un 1.

# SQL: CREATE TABLE

```
CREATE TABLE <table_name> (
    <column_name> <datatype> <constraint>,
    ....
    <constraints>
);
```

Una constraint es una limitación o regla que se impone sobre el campo puntual.

Algunas posibles son:

`NOT NULL` - no permite valores nulos

`UNIQUE` - no permite valores repetidos

Se pueden poner junto a la definición del campo o al final de la especificación.

Creemos nuestra tabla de alumnos para representar a un alumno como hicimos la clase pasada.

# SQL: CREATE TABLE

```
CREATE TABLE alumno (
    nombre TEXT NOT NULL,
    apellido TEXT NOT NULL,
    notas BLOB,
);
```

Y el padrón?

Por defecto, toda tabla en SQLite tiene una columna `ROWID` que es un `INTEGER` auto incremental. Podemos usar ésto como nuestro padrón.

También podemos deshabilitarlo indicando a la tabla `[WITHOUT ROWID]` y teniendo nuestro campo padron `INTEGER NOT NULL AUTOINCREMENT`. Si hacemos esto, debemos usar también la constraint *`PRIMARY KEY`*.

# SQL: primary key

En casi todos los casos, toda tabla debería tener una *primary key*.

Una primary key es una propiedad que se le puede asignar a una columna de una tabla para indicar que va a ser el identificador inequívoco mediante el cual se va a identificar cada registro (fila).

Solo puede haber una primary key por tabla que puede ser simple (una sola columna)

```
(PRIMARY KEY column)
```

o compuesta (la unicidad está dada por una combinación de dos o más columnas).

```
(PRIMARY KEY column1, column2)
```

Si definimos una columna de tipo INTEGER y le agregamos la constraint PRIMARY KEY esa columna va a ser un alias de ROWID. Es decir, van a ser efectivamente la misma columna.

# SQL: CREATE TABLE

```
CREATE TABLE alumno (
    padron INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    apellido TEXT NOT NULL,
    notas BLOB
);
```

De esta manera, tenemos nuestra tabla alumnos con los campos que nosotros queremos.

Y cómo veo que esta tabla se creó bien? O como veo el detalle de una tabla creada anteriormente?

# SQL: CREATE TABLE

sqlite tiene varios comandos para ver la información de nuestra base de datos

`.tables` - lista todas las tablas en la base

`.schema <tabla>` - muestra la estructura de una tabla puntual

Todos los comandos están listados acá.

# SQL: INSERT

Ahora que creamos una tabla, vamos a agregarle data.

```
INSERT INTO <table_name> (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Podemos poner las columnas en cualquier orden (con los valores que queremos para cada una en el mismo orden).

Si vamos a insertar valores para todas las columnas podemos simplemente hacer

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

Los valores tienen que estar en el orden que las columnas tienen en la tabla.

Podemos insertar múltiples rows en un mismo INSERT: `VALUES (value1_1, ...), (value2_1, ...);`

# SQL: SELECT

Necesitamos visualizar la información que almacenamos.

```
SELECT column1, column2, ...
FROM table_name;
```

Podemos elegir exactamente qué columnas queremos ver de la tabla en cuestión.

Si queremos seleccionar todas las columnas de la tabla

```
SELECT *
FROM table_name;
```

SELECT nos va a devolver todas las filas/registros de la tabla en cuestión.

No siempre  vamos a querer todos los registros. Podemos filtrar qué cosas queremos que nos devuelva en base a reglas usando *WHERE*

# SQL: WHERE

La sentencia `WHERE` nos permite definir una o múltiples reglas tan complejas como queramos para determinar qué registros queremos obtener de una tabla puntual.

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN <op> <condicion>;
```

El uso más común que se le da es el de poner alguna condición sobre una o múltiples columnas.

Los operadores que podemos usar:

Los conocidos: `=, >, <, >=, <=, <>`

`LIKE` parecido a una regexp, para buscar patrones en campos de tipo texto.

`IN` para matchear un valor contra una lista de valores posibles

# SQL: AND, OR, NOT, BETWEEN

Se usan para crear condiciones más complejas sobre un `WHERE`.

Para ver más detalles ir a este link.

# SQL: COUNT, SUM, AVG, MAX, MIN

Todas estas son funciones que nos permiten calcular datos sobre los resultados obtenidos de una query.

`COUNT` - contar cantidad de resultados

`SUM` - suma de los valores sobre una columna puntual

`AVG` - promedio de los valores sobre una columna puntual

`MAX/MIN` - maximo/minimo de los valores sobre una columna puntual

# SQL: DISTINCT

Modificador que se aplica a varias otras sentencias, por ejemplo a las funciones que vimos recién, para especificar que no se tengan en cuenta valores repetidos.

`COUNT(columna)` - cuenta cantidad de resultados totales que no son nulos de esa columna

`COUNT(DISTINCT columna)` - cuenta la cantidad de resultados totales que no son nulos *sin contar repetidos* de esa columna

O mismo tambien

`SELECT DISTINCT <columna> FROM <tabla>` - trae todos los posibles valores de esa columna en la tabla, sin repetir.

# SQL: ORDER BY

Podemos especificar en qué orden queremos el resultado

`ORDER BY <column> [ASC/DESC]` ordenará los resultado devueltos por la columna especificada en orden ascendente o descendente.

# SQL: LIMIT

`LIMIT N` - limita la cantidad de resultados devueltos hasta N elementos.

Se suele usar en combinación con `ORDER BY` para traer los N elementos con mayor/menor valor.

# SQL: UPDATE

```
UPDATE <table>
SET column_1 = new_value_1,
   column_2 = new_value_2
WHERE
   <search_condition>;
```

Aplicará el mismo update a todas las entradas de la tabla que cumplan la condición de búsqueda.

Para la búsqueda podemos usar todas las instrucciones mencionadas hasta el momento.

# SQL: DELETE

```
DELETE FROM table
WHERE <search_condition>;
```

Eliminará de la tabla a todas las entradas que cumplan la condición de búsqueda.

Para la búsqueda podemos usar todas las instrucciones mencionadas hasta el momento.

Como hacer un delete es permanente y no hay una forma fácil de recuperar los datos, una buena práctica es hacer en cambio ***soft deletes***.

Esto implica agregar en las tablas una columna extra para indicar si una entrada está activa o fue borrada. Esto se puede hacer de muchas maneras (un booleano, un estado, un timestamp para indicar cuándo fue borrado)

Todas las queries normales deberían agregar la condición `where deleted = false/where deleted_at = null/where status != 'DELETED'.`

Si se quiere investigar o recuperar entradas 'borradas' se puede hacer fácilmente con queries que incluyan esas filas.

# SQL: ALTER TABLE

Como todo, los sistemas evolucionan en el tiempo. Lo que originalmente estaba cubierto con un diseño inicial, hoy en día puede no estarlo.

A veces es necesario modificar las tablas de una DB para agregar, sacar o modificar columnas y otras cosas.

```
ALTER TABLE <table_name>
ADD <column_name> <datatype>;
```

```
ALTER TABLE <table_name>
RENAME COLUMN <current_name> TO <new_name>;
```

También se puede cambiar el nombre de la tabla

```
ALTER TABLE <table_name>
DROP COLUMN <column_name>;
```

```
ALTER TABLE <table_name>
ALTER COLUMN <column_name> <datatype>;
```

Pero hay un problema.
Qué pasa si ya tengo información en la tabla y la modifico?

Si elimino la columna, pierdo la información. Y qué pasa si le cambio el tipo?

*Más sobre esto la próxima clase*

# Relaciones entre Tablas

# SQL: relación entre tablas

Hasta ahora todas las operaciones que hicimos interactúan con una sola tabla. Pero en la realidad, nuestros modelos y datos están relacionados entre sí.

En estos casos, nuestras tablas tendrán información de cómo se relacionan entre sí (esta información estará guardada en alguna tabla) y luego deberemos hacer queries con sentencias especiales para poder traer toda esta información conjunta.

Las relaciones entre entidades/tablas pueden ser de varios tipos que son determinados según la cantidad de posibles instancias entre cada parte de la interacción.

# Relación 1:1

Si tuviéramos en nuestro sistema, un modelo de Direccion donde todo alumno tiene que tener una dirección registrada, y sólo puede tener una. Decimos entonces que Alumno tendría una **relación 1:1** con Direccion. Es decir, para cada Alumno existe únicamente una Direccion, y para cada Direccion existe únicamente un Alumno.

En nuestras tablas esto se vería reflejado con un nuevo tipo de constraint: *foreign keys*

```
CREATE TABLE <table_name> (
   ...
   FOREIGN KEY (<field_name>)
      REFERENCES <other_table_name> (<other_table_field_name>)
);
```

Nombre del campo en `<table_name>`

Nombre de la tabla referenciada

Nombre del campo referenciado en la otra tabla

# Relación 1:1

Si tuviéramos en nuestro sistema, un modelo de Direccion donde todo alumno tiene que tener una dirección registrada, y sólo puede tener una. Decimos entonces que Alumno tendría una **relación 1:1** con Direccion. Es decir, para cada Alumno existe únicamente una Direccion, y para cada Direccion existe únicamente un Alumno.

```
CREATE TABLE direccion (
    direccion_id INTEGER PRIMARY KEY,
    calle TEXT NOT NULL,
    altura INTEGER NOT NULL,
    localidad TEXT NOT NULL,
    ...
);
```

```
CREATE TABLE alumno (
    ...
    direccion_id INTEGER UNIQUE NOT NULL,
    FOREIGN KEY (direccion_id)
        REFERENCES <direcciones> (<direccion_id>)
);
```

No puede haber una foreign key sobre un ROWID, hay que especificar un campo puntual que puede ser el alias.

# Relación 1:1

Una foreign key es una relación muy fuerte entre dos tablas.
No podemos borrar una dirección sin antes borrar la referencia del alumno que lo apunta (cambiando la referencia por otra instancia, o eliminando el Alumno directamente).

Podemos sin embargo darle propiedades a la foreign key para decirle a SQLite qué debe pasar en esos casos.

```
FOREIGN KEY (direccion_id)

    REFERENCES <direcciones> (<direccion_id>)

        ON DELETE <action>
```

Si queremos que cuando se elimine un Alumno se elimine la Dirección la acción a usar es `CASCADE`.

Si queremos que siga existiendo la Dirección sin ningún Alumno que la apunte, entonces `SET NULL`.

Si queremos prevenir la acción (el comportamiento default) podemos usar `RESTRICT`.

Todo esto es válido para `DELETE` y/o para `UPDATE`.

Chequear la sección *SQLite foreign key constraint actions* de este link.

# Relación 1:1

No necesariamente por ser una relación 1:1 significa que ambos modelos tengan que estar siempre presentes.

Suponer que se tienen dos tablas: Vehículos y PolizasDeSeguro.

Cada vehículo puede tener una póliza o no, pero no más de una.
Una póliza tiene sí o sí un vehículo a la que pertenece, sino no puede existir.

En este caso, la foreign key estará presente en el modelo PolizaDeSeguro.

# Relación 1:N

Ahora queremos modelar la siguiente situación: dentro la facultad (y particularmente dentro de nuestro materia pero podemos ignorar esto por ahora) tenemos grupos para  hacer el TP2.

Cada Alumno tiene un único Grupo asignado, pero cada Grupo tiene varios Alumnos. Decimos que esta es una relación 1:N entre estas entidades (acá importa mucho el orden en que se exprese porque cambia totalmente el sentido de la relación).

De nuevo necesitamos usar foreign keys pero esta vez sin las constraints de `UNIQUE NOT NULL`.

```
CREATE TABLE grupo_tp (
    ...
    grupo_id INTEGER PRIMARY KEY
);
```

```
CREATE TABLE alumno (
    ...
    grupo_id INTEGER,
    FOREIGN KEY (grupo_id)
        REFERENCES <grupo_tp> (<grupo_id>)
);
```

# Relación 1:N

```
CREATE TABLE grupo_tp (
    ...
    grupo_id INTEGER PRIMARY KEY
);
```

```
CREATE TABLE alumno (
    ...
    grupo_id INTEGER,
    FOREIGN KEY (grupo_id)
        REFERENCES <grupo_tp> (<grupo_id>)
);
```

Si la relación es (en este caso) Grupo 1 : N Alumnos

Significa que para cada grupo va a haber N Alumnos, y para cada Alumno, 1 Grupo.

Por lo tanto la *foreign key* se tiene que definir en la tabla `alumno`

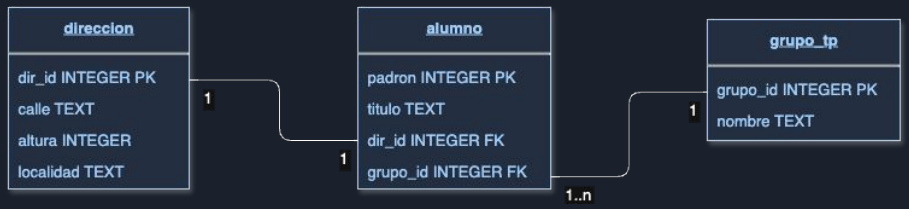
La constraint de `NOT NULL` podría estar presente también si fuese necesario (en este caso, es necesario?)

Cómo obtenemos todos los alumnos asignados a un grupo?

# DER

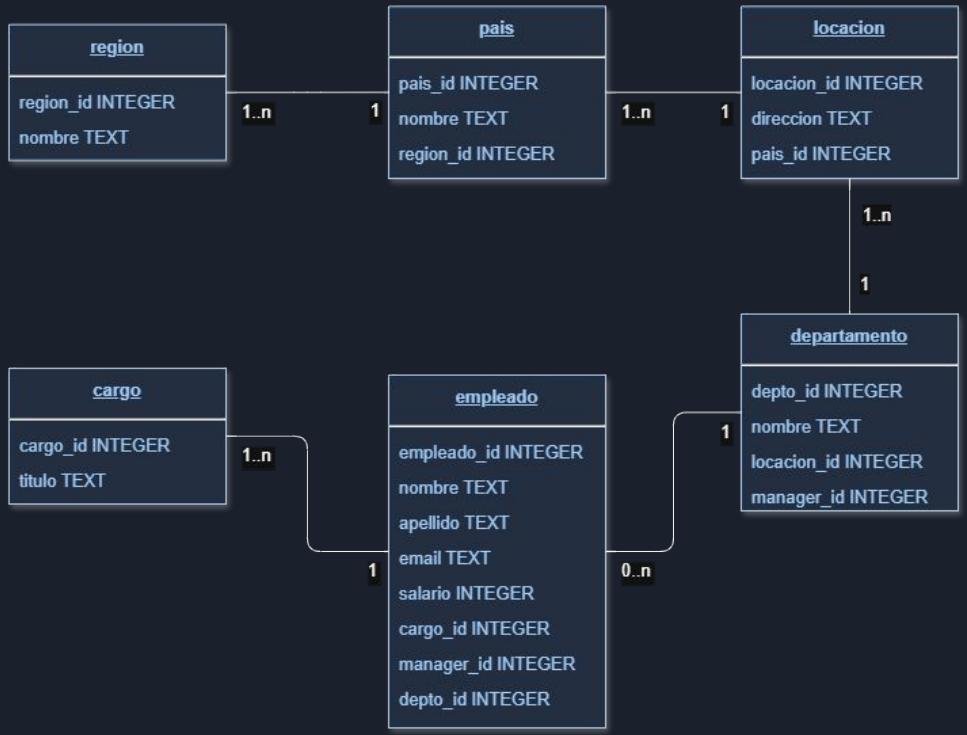
Diagrama Entidad Relación

Este tipo de diagramas sirven para representar las relaciones entre las tablas de una DB, y los campos de cada una.
Siguiendo el ejemplo que veníamos viendo, el DER de nuestra DB por ahora sería algo así:




Cada tabla tiene información de sus campos con sus tipos y sus constraints. Además, se reflejan las relaciones entre cada tabla.

# Ejemplo DER




Cada país tiene exactamente una región, pero una región tiene muchos países.

Qué otras relaciones hay?

Hay algunas que no están marcadas explícitamente en el diagrama. Cuáles son?

Cómo haría para conseguir la info de un Empleado junto con el nombre del departamento donde trabaja? Y su título? Y su país?

# Relación N:M

Ahora queremos modelar la siguiente situación: dentro la facultad tenemos Alumnos y también tenemos Materias. Queremos modelar la relación entre estos de manera que podamos ver en qué materias se anotó cada alumno y ver qué alumnos hay anotados en cada materia.

Un alumno va a tener N materias las cuales está cursando, y una materia tiene M alumnos que la están cursando. Para modelar esta relación debemos hacer uso de una tabla intermedia.

```
CREATE TABLE cursada (
    ...
    materia_id INTEGER,
    alumno_padron INTEGER,
    FOREIGN KEY (materia_id) REFERENCES <materias> (<materia_id>)
    FOREIGN KEY (alumno_padron) REFERENCES <alumnos> (<padron>)
);
```

El curso además debería tener información del cuatrimestre, por ejemplo

Un alumno no debería poder cursar la misma materia más de una vez por cuatrimestre.

Cómo modelaríamos eso?

# Tabla Intermedia

Para pensar las relaciones en una tabla, hay que tratar de razonarlo desde el lado opuesto al que uno normalmente lo haría en una lenguaje de programación.

Si un Grupo de TP tiene varios alumnos, en Python, lo reflejaría poniendo una atributo en Grupo que sea una lista de alumnos.

En SQL, la relación sería al revés: en Alumno, hay una FK relacionándolo con un Grupo.

Cuando la relación es de muchos a muchos, no hay ninguna entidad que pueda tener un campo con un ID, ya que ambos tendrían listas. Para esto surge la tabla intermedia.

Si la relación es TablaA N : M Tabla B entonces habrá una tabla intermedia TablaAB con el siguiente formato:

```
CREATE TABLE tablaAB (
  ...
  tablaA_id INTEGER,
  tablaB_id INTEGER,
  FOREIGN KEY (tablaA_id) REFERENCES <tablaA> (<tablaA_id>)
  FOREIGN KEY (tablaB_id) REFERENCES <tablaB> (<tablaB_id>)
);
```

# Tabla Intermedia

```
CREATE TABLE tablaAB (
  ...
  tablaA_id INTEGER,
  tablaB_id INTEGER,
  FOREIGN KEY (tablaA_id) REFERENCES <tablaA> (<tablaA_id>)
  FOREIGN KEY (tablaB_id) REFERENCES <tablaB> (<tablaB_id>)
);
```

La tabla intermedia tendrá entonces al menos dos columnas, una para el ID de cada tabla que quiere conectar.
Cada una de estas será una FK con la tabla original.

Para saber todos los B de un A:

```
SELECT tablaB_id
FROM tablaAB
WHERE tablaA_id = <id>;
```

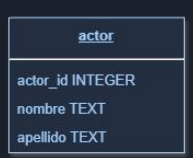
# Tabla Intermedia

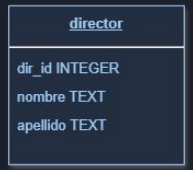
Cuando tiene sentido que la tabla intermedia tenga más campos?

En el caso anterior la tabla intermedia sólo servía para conectar ambas tablas en la relación N:M, pero en otros casos, es una entidad en sí misma.

En el ejemplo anterior, Cursada es una tabla intermedia entre Alumno y Materia pero tiene campos adicionales como el cuatrimestre que corresponde entre otras.

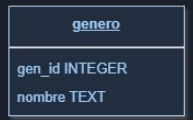
# Ejemplo DER




Tenemos estas 4 tablas y las queremos relacionar entre sí. Qué relaciones tendríamos y de qué tipo?


Algunas tablas intermedias pueden tener campos extra si tiene sentido más allá de solamente conectar las tablas.



# Ejemplo


Al tener una relación 1:N entre `actor` y `peli_actor`; y otra 1:M entre `pelicula` y `peli_actor`, hace que efectivamente haya una relación N:M entre `película` y `actor`.

# Join

En una misma query podemos obtener información de un tabla junto con los datos que precisemos de las tablas con las cuales se relacione.

Esto lo hacemos mediante la operación JOIN.

```
SELECT
 <tablaA>.<campo>,
 ...
 <tablaB>.<campo>
FROM
 <tablaA>
 INNER JOIN <tablaB> ON <tablaA>.<campo_foreign_con_tablaB_id> = <tablaB>.<tablaB_id>;
```

Podemos seleccionar los campos que queremos de cualquiera de las dos tablas

Indicamos la tabla que queremos unir

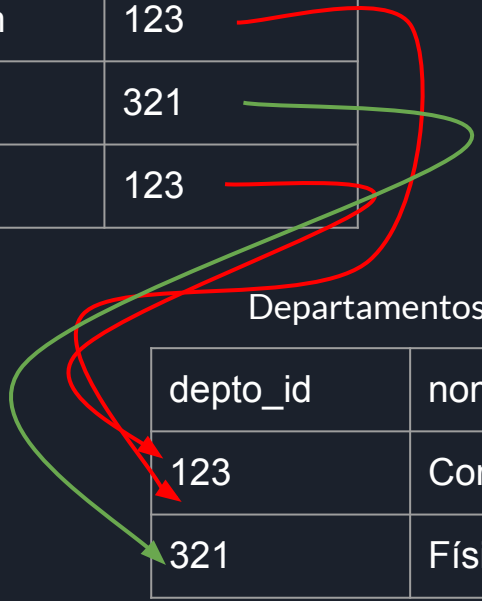
Indicamos qué campo usar para unir ambas tablas

# Ejemplo

Empleados

| empleado_id | nombre | apellido | depto_id |
|---|---|---|---|
| 1 | Fede | Esteban | 123 |
| 2 | Juani | Kristal | 321 |
| 3 | Dani | Riesgo | 123 |

```
SELECT ..
FROM empleados
JOIN departamentos
     ON empleados.depto_id =
departamentos.depto_id
```



Departamentos

| depto_id | nombre | manager_id | region_id |
|---|---|---|---|
| 123 | Computación | 99 | 3 |
| 321 | Física | 99 | 3 |

# Ejemplo

joined_table

| empleado_id | nombre | apellido | depto_id | nombre | manager_id | region_id |
|---|---|---|---|---|---|---|
| 1 | Fede | Esteban | 123 | Computación | 99 | 3 |
| 2 | Juani | Kristal | 321 | Física | 99 | 3 |
| 3 | Dani | Riesgo | 123 | Computación | 99 | 3 |

Al hacer el JOIN, se combinan ambas tablas en una nueva tabla temporal con LAS COLUMNAS de ambas y LAS FILAS de la tabla sobre la cual se hace la query (la del FROM).

De la tabla original (`Empleados`) se lee el `depto_id` de cada row y se busca en la tabla del JOIN (`Departamentos`) la que matchee.

Cuando se encuentra el match, coloca todo el contenido de esa row en las columnas restantes de la nueva tabla temporal.

# JOIN

Hay distintos tipos de JOINs cuyas diferencias tienen que ver como manejan el conjunto de resultados obtenido de la combinación de ambas tablas.

Por defecto, se realiza un INNER JOIN.

Para más detalles, consultar esta respuesta de Stackoverflow que es muy completa.

Y particularmente para SQLite esta página.

# Bonus Tracks

# SQL: GROUP BY

La información que está en nuestras tablas podríamos querer tenerla agrupada por algún criterio.

En el ejemplo de Chinook, tenemos Tracks (canciones) que tienen un Artist que las compuso. Podría querer ver información agrupada por artista:

- cuantas canciones hizo cada artista
- cual es la canción más larga por cada uno
- cuantas canciones hay por cada género en cada artista

```
SELECT <fields>
FROM <table>
WHERE <condition>
GROUP BY <fields>
```

# SQL: HAVING

# SQL: INDEXES

# Fin