SQL

desde cero

Pere Chardi Garcia

Índice

Introducción

Soy Pere Chardi, ingeniero técnico informático. Este proyecto es fruto en primer lugar de una inquietud personal por explorar el campo de la docencia, y en segundo lugar del fracaso cosechado al intentar encontrar un trabajo para ejercer de docente. No tuve éxito, por lo que de momento he descartado dedicarme a ello. Sin embargo he decidido convertir este fracaso en una oportunidad para desarrollar un curso sobre SQL, como hobby, como ejercicio personal, con el deseo de que usted encuentre aquí el curso que andaba buscando.

El **objetivo** de este proyecto es ofrecer un curso para que usted, si lo desea, pueda aprender el lenguaje de consulta estructurado SQL desde cero. Aunque en un futuro quizás amplíe los contenidos para que los iniciados, e incluso los entendidos, también puedan encontrar aquí técnicas y métodos que les permitan enriquecer sus conocimientos en materia de bases de datos relacionales y lenguaje de consulta SQL.

La metodología de este curso pretende ofrecer **un sistema de aprendizaje motivador**, sin más teoría que la necesaria para empezar a abordar las bases del lenguaje SQL, y usando siempre ejemplos y **casos prácticos** con el propósito de facilitar la comprensión e ilustrar toda su potencia. Con **ejercicios resueltos** que le permitirán desarrollar sus conocimientos, además de disponer de una extraordinaria herramienta, un **banco de pruebas** para practicar y jugar con el lenguaje que encontrará en la página web: deletesql.com. En esencia pretende ser un procedimiento de aprendizaje ameno, donde el principiante encuentre la motivación y el interés suficiente por la materia que muchas veces no se alcanza por parecer al principio un asunto demasiado tedioso y difícil de entender, cuando en realidad no lo es, o si usted quiere, su complejidad es relativa.

Durante el desarrollo del curso se ha intentado abordar los conceptos desde una perspectiva divulgativa, acompañando las explicaciones de símiles cotidianos que ayuden a comprender ciertos aspectos de la materia. Esto no siempre ha sido posible y algunas partes del curso requieren más esfuerzo por parte del alumno. En cualquier caso el Curso SQL desde cero trata el lenguaje SQL con suficiente amplitud como para que al finalizarlo el alumno tenga una base solida de la materia.
Tómeselo con paciencia y lea las lecciones con calma y asimilando lo que se explica. Intente desarrollar los ejercicios sin mirar las soluciones de entrada y solo hágalo después de realizarlos, encuentre o no una solución. Es muy importante que trabaje los conceptos con ayuda de los ejercicios. Equivocarse le indicará cuales son sus carencias y le permitirá mejorar, equivocarse es recorrer el camino. Hallar la solución es solo la confirmación de que ha asimilado lo que se pretendía, es el final del camino.

Agradecería su participación para que, con sus comentarios y aportaciones, pueda orientarme sobre las carencias, ambigüedades o discrepancias de los contenidos. **Puede plantear dudas o sugerencias** en la página web deletesql.com. Intentaré responder a todos en la medida de lo posible. Espero que disfrute tanto realizando los ejercicios como he disfrutado yo preparándolos. Si además logra comprender buena parte de lo que aquí se expone e iniciarse en el lenguaje SQL, entonces se habrá alcanzado el objetivo principal de este curso.

Gracias y buena suerte.

Lección 1 - Bases de datos relacionales

Base de datos:

Las bases de datos existen desde que el ser humano empezó a almacenar datos en algún soporte. Si por datos entendemos dibujos, que lo son, entonces las primeras bases de datos fueron las paredes de las cuevas donde nuestros ancestros dibujaron las pinturas rupestres.

Posteriormente los egipcios crearon grandes estructuras arquitectónicas que usaron, entre otras cosas, como soporte para almacenar datos y narrar la historia del antiguo Egipto en sus paredes. El tiempo transcurrió hasta el punto de que el significado de todos esos símbolos se perdió, sin embargo la base de datos perduró lo suficiente para que alguien consiguiera descifrar los jeroglíficos a tiempo, de modo que todos esos datos, esa faraónica base de datos, cobró de nuevo todo su sentido. De hecho el valor de toda esa información es mayor que todos los tesoros que pudiesen esconder tumbas y templos. Los arqueólogos esperan encontrar en los nuevos hallazgos, antes que objetos y tesoros, nuevos jeroglíficos que les permitan conocer algún episodio olvidado de la historia de esta fascinante civilización. En ocasiones es esa misma información la que proporciona las pistas para descubrir nuevos hallazgos.

En la actualidad las bases de datos informáticas han quitado todo el protagonismo a sus antecesoras, los archivos de papel, que aun se siguen usando en algunos ámbitos concretos. De bases de datos informáticas han habido de varios tipos, pero las que más han proliferando son las que se tratarán en este curso, las bases de datos relacionales. Mencionar que antes de estas últimas se usaron las bases de datos jerárquicas y posteriormente las bases de datos en red, actualmente sistemas en desuso.

Para encauzar el aprendizaje del lenguaje de consulta SQL empezaremos por conocer la estructura de almacenamiento que usa una base de datos relacional. En este caso no son paredes, ni montones de papel lo que se usa para almacenar la información, sino que se almacena en soportes informáticos bajo una estructura lógica de almacenamiento, como la tiene un archivo de papel, por ejemplo: edificio, planta, pasillo, ubicación, ficha. De este modo es posible recuperar la información que interesa de un modo ágil, gracias a los incides y la estructura organizada del archivo. A continuación se verá como estructura la información una base de datos relacional, pero antes, establezcamos una pocas definiciones.

Base de datos relacional:

Una base de datos (BD), o mejor dicho, un sistema gestor de bases de datos (SGBD), es un software que gestiona una o más bases de datos y nos permite explotar los datos almacenados en ellas de forma relativamente simple mediante SQL.

Esta es una definición muy simplificada, pero para que el aprendizaje sea distendido lo supondremos así, de ese modo podemos centrarnos en aprender como y con que propósito accedemos a los datos, dejando para el final como creamos, alimentamos o modificamos la BD.

Algunos ejemplos de SGBD son: Oracle, MySQL, MS SQL Server…

> **En este curso se empleará un SGBD MySQL**, de modo que los ejemplos y ejercicios están diseñados para MySQL, y el banco de pruebas, accesible desde la web deletesql.com, accede a una base de datos MySQL. **No se debe confundir con un curso para MySQL**, no lo es, aplicar lo aprendido a uno u otro SGBD será cuestión únicamente de conocer la sintaxis de cada sistema y sus funcionalidades para interactuar con sus bases de datos.
> Por ejemplo, si usted realiza un curso para escritores en castellano, donde aprende técnicas y trucos para escribir un thriller, es de esperar que no tenga que realizar el mismo curso en francés porque desea escribir su thriller en francés, para ello bastará con que sepa usted francés. Afortunadamente el estándar SQL empleado por los distintos SGBD es muy similar y en muchas cosas idéntico, no comparable a las diferencias que encontramos entre dos idiomas como puedan ser el castellano y el francés.

Estructura mínima de almacenamiento:

- Tabla:

  Objeto de almacenamiento perteneciente a una BD. Es una estructura en forma de cuadrante donde se almacenan registros o filas de datos. Cada tabla tiene un nombre único en la BD.

- Registro:

  Cada una de las filas de una tabla, está compuesto por campos o atributos.

- Campo:

  Cada uno de los “cajoncitos” de un registro donde se guardan los datos. Cada campo tiene un nombre único para la tabla de la cual forma parte, además es de un tipo (naturaleza) determinado, por tanto no podemos guardar limones en el cajón de las naranjas, en términos informáticos y a modo de ejemplo, no encontraremos un dato alfanumérico (letras y números) en un campo diseñado para guardar datos numéricos. Dedicaremos una lección a los tipos de datos más adelante.

Por el momento estas son las definiciones que necesitamos, veamos ahora un ejemplo concreto de tabla.

Ejemplo de tabla:

Tabla EMPLEADOS

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
2	Elena	Rubio Cuestas	1978-09-25	M	Secretaria	1300
3	José	Calvo Sisman	1990-11-12	H	Mozo	1400
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

Cada registro o fila de datos contiene información de un empleado. En el ejemplo observamos que la tabla tiene un diseño de siete campos y que almacena cuatro registros. El nombre de cada campo viene dado por la fila de encabezado. El dato que contiene el campo ID_EMPLEADO identifica cada registro, pero por ahora no le demos importancia a esto.

Los registros o miembros de una tabla tienen en común sus atributos, no el dato en sí, que lo más probable es que difiera de un registro a otro, pero sí el hecho de que todos ellos poseen esos atributos. En el ejemplo los miembros de la tabla EMPLEADOS tiene en común que todos ellos son personas empleadas en una empresa, que tienen un nombre y un salario, una fecha de nacimiento, etc... Por lo tanto las tablas de una BD guardan información de individuos o unidades de una misma naturaleza con una serie de atributos en común.

*** * ***

Resumen:

Una BD contendrá tablas que a su vez contendrán registros y en estos se encontrarán los datos distribuidos en una serie de campos. Cada registro de la tabla guarda la información particular de una unidad o miembro de un mismo grupo. El SGBD cumple la función de interface entre el usuario y la BD, permitiéndonos interactuar con ella mediante SQL.

Lección 2 - Consultas I (SQL SELECT FROM WHERE)

Si este fuese un manual de SQL, o alguno de los muchos cursos SQL que se pueden encontrar por la web, ahora tocaría abordar la instalación de un SGBD para posteriormente crear una BD con algunas tablas de ejemplo con las que empezar a trabajar. Pero este curso pretende ser diferente. Si ahora se expusiera lo antes mencionado, corremos el riesgo de que usted pierda el interés por la materia, además, el SQL no incluye la instalación del SGBD ni la creación de tablas. Si usted, por ejemplo, desea aprender electricidad, ¿a caso ha de fabricar bombillas, cables y el generador eléctrico con el que poder trabajar? Obviamente no.

En este curso la BD y las tablas con las que trabajar las tiene accesibles mediante el banco de pruebas (disponible en la web: http://deletesql.com), así que empezaremos directamente por lo que será la tónica de este curso: las consultas SQL, que es digamos donde está la miga, y no será hasta el final del curso que se verá la creación de tablas y como modificar la información. A fin de cuentas no tiene demasiado sentido aprender a crear tablas cuando aun no sabe que hacer con ellas.

Consultas SQL

Abordemos las consultas SQL con un caso práctico. Sobre la tabla EMPLEADOS se plantea la siguiente cuestión:

¿Qué empleados tienen un salario mayor a 1350?

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
2	Elena	Rubio Cuestas	1978-09-25	M	Secretaria	1300
3	José	Calvo Sisman	1990-11-12	H	Mozo	1400
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

La respuesta es simple: José y Carlos tiene un salario mayor a 1350, pero si tuviésemos 500 empleados nos llevaría más tiempo responder, y al final tampoco tendríamos la certeza de no habernos equivocado. El SQL nos permite responder estas preguntas de forma rápida y fiable, salvo error al construir la consulta o errores en los propios datos.

Vamos pues a construir la consulta que nos permita responder a esta cuestión.

Preguntas de Construcción

Para construir una consulta SQL debemos hacernos como mínimo tres preguntas:

Primero hemos de preguntarnos: ¿qué datos nos están pidiendo?
En este caso, el nombre y los apellidos de los empleados.

Lo siguiente que nos preguntamos es: ¿dónde están esos datos?
Obviamente están en la tabla empleados.

Y por último: ¿qué requisitos deben cumplir los registros?
En este caso, que el sueldo del empleado sea superior a 1350.

Vamos a suponer por un momento que el SGBD, que es quien intermedia entre el usuario y la BD, fuese una persona; la BD un archivo de papel, y el jefe pide lo siguiente: "Necesito saber ¿qué empleados cobran más de 1350 euros? Usted, que conoce bien el archivo(tablas) y que datos contiene la ficha de un empleado (campos de la tabla EMPLEADOS), mentalmente se hace las preguntas de construcción y le dice a su ayudante que siempre espera ordenes concretas:

Selecciónname el NOMBRE y los APELLIDOS
del archivo EMPLEADOS
cuyo SALARIO sea mayor a 1350

El ayudante sirve la petición y se la entrega para que finalmente usted se la facilite a su jefe. ¿Qué papel ocuparían hoy en una empresa moderna? El jefe sigue siendo el jefe, eso está claro, pero usted ha pasado a ser el informático, y su ayudante el SGBD.

Sintaxis SQL

En SQL la forma de operar es parecida, esta información se obtiene mediante la siguiente consulta:

CÓDIGO:
```
select NOMBRE , APELLIDOS
  from EMPLEADOS
 where SALARIO > 1350
```

“

Obsérvese que en la consulta los nombres de los objectos de base de datos (tabla y campos) los escribimos en mayúsculas, mientras que para las palabras reservadas de la consulta SQL (select, from, where) lo hacemos en minúsculas; esto tiene únicamente un propósito estético, con intención de hacer el código más ordenado y legible. Puede no ser así siempre, dependiendo del SGBD y/o del sistema operativo(windows, linux, ...) donde trabaje el SGBD, este puede ser sensible a mayúsculas y minúsculas.

Y el resultado que nos devuelve el SGBD es:

NOMBRE	APELLIDOS
Carlos	Jiménez Clarín
José	Calvo Sisman

Parecido a lo que nos entregaría nuestro ayudante en el archivo, una lista con la respuesta o solución a la cuestión planteada. Como ve, tanto el modo de solicitar la información al SGBD, sea clásico o informatizado, como el modo en que este nos muestra la información, son muy similares, al menos en este caso. No podemos afirmar lo mismo del tiempo en que uno y otro tardan en facilitarnos la información solicitada.

Forma general

En general una consulta SQL simple tendrá la siguiente forma:

CÓDIGO:
```
select CAMPOS(separados por comas)
  from TABLA
 where CONDICION
```

El SQL permite al usuario desentenderse de como el SGBD ejecuta la consulta, al igual que usted se desentiende de como su ayudante en el archivo de papel se las ingenia para facilitarle la información. Usted esperará pacientemente tras el mostrador a que su ayudante prepare su pedido y le entregue los datos. Dicho de otro modo, basta con saber como pedir la información y no como proceder a reunirla.

*** * ***

Resumen

Hemos visto como construir una consulta SQL simple y concreta, que nos da la solución a una cuestión concreta.

Se ha definido la forma general de una consulta SQL simple:

CÓDIGO:
```
select CAMPOS(separados por comas)
  from TABLA
 where CONDICION
```

Destacar también la utilidad de las preguntas de construcción para ayudarnos a construir la consulta.

1. ¿Qué datos nos piden?
2. ¿Dónde están los datos?
3. ¿Qué requisitos debe cumplir los registros?

En una empresa moderna un informático cumple la función de encargado del archivo, y sus ayudantes son hoy los sistemas informatizados.

*** * ***

Ejercicio

Intente hallar una consulta que devuelva el nombre, apellidos y la fecha de nacimiento de aquellos empleados que cobren más de 1350 euros.

Lección 3 - Consultas II (SQL SELECT FROM WHERE)

En la lección anterior hemos construido con éxito nuestra primera consulta:

CÓDIGO:
```
select NOMBRE , APELLIDOS
  from EMPLEADOS
 where SALARIO > 1350
```

Las tres cláusulas y las preguntas de construcción

Fijémonos ahora en las tres cláusulas de la anterior consulta SQL y que relación guardan con las preguntas de construcción:

1. Cláusula SELECT: Donde indicamos los campos de la tabla que queremos obtener, separados por comas. Responde a la pregunta: ¿Qué datos nos piden?
2. Cláusula FROM: Donde indicamos en que tabla se encuentran estos campos. Responde a la pregunta: ¿Dónde están los datos?
3. Cláusula WHERE: Donde establecemos la condición que han de cumplir los registros de la tabla que serán seleccionados. Responde a la pregunta: ¿Qué requisitos deben cumplir lo registros? Es de hecho donde se establece el filtro de registros, es decir, que registros serán considerados para mostrar sus datos y cuales no.

Modificando la cláusula where

Imaginemos ahora la siguiente cuestión: ¿Qué empleados tienen un sueldo comprendido entre 1350 y 1450?

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
2	Elena	Rubio Cuestas	1978-09-25	M	Secretaria	1300
3	José	Calvo Sisman	1990-11-12	H	Mozo	1400
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

Si nos hacemos las preguntas de construcción:

1. ¿Qué datos nos piden?
2. ¿Dónde están los datos?
3. ¿Qué requisitos deben cumplir los registros?

Observamos que para las dos primeras preguntas las respuestas son idénticas a la anterior cuestión, pero para la tercera es distinta. Esto nos indica que las cláusulas

SELECT y FROM no van a cambiar respecto a la anterior consulta, y sólo lo hará la cláusula WHERE, así que podemos tomar la anterior consulta como patrón y modificarla para adaptarla a lo que se nos pide ahora.

Consulta patrón:

CÓDIGO:
```
select NOMBRE , APELLIDOS
  from EMPLEADOS
 where SALARIO > 1350
```

Antes el salario debía ser mayor a 1350, ahora debe estar comprendido entre 1350 y 1450, ambos inclusive. La cláusula WHERE la construiremos de la siguiente manera:

CÓDIGO:
```
where SALARIO >= 1350 and SALARIO <= 1450
```

Y se lee así: donde el salario sea mayor o igual a 1350 y menor o igual a 1450.

La consulta quedaría:

CÓDIGO:
```
select NOMBRE , APELLIDOS
  from EMPLEADOS
 where SALARIO >= 1350 and SALARIO <= 1450
```

Si comparamos las dos consultas, se observa como únicamente difieren en la cláusula where.

El operador between

Existe otro modo de obtener la mismo resultado aprovechando más los recursos del SQL mediante el operador BETWEEN (entre). La consulta es equivalente y quedaría de la siguiente manera:

CÓDIGO:
```
select NOMBRE , APELLIDOS
  from EMPLEADOS
 where SALARIO between 1350 and 1450
```

Es decir: donde el salario esté entre 1350 y 1450 ambos inclusive.

Y el resultado que nos devuelve el SGBD es:

NOMBRE	APELLIDOS
José	Calvo Sisman

El motor SQL

Aunque en la lección anterior se dijo que el SQL nos permite desentendernos de como se reúnen los datos, en un futuro nos vendrá bien entender de forma lógica su manera de proceder. Veamos como ejecuta esta consulta el motor SQL del SGBD. Primero seleccionará los registros que cumplen la condición de la cláusula WHERE, para ello debe recorrer todos los registros de la tabla y decidir, en función de la condición, si lo toma en consideración o no. Al final reunirá los campos indicados en la cláusula SELECT de la tabla indicada en la cláusula FROM cuyos registros han sido seleccionados por la clausula WHERE.

*** * ***

Resumen

Hemos dividido una consulta SQL concreta en tres cláusulas, se ha relacionado cada cláusula con las preguntas de construcción, hemos tomado como patrón una consulta para modificarla y adaptarla a los nuevos requisitos.

*** * ***

Ejercicio

Intente hallar una consulta que devuelva el nombre y apellidos de los empleados que cobren menos de 1350 euros.

Lección 4 - Tipos de dato

Como ya se dijo la tónica de este curso va a ser selects y más selects, sin embargo antes de seguir desarrollando las consultas SQL debemos hacer un alto en el camino y hablar de los **tipos de dato**, asunto que trataremos a continuación, y de los **operadores**, que veremos en la próxima lección. Es importante entender bien estos conceptos.

¿Qué es un tipo de dato?

Cuando usted durante su infancia jugaba a encajar piezas en un juego similar a este,

una de las cosas que aprendía era diferenciar tipologías de estructuras y encajarlas en una ubicación compatible. A base de prueba y error finalmente lograba completar el rompecabezas, y tras muchas sesiones el juego era trivial y su ejecución evidente, usted había asimilado completamente el concepto que el juego plantea. Podemos decir que usted aprendió a asociar tipos de objetos con tipos de ubicación después de entender que, un tipo de objeto concreto sólo encaja en su tipo de ubicación correspondiente.

Los tipos de dato son como este juego de niños, cualquier dato es de un tipo concreto, como las piezas del juego, y cuando diseñamos una tabla, o un impreso de papel, diseñamos los campos para ubicar un tipo de dato concreto.

Otro aspecto a tener en cuenta es el tamaño del dato, si al niño que juega con las piezas se las cambiamos por piezas de mayor tamaño, pero las ubicaciones las mantenemos igual, le crearemos un conflicto. De pronto lo que funcionaba ya no funciona. El niño ejecuta como antes el procedimiento exitoso, pero por mucho que se esfuerza las piezas no logran encajar. Si el niño es muy bruto y la diferencia de tamaño entre el objeto y su ubicación es suficientemente pequeña, las piezas pueden llegar a encajar, como pasa con los impresos donde, por ejemplo, en un campo diseñado para albergar treinta letras como máximo, siempre hay quien logra escribir treinta y dos, pero en una BD relacional esto es mucho más cerrado, no se puede calzar un cuarenta en un zapato del treinta y nueve, el SGBD devuelve un error y la operación es rechazada.

Así pues diremos que cualquier dato es de un tipo concreto y tiene un tamaño determinado, en consecuencia cuando diseñamos una tabla, diseñaremos sus campos para ubicar un tipo de dato concreto y estableceremos un tamaño máximo para cada campo.

Tipos de dato

Tipo de dato es un concepto propio de la informática, presente en cualquier lenguaje de programación, donde cada entorno de programación tiene su modo particular de definirlos. Los hay muy diversos, clasificados por grupos, definidos por el usuario... pero para lo que aquí nos ocupa vamos a considerar solamente estos cuatro tipos de dato:

- **Cadena** (cadena de texto o alfanumérica)
- **Número entero** (sin decimales)
- **Número decimal** (parte entera + parte decimal)
- **Fecha**

Como ya se ha dicho, cada entorno tiene su modo particular de definir los tipos de dato, aunque todos ellos se rigen en mayor o menor medida por un estándar. Veamos como se definen cada uno ellos en MySQL:

- Cadena: **VARCHAR**(tamaño)

  Las cadenas de texto son de tipo VARCHAR, y su tamaño máximo para un campo concreto se especifica indicando su longitud entre paréntesis. Por ejemplo: Al diseñar la tabla EMPLEADOS, se debe valorar que longitud máxima se establece para el campo NOMBRE, de manera que pueda albergar cualquier nombre de persona que se pueda dar. En el caso de la tabla EMPLEADOS se decidió que como mucho un nombre no sobrepasa las 30 letras o caracteres, de modo que el tipo de dato para este campo se definió como: VARCHAR(30). Es decir, en el campo NOMBRE de la tabla EMPLEADOS se puede guardar cualquier cadena alfanumérica de hasta 30 caracteres. Fíjese que al hablar de tamaño máximo estamos hablando de la ubicación para muchos posibles datos, no del dato en sí.

Cuando indicamos constantes en una consulta SQL, **las cadenas de texto** a diferencia de los números, y ahora estamos hablando del dato en sí y no del tipo, **siempre se entrecomillarán** para indicar al SGBD que se trata de un dato de tipo cadena y no de un dato de tipo número. Esto es muy importante y es necesario que quede bien entendido. Por ejemplo, supongamos que queremos saber ¿qué salario tiene Elena Rubio Cuestas?, un modo de hacerlo es realizar una consulta SQL que filtre por los apellidos, para tal propósito estableceremos una constante alfanumérica en la clausula WHERE exigiendo que sea igual al campo APELLIDOS, y esta deberá ir entrecomillada.

CÓDIGO:

```
select SALARIO
  from EMPLEADOS
 where APELLIDOS = 'Rubio Cuestas'
```

Fíjese que ahora no hablamos de tamaño máximo, en todo caso podemos hablar del tamaño concreto del dato. En el caso de la constante 'Rubio Cuestas' su tamaño es 13.

De hecho existen más tipos de dato para cadenas, cada uno con diferentes prestaciones, pero con el tipo VARCHAR nos sobra por ahora, así que asumiremos que es el único que existe.

- Número entero **INT**:

  A diferencia del tipo VARCHAR, donde establecemos el tamaño máximo, para los números enteros existen varios tipos de dato de tamaño fijo, elegiremos uno u otro en función del tamaño máximo que necesitamos establecer. Cuando tratamos números es más correcto hablar de rango que de tamaño, es decir, bajo que rango de valores(máximo y mínimo) podemos operar con ese tipo. Como hicimos con el tipo cadena, nos quedaremos con uno para simplificar: el tipo INT.

  El valor máximo y mínimo que puede alcanzar un número de este tipo es suficientemente alto y bajo como para no preocuparse por ello en el contexto de este curso. Recuerde que a diferencia de las cadenas NO debe entrecomillarse en las Consultas SQL cuando aparezca como constante.

  Ejemplo de número entero: 3467

- Número decimal **FLOAT** (coma flotante):

  Para los números decimales también existen varios tipos de dato con diferente rango de valores posibles, la parte entera la separamos por un punto de la parte decimal. Asumiremos que solo existe el tipo FLOAT, con un rango de valores posibles suficientemente amplio. Y como aunque decimal no deja de ser un número, NO debe entrecomillarse en las Consultas SQL.

Ejemplo de número decimal: 3467.00

- Fecha: **DATE**

  El tipo DATE tiene el tamaño apropiado para registrar un dato de: año + mes + día. Existe también el tipo DATATIME, por ejemplo, que además del día registra la hora, y algunos más que no vamos a considerar por ahora.

  Un dato de tipo DATE y/o DATETIME se expresa en forma de cadena con un formato determinado, de modo que quien procesa ese dato sabe cual es el año, el día o el mes en función de la posición que ocupa en la cadena alfanumérica.

  Por ejemplo, tenemos el siguiente dato: 4 de noviembre de 2006, para expresar este dato debemos hacerlo de la siguiente forma: 'aaaammdd' donde aaaa indica cuatro dígitos para el año, mm dos dígitos para el mes y dd dos dígitos para el día. De modo que el 4 de noviembre de 2006 lo expresaríamos así: '20061104'.

  Al expresarse en forma de cadena, o si usted quiere, como una cadena con un formato concreto y sus posibles valores restringidos a una fecha valida, debe siempre entrecomillarse cuando aparece como constante en una consulta SQL.

  Existen otros modos de trabajar con datos de tipo DATE, pero por el momento supondremos que es el único modo que tenemos.

Ejemplo

Para finalizar la lección echemos un vistazo a los tipos de dato, bajo la columna titulada Type, de cada uno de los campos de la de la tabla EMPLEADOS:

FIELD	TYPE	NULL	KEY	DEFAULT	EXTRA
ID_EMPLEADO	int(10) unsigned	NO	PRI		
NOMBRE	varchar(30)	NO			
APELLIDOS	varchar(50)	NO			
F_NACIMIENTO	date	NO			
SEXO	varchar(1)	NO			
CARGO	varchar(30)	NO			
SALARIO	float unsigned	NO			

No confundir int(10) con número entero de 10 dígitos, el 10 entre paréntesis no expresa el rango de valores posibles, esto es una particularidad de MySQL y tiene que ver con el formato del número para impresión y no con su tamaño, como ya se dijo el rango de valores para el tipo INT es fijo y va implícito en el propio tipo de dato.

*** * ***

Resumen
Cualquier dato es de un tipo concreto y tiene un tamaño determinado, en consecuencia cuando diseñamos una tabla, diseñaremos sus campos para ubicar un tipo de dato concreto y estableceremos un tamaño máximo para cada campo.

El tamaño máximo o rango de valores puede ir implícito en el propio tipo de dato, como es el caso del tipo INT, o bien debe especificarse en tiempo de diseño como ocurre con el tipo VARCHAR.

Los datos de tipo VARCHAR o cadenas de texto van siempre entrecomillados, a diferencia de los números(INT y FLOAT).

Las fechas o datos de tipo DATE, se expresan en forma de cadena con un formato determinado, concretamente: 'aaaammdd' donde aaaa es el año, mm es el mes y dd es el día. Al ser cadenas deben siempre entrecomillarse.

*** * ***

Ejercicio 1

Defina de que tipo de dato crearía los campos, y su tamaño mínimo si se tercia, para albergar los siguientes datos:

- 'Hola mundo'
- 9.36
- 4564
- 'Esto es un ejercicio de tipos de datos'
- 8 de enero de 1998

Ejercicio 2

Formatee en una cadena, según se ha visto en esta lección, las siguientes fechas.

- 23 de agosto de 1789
- 8 de enero de 1998

Lección 5 - Operadores (SQL WHERE)

Un operador, como su propio nombre indica, es quien opera, normalmente entre dos operandos, estableciendo una operación que al ejecutarla se obtiene un resultado.

Por ejemplo en matemáticas: 3 + 4

"+" es el operador y, "3" y "4" son los operandos. el resultado de esta operación es 7.

> “
>
> A modo de apunte diremos, aunque poco tenga que ver con esta lección, que el SQL nos permite calcular ciertas operaciones matemáticas tanto en la clausula SELECT, para obtener resultados, como en la cláusula WHERE, para establecer condiciones:
>
> **CÓDIGO:**
> ```
> select 3 + 4
> ```
>
> A la pregunta: ¿Qué empleados tienen un salario mayor de 1350?, podríamos construir la consulta SQL así, sustituyendo 1350 por: 1300 + 50.
>
> **CÓDIGO:**
> ```
> select NOMBRE , APELLIDOS
>   from EMPLEADOS
>  where SALARIO > 1300 + 50
> ```

Lógica booleana

Lo que se pretende abordar en esta lección principalmente es la lógica booleana, que es la que nos permite establecer condiciones. Advierto que esta lección puede resultar un poco dura, si tiene dificultades para entender lo que trataremos, no se preocupe e intente quedarse con la idea de fondo.

Cuando usted jugaba al ya clásico juego de ¿quién es quién?,

usted estaba aplicando lógica booleana para descubrir el personaje que su contrincante escondía. Usted lanzaba preguntas con el propósito de saber si una característica del personaje misterioso era cierta o falsa, y si era cierta, descartaba los personajes que no presentaban esa característica. El SGBD hace algo muy parecido, usted establece una característica en la clausula WHERE de la consulta SQL, y el SGBD descarta los registros en que la característica es falsa, mientras que mantiene los que es cierta.

Expresiones booleanas

Si usted jugando a ¿Quién es quién? preguntaba a su contrincante: ¿tiene barba?, lo que en realidad estaba preguntando era: ¿EL PERSONAJE MISTERIOSO tiene BARBA? pues bien, esto es una expresión booleana donde:

"tiene" es el operador, "EL PERSONAJE MISTERIOSO" es un operando variable, que depende del personaje elegido por su contrincante, y "BARBA" es un operando constante. El resultado de aplicar esta expresión a un personaje concreto dará cierto si el personaje tiene barba, y falso en caso contrario.

Cuando en la primera consulta de este cuso:

CÓDIGO:
```
select NOMBRE , APELLIDOS
  from EMPLEADOS
 where SALARIO > 1350
```

indicábamos en la clausula WHERE: SALARIO > 1350(¿es SALARIO mayor a 1350?), estábamos estableciendo una expresión booleana donde:

">" es el operador, "SALARIO" es un operando variable, que tomará valores de cada registro de la tabla EMPLEADOS, y "1350" es un operando constante. El resultado de esta

expresión depende del valor que tome la variable SALARIO, pero en cualquier caso sólo puede dar dos posibles resultados, o cierto o falso. Por lo tanto diremos que una expresión booleana sólo tiene dos posibles resultados.
El motor SQL evalúa la expresión booleana de la clausula WHERE para cada registro de la tabla, y el resultado determina si el registro que se está tratando se tomará en consideración. Lo hará si el resultado de evaluar la expresión es cierto, y lo ignorará si el resultado es falso.

Ejemplo de expresiones booleanas:

- "4 > 3" : ¿es cuatro mayor que tres?
- "3 = 12" : ¿es tres igual a doce?

> “
>
> Fíjese que los operandos son del mismo tipo, en este caso tipo INT. Sin embargo el resultado obtenido no es un dato de tipo INT, sino booleano, sus posibles valores son cierto o falso.
>
> (4 > 3) = cierto
> (3 = 12) = falso

Operadores

Algunos de los operadores que nos permiten construir expresiones booleanas son:

- **>** : "A > B" devuelve cierto si A es estrictamente **mayor que** B, de lo contrario devuelve falso.
- **<** : "A < B" devuelve cierto si A es estrictamente **menor que** B, de lo contrario devuelve falso.
- **=** : "A = B" devuelve cierto si A es **igual a** B, de lo contrario devuelve falso.
- **>=** : "A >= B" devuelve cierto si A es **mayor o igual a** B, de lo contrario devuelve falso.
- **<=** : "A <= B" devuelve cierto si A es **menor o igual a** B, de lo contrario devuelve falso.
- **!=** : "A != B" devuelve cierto si A es **distinto a** B, de lo contrario devuelve falso.

Al construir expresiones con estos operadores, los dos operandos deben ser del mismo tipo, ya sean números, cadenas o fechas.

Ejemplo de expresión booleana con cadenas: ('Aranda, Pedro' < 'Zapata, Mario') = cierto, puesto que por orden alfabético 'Aranda, Pedro' está posicionado antes que 'Zapata, Mario' , por lo tanto es menor el primero que el segundo.

Operadores lógicos

Los operadores lógicos permiten formar expresiones booleanas tomando como operandos otras expresiones booleanas. Fíjese que en las expresiones vistas anteriormente, los operandos debían ser números, cadenas o fechas, ahora sin embargo los operandos deben ser expresiones booleanas, el conjunto forma una nueva expresión booleana que, como toda expresión booleana, dará como resultado cierto o falso.

- **AND** : "A and B" devuelve cierto si A y B valen cierto, y falso en cualquier otro caso.
- **OR** : "A or B" devuelve cierto si A o B valen cierto, y falso únicamente cuando tanto A como B valen falso.
- **NOT** : "not A" devuelve falso si A vale cierto, y cierto si A vale falso.

Veamos una aplicación en el mundo cotidiano.

Supongamos el siguiente anunciado:

Mi jefe quiere contratar a una persona para repartir genero, solamente pueden optar a la vacante aquellos candidatos que tengan vehículo propio y licencia de conducir automóviles. Como candidatos tenemos a Ángela, que tiene licencia pero no tiene vehículo. A Salva, que tiene licencia y vehículo. Y a Teresa, que tiene vehículo pero de momento no tiene licencia. ¿Quiénes pueden pasar al proceso de selección?

Convertimos el anunciado en una expresión booleana:

Sea C: pasa al proceso de selección.
Sea A: tiene vehículo propio.
Sea B: tiene licencia de conducir automóviles.

Entonces para que un candidato satisfaga C, se debe dar A y B:
C = A and B

Resolvamos la expresión para cada candidato:

Aplicado a Ángela: **C** = (A and B) = (falso and cierto) = **falso**. Luego no pasa al proceso de selección.
Aplicado a Salva: **C** = (A and B) = (cierto and cierto) = **cierto**. Luego pasa al proceso de selección.
Aplicado a Teresa: **C** = (A and B) = (cierto and falso) = **falso**. Luego no pasa al proceso de selección.

Veamos ahora esto mismo aplicado al SQL:

Consideremos ahora la tabla PERSONAS, donde hemos guardado una "S" en el campo RUBIA si la persona es rubia y una "N" en caso contrario, análogamente se ha aplicado el mismo criterio para ALTA y GAFAS, es decir, para indicar si es alta y si lleva gafas.

ID_PERSONA	NOMBRE	RUBIA	ALTA	GAFAS
1	Manuel	S	S	N
2	María	N	N	S
3	Carmen	S	N	S
4	José	S	S	S
5	Pedro	N	S	N

El operador AND

Como ya hemos dicho el operador AND devuelve cierto si ambas expresiones son ciertas, y falso en cualquier otro caso. Supongamos que queremos saber ¿qué personas son rubias y altas?, para ello construimos la siguiente consulta SQL:

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where (RUBIA = 'S') and (ALTA = 'S')
```

Resultado:

NOMBRE
Manuel
José

Evaluar (RUBIA = 'S') da como resultado cierto o falso, al igual que evaluar (ALTA = 'S'), son de echo los dos operandos booleanos del operador AND. Si para un registro ambos son ciertos el resultado es cierto, y se mostrarán los datos de ese registro que indica la clausula SELECT.

En el caso de tener una expresión de la forma: "A and B and D" la expresión se evalúa por partes por orden de aparición:

primero se evalúa (A and B) = E
y finalmente (E and D)

Si todos los operadores de la expresión son AND, entonces todos las expresiones deben valer cierto para que el resultado sea cierto.
Veamos un ejemplo de esto mientras jugamos a ¿Quién es quién?

Usted pregunta:
¿es **rubia**? y la respuesta es **cierto**
¿es **alta**? y la respuesta es **falso**
¿lleva **gafas**? y la respuesta es **cierto**

Por lo tanto debe ser una persona que sea rubia y, no sea alta y, lleve gafas.

¿Quién es el personaje?

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where (RUBIA = 'S') and (ALTA = 'N') and (GAFAS='S')
```

Resultado:

NOMBRE
Carmen

Antes de dejar el operador AND, recordar del modo equivalente y más simplificado que se comentó en la lección 3 en que podemos condicionar un campo a un rango de valores mediante el operador BETWEEN:

CÓDIGO:
```
where SALARIO >= 1300 and SALARIO <=1500
```

que el salario sea mayor o igual a 1300 y menor o igual a 1500

forma equivalente:

CÓDIGO:
```
where SALARIO between 1300 and 1500
```

que el salario este entre 1300 y 1500

El operador OR

Con el operador OR basta que uno de los dos operandos sea cierto para que el resultado sea cierto:

Supongamos que queremos saber las personas que son rubias o bien altas, es decir, queremos que si es rubia la considere con independencia de su altura, y a la inversa, también queremos que la seleccione si es alta independientemente del color de pelo. La consulta sería la siguiente.

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where (RUBIA = 'S') or (ALTA = 'S')
```

Resultado:

NOMBRE
Manuel
Carmen
José
Pedro

Si todos los operadores de la expresión son OR, por ejemplo "A or B or C", entonces todos las expresiones deben valer falso para que el resultado sea falso, con que solo una valga cierto el resultado es cierto.

Supongamos que quiere seleccionar tres registros concretos de la tabla EMPLEADOS para ver sus datos, le interesan los registros con identificador 1, 2 y 4. Para esta situación debe usar el operador OR, puesto que su consulta debe establecer la condición: que el identificador sea 1, 2 o 4:

CÓDIGO:
```
select *
  from EMPLEADOS
 where ID_EMPLEADO = 1 or ID_EMPLEADO = 2 or ID_EMPLEADO = 4
```

Resultado:

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
2	Elena	Rubio Cuestas	1978-09-25	M	Secretaria	1300
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

“

El asterisco presente en la cláusula SELECT equivale a indicar todos los campos de la tabla.

Fíjese como en la anterior consulta para cualquier registro de la tabla EMPLEADOS que el campo ID_EMPLEADO contenga un valor distinto a 1, 2 o 4, el resultado de evaluar la expresión será falso, puesto que todos las expresiones booleanas darán falso, pero con que una de ellas valga cierto, el registro será seleccionado. De hecho si un mismo campo aparece dos o más veces en expresiones de la cláusula WHERE como en el ejemplo anterior, carece de sentido que el operador sea AND: usted puede esperar que una persona sea rubia o morena, pero no puede esperar que sea rubia y morena. Del mismo modo el identificador de un registro nunca podrá ser 1 y 2, o es 1 o es 2 o es x.

Un modo de simplificar la anterior consulta es mediante la palabra clave IN, donde se establece una lista de valores posibles que debe contener el campo indicado para que el registro sea seleccionado. La palabra clave IN equivale a establecer condiciones sobre un mismo campo conectadas por el operador OR.

CÓDIGO:
```
select *
  from EMPLEADOS
 where ID_EMPLEADO in (1,2,4)
```

Resultado:

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
2	Elena	Rubio Cuestas	1978-09-25	M	Secretaria	1300
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

El operador NOT

Este operador tan solo tiene un operando, el resultado es negar el valor del operando de modo que:

"(4 > 3) = cierto" luego "not (4>3) = falso"

Si negamos dos veces una expresión booleana es equivalente a la expresión original:
"(4 > 3) = cierto" luego "not (not (4>3)) = cierto"

Cuando descubrimos al personaje misterioso, a la pregunta ¿es alta? la respuesta era falso, luego podríamos haber establecido lo siguiente: "not (ALTA = 'S')" en lugar de "(ALTA = 'N')":

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where (RUBIA = 'S') and not (ALTA = 'S') and (GAFAS='S')
```

Resultado:

NOMBRE
Carmen

Otro ejemplo: si negamos toda la expresión de la clausula WHERE, estaremos precisamente seleccionando los registros que antes descartábamos, y al revés, descartando los que antes seleccionábamos.

Tomemos la anterior consulta y neguemos la clausula WHERE, si antes el resultado era: Carmen, ahora el resultado ha de ser todas las persona menos Carmen. Para hacer esto cerramos entre paréntesis toda la expresión y le colocamos el operador "not" delante, de ese modo primero se resolverá lo que está dentro de los paréntesis y finalmente se negará el resultado.

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where not ((RUBIA = 'S') and not(ALTA = 'S') and (GAFAS= 'S'))
```

Efectivamente el resultado es justo lo contrario que antes:

NOMBRE
Manuel
María
José
Pedro

Y aun otro ejemplo de este operador junto la palabra clave IN: tomemos la consulta que seleccionaba tres registros concretos de la tabla EMPLEADOS, y modifiquémosla únicamente incluyendo el operador NOT para que devuelva lo contrario, es decir, todos los registros menos los tres que antes seleccionaba:

CÓDIGO:
```
select *
  from EMPLEADOS
 where ID_EMPLEADO not in (1,2,4)
```

Resultado:

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
3	José	Calvo Sisman	1990-11-12	H	Mozo	1400

El uso de paréntesis

Los paréntesis se comportan como en matemáticas, no es lo mismo "5 + 4 / 3" donde primero se calculará la división y después se sumará 5, que "(5 + 4) / 3" donde primero se

resolverá la suma y el resultado parcial se dividirá por 3. Sin paréntesis la división tiene prioridad sobre la suma, con paréntesis forzamos a que la operación se realice en el orden deseado.
Los operadores AND y OR tienen la misma prioridad de modo que se evalúa la expresión por orden de aparición:

No es lo mismo: "RUBIA and ALTA or GAFAS" = "(RUBIA and ALTA) or GAFAS" que, "RUBIA and (ALTA or GAFAS)". En primer caso estamos diciendo: "que sea rubia y alta, o bien lleve gafas", y en el segundo "que sea rubia y, sea alta o lleve gafas".

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where RUBIA = 'S' and ALTA = 'S' or GAFAS= 'S'
```

Resultado:

NOMBRE
Manuel
María
Carmen
José

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where RUBIA = 'S' and (ALTA = 'S' or GAFAS= 'S')
```

Resultado:

NOMBRE
Manuel
Carmen
José

*** * ***

Resumen

En esta lección se ha descrito como construir expresiones booleanas y como trabajar con ellas para establecer condiciones en la clausula WHERE de una consulta SQL.

Las expresiones booleanas con operadores tales como (> , = , != ...) precisan operandos de tipo número, cadena o fecha, y el resultado de evaluar la expresión devuelve siempre cierto o falso.
Ejemplo: (SALARIO > 1350)

Las expresiones con operadores tales como (AND , OR , NOT) precisan expresiones booleanas como operandos, el conjunto es una nueva expresión booleana que al evaluarla devolverá siempre cierto o falso.
Ejemplo: RUBIA = 'S' and ALTA = 'S'

El uso de paréntesis garantiza que, en una expresión compleja las expresiones simples que la forman se evalúen en el orden que usted desea.
Ejemplo: not ((RUBIA = 'S' or ALTA = 'S') and (ALTA ='N' or GAFAS = 'S'))

*** * ***

Ejercicio 1

Cree una consulta SQL que devuelva las personas que son altas, o bien son rubias con gafas.

Ejercicio 2

Cree una consulta SQL que devuelva los empleados que son mujer y cobran más de 1300 euros.

> “
>
> En la tabla empleados se guarda una "H" en el campo SEXO para indicar que es hombre, o una "M" para indicar que es mujer.

Ejercicio 3

Usando solo expresiones (ALTA = 'S') , (RUBIA = 'S') , (GAFAS = 'S') combinadas con el operador NOT resuelva:

¿Quién es quién? Lleva gafas y no es alta ni rubia.

Ejercicio 4 (optativo)

Suponiendo que A vale cierto y B vale falso, evalúe la siguiente expresión booleana:

C= ((A and B) and (A or (A or B))) or A

Lección 6 - Totalizar datos / Alias de campos (SQL AS)

Dejemos de lado por ahora los tipos de dato y sus operaciones para abordar un recurso del SQL de gran potencia que resulta muy interesante: **Totalizar datos**.
Tal como está planteado este curso, no podría exponerse este concepto de otro modo que no fuese mediante ejemplos prácticos. Supongamos entonces que nos piden lo siguiente:

¿Cuál es el salario medio de los empleados?

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
2	Elena	Rubio Cuestas	1978-09-25	M	Secretaria	1300
3	José	Calvo Sisman	1990-11-12	H	Mozo	1400
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

Bien, si recuperamos las preguntas de construcción que tanto nos ayudan para construir nuestras consultas SQL:

- ¿Qué datos nos piden?
- ¿Dónde están los datos?
- ¿Qué requisitos deben cumplir los registros?

A la pregunta: ¿Dónde están los datos?, se nos plantea una duda. El dato que nos piden es: “el salario medio de los empleados”, pero este dato no está en ninguna tabla, entonces ¿Cómo resolvemos el problema? Afortunadamente el SQL nos permite calcularlo, para calcular el salario medio basta con sumar todos los salarios y dividirlo por el número de empleados. Esto es posible hacerlo mediante la funciones SUM(suma) y COUNT(contar) de SQL.

Obtengamos primero la suma de salarios, para ello nos hacemos las preguntas de construcción:

¿Qué datos nos piden?
La suma de los salarios.

¿Dónde están los datos?
En la tabla empleados.

¿Qué requisitos deben cumplir los registros?
Ninguno, queremos sumar todos los salarios por lo tanto no hemos de filtrar registros, los seleccionaremos todos, o lo que es lo mismo, **omitiremos la cláusula WHERE** de la consulta SQL.

La consulta la construiríamos así:

CÓDIGO:
```
select sum(SALARIO)
  from EMPLEADOS
```

Resultado:

SUM(SALARIO)
5525.5

> “
>
> Fíjese que el resultado de esta consulta SQL devuelve una sola fila. Obsérvese también que el motor SQL debe recorrer toda la tabla para obtener el resultado, puesto que debe sumar todos los salarios.

Análogamente contamos el número de empleados, es decir, el número de registros de la tabla empleados.

CÓDIGO:
```
select count(*)
  from EMPLEADOS
```

Resultado:

COUNT(*)
4

> “
>
> El asterisco que encontramos en "COUNT(*)" equivale a decir: cualquier campo, fíjese que en este caso queremos contar registros, por lo tanto lo mismo nos da contar nombres, que apellidos, que cualquier otro campo. Veremos en otra lección las particularidades de la función COUNT aplicada a un solo campo, por ahora entandamos que "COUNT(*)" cuenta los registros seleccionados.

Ahora ya podemos resolver la cuestión planteada, basta con dividir el primer resultado por el segundo, pero... vallamos más allá. De entrada estamos recorriendo la tabla dos veces, una para sumar los salarios, y otra para contar los empleados, es de esperar que recorriéndola una sola vez el motor SQL sea capaz de reunir ambos datos. Y así es, para ello construimos la siguiente consulta SQL:

CÓDIGO:
```
select sum(SALARIO) , count(*)
  from EMPLEADOS
```

Resultado:

SUM(SALARIO)	COUNT(*)
5525.5	4

Vallamos todavía un poco más allá. En la lección anterior se mencionó, a modo de apunte, que el SQL permite calcular algunas operaciones matemáticas en la cláusula SELECT. No parece descabellado esperar entonces, que pueda dividir estas dos columnas obteniendo, en única columna, el resultado de la división:

CÓDIGO:
```
select sum(SALARIO) / count(*)
  from EMPLEADOS
```

Resultado:

SUM(SALARIO) / COUNT(*)
1381.375

Efectivamente funciona. Con esto queda resuelta la cuestión planteada.

*** * ***

Alias de campo

Este es un buen momento para hacer un paréntesis y explicar los Alias de campo de SQL.

Usted, como informático que además de construir la anterior consulta conoce bien el lenguaje SQL, no tendrá dificultades en interpretar la cabecera de la columna del resultado anterior: “sum(SALARIO) / count(*)”. Pero… estaremos de acuerdo en que no podría entregar a su jefe un informe con semejante encabezamiento. Para solucionar esto el SQL pone a su disposición la palabra clave AS, que permite rebautizar con un alias o sobrenombre las cabeceras de las columnas de resultado:

CÓDIGO:
```
select sum(SALARIO) / count(*) as MEDIA_SALARIOS
  from EMPLEADOS
```

Resultado:

MEDIA_SALARIOS
1381.375

Con ello usted ha conseguido un título mucho más explícito, además de ahorrarse una bronca de su jefe.

En general podemos rebautizar cualquier campo o expresión de la cláusula SELECT, para ello basta con colocar seguido del campo que interese la palabra clave AS, precediendo al ALIAS que se quiere aplicar. Pongamos otro ejemplo: nos piden una consulta que devuelva el nombre, apellidos y sueldo de todos los empleados, pero con los encabezamientos de cada columna en inglés:

CÓDIGO:
```
select NOMBRE as NAME, APELLIDOS as SURNAMES, SALARIO as SALARY
  from EMPLEADOS
```

Resultado:

NAME	SURNAMES	SALARY
Carlos	Jiménez Clarín	1500
Elena	Rubio Cuestas	1300
José	Calvo Sisman	1400
Margarita	Rodríguez Garcés	1325.5

“

En realidad la palabra clave AS es solo un modo de enfatizar que se está renombrando el campo de tabla, puede omitirla y el resultado será el mismo. El motor SQL entiende que si después de un campo en la cláusula SELECT, colocamos una palabra y ambas están separados por un espacio en lugar de una coma, la primera es un campo de tabla y la segunda es su alias o sobrenombre. Pruebe a llevar la anterior consulta al banco de pruebas, accesible desde la web deletesql.com, eliminando antes de ejecutarla las palabras claves AS, dejando como mínimo un espacio entre los campos de tabla y sus alias .

*** * ***

Sigamos con la totalización de datos. Acabábamos de explicar como calcular la media de salarios de la tabla EMPLEADOS. El asunto ha quedado resuelto, pero en realidad nos hemos complicado la vida sobremanera. Se ha hecho así adrede, con el propósito de ver más recursos del lenguaje y explicar mejor la totalización de datos. En SQL existe un modo más simplificado para calcular la media de un campo mediante la función AVG (average, término en inglés que significa promedio) Es de hecho una función como SUM o COUNT,

pero con distinta funcionalidad. La consulta es equivalente a la que construimos anteriormente y quedaría de la siguiente manera:

CÓDIGO:
```
select avg(SALARIO) as MEDIA_SALARIOS
  from EMPLEADOS
```

Resultado:

MEDIA_SALARIOS
1381.375

Antes de finalizar debo insistir en que estas funciones no devuelven un dato de la tabla, sino que devuelven un cálculo en función de los datos que contienen los registros seleccionados, dando como resultado una única fila. Por lo que **no tiene sentido mezclar en la clausula SELECT campos de la tabla con funciones de totalización**. Una consulta como la siguiente **no tiene sentido**, y el SGBD devolverá un error:

CÓDIGO:
```
select NOMBRE, avg(SALARIO)
  from EMPLEADOS
```

Si estoy obteniendo un dato calculado sobre un grupo de registros, ¿qué sentido tiene acompañarlo de un dato singular de un solo registro? En la próxima lección abordaremos esto con más detalle pero, veamos como estas situación puede darse fácilmente al malinterpretar este recurso, por ejemplo, supongamos que a usted le piden: ¿qué porcentaje del dinero que desembolsa la empresa percibe cada empleado?

el porcentaje de un empleado = (salario_empleado / total_salarios) x 100

Usted puede pensar incorrectamente en crear la siguiente consulta:

CÓDIGO:
```
select SALARIO / sum(SALARIO) * 100 as PORCENTAJE
  from EMPLEADOS
```

Pero fíjese lo que usted pretende que haga el motor SQL, primero debe obtener la suma de salarios, para ello debe recorrer toda la tabla, acto seguido, con el total de la suma resuelto, debe volver a recorrer la tabla para aplicar la formula a cada registro y obtener así los porcentajes de todos los empleados. Bien, esto no funciona así. Puede ayudarle a no caer en este error el saber que: con una consultas SQL donde sólo interviene una tabla, el motor SQL jamás recorrerá la tabla dos veces para brindarle el resultado.

Resolvamos por partes esta cuestión:

Primero obtenemos el total de sueldos:

CÓDIGO:
```
select sum(SALARIO)
  from EMPLEADOS
```

Resultado:

SUM (SALARIO)
5525.5

Y aplicado a cada empleado obtenemos los porcentajes:

CÓDIGO:
```
select NOMBRE , APELLIDOS , SALARIO / 5525.5 * 100 as PORCENTAJE
  from EMPLEADOS
```

Resultado:

NOMBRE	APELLIDOS	PORCENTAJE
Carlos	Jiménez Clarín	27.146864537146
Elena	Rubio Cuestas	23.5272825988598
José	Calvo Sisman	25.3370735680029
Margarita	Rodríguez Garcés	23.9887792959913

Para concluir veamos las funciones MAX (máximo) y MIN (mínimo), que intuitivamente ya se ve que se utilizan para obtener el valor máximo y mínimo de un campo de entre todos los registros seleccionados. Pero... no voy a poner ejemplos esta vez, le propongo a usted que intente resolver el ejercicio sobre las funciones MAX y MIN que encontrará al final de la lección, puede tomar como patrón las consultas de ejemplo vistas anteriormente donde aparecen las funciones SUM, COUNT o AVG, ya que las funciones MAX y MIN se aplican de igual modo a una consulta SQL.

Usted puede pensar que en este caso la funciones de totalización MAX y MIN sí devuelven un dato de la tabla, y es verdad, devolverán el valor máximo o mínimo del campo que indiquemos a cada una de ellas pero, en realidad no deja de ser un cálculo. El resultado no tiene porque estar vinculado a un solo registro. El motor SQL calcula el valor máximo o mínimo sobre un grupo de registros y usted no puede saber de entrada si ese dato está en uno, dos, o más registros.

Resumen

Las funciones de totalización SUM (suma), COUNT (contar), AVG (promedio), MAX (máximo) y MIN (mínimo), devuelven en una sola fila el cálculo sobre un campo aplicado a un grupo de registros. Los registros que intervienen para el cálculo dependen de los filtros establecidos en la clausula WHERE, interviniendo todos los registros de la tabla si la omitimos.

Podemos realizar varios cálculos sobre el mismo grupo de registros de una sola vez indicando varias funciones separadas por comas en la cláusula SELECT, pero no podemos mezclar en dicha cláusula campos de la tabla con funciones de totalización, puesto que carece de sentido.

El SQL nos permite rebautizar cualquier campo de la consulta por un alias o sobrenombre mediante la palabra clave AS:
select CAMPO1 as ALIAS1 , CAMPO2 as ALIAS2 , ...

*** * ***

Ejercicio 1

En todos los ejemplos de esta lección se ha omitido la clausula WHERE, construya una consulta, donde necesitará establecer una condición en la clausula WHERE, que devuelva el salario medio de los empleados que son hombres. Renombre la cabecera del resultado con un título que deje claro que dato se está mostrando.

Ejercicio 2

Construya una consulta que devuelva en la misma fila el salario máximo y mínimo de entre todos los empleados. Renombre las cabeceras de resultados con un titulo que deje claro que datos se están mostrando.

Ejercicio 3

Construya una consulta que responda a lo siguiente: ¿Que cuesta pagar a todas las mujeres en total? Renombre la cabecera del resultado con un título que deje claro que dato se está mostrando.

Lección 7 - Agrupación de datos (SQL GROUP BY)

En esta lección vamos a continuar hablando de la totalización de datos, ya que esta es un caso particular de la agrupación de datos, es decir, cuando usted totaliza datos de una tabla, en realidad está totalizando datos dentro de un solo y único grupo. De ahí que todos los ejemplos de totalización mostrados en la lección anterior devuelvan una sola y única fila.

Si logró asimilar lo que expone la lección 6, usted puede responder a la pregunta: ¿cuántos empleados hay? La respuesta es en realidad el total de registros de un solo grupo que podemos llamar: empleados. Puede responder también a la pregunta: ¿cuántos hombres hay? La respuesta no deja de ser el total de registros de un solo grupo que podemos llamar: hombres. Y podríamos seguir totalizando datos así, construyendo consultas SQL que devuelven una sola fila con el total de un solo grupo de registros, de manera que para totalizar por ejemplo el número de empleados por edades, deberíamos construir tantas consultas como grupos o rango de edades queramos evaluar. No parece práctico, ¿no sería más razonable construir una sola consulta SQL que en lugar de devolver una sola fila con el total de un solo grupo, devolviese varias filas, una por cada grupo, donde cada fila exprese el total de su grupo?

Supongamos por ejemplo una clase de alumnos, usted puede preguntarse: ¿cuántos alumnos hay?, y la respuesta es un solo dato que hace referencia a un solo grupo que podemos llamar: alumnos. Ahora dividamos la clase en dos grupos de manera que uno lo formen las chicas y otro los chicos. Una vez formados los dos grupos puede preguntarse: ¿cuántos alumnos hay en cada uno?, y la respuesta son dos datos, uno para cada grupo. El resultado es en realidad una totalización por grupos. Primero usted establece los grupos por sexo y luego totaliza la cantidad de miembros de cada uno. Pues bien, el SQL permite agrupar totales mediante la cláusula: GROUP BY.

En la lección anterior se dijo que no tiene sentido acompañar campos de tabla con funciones de totalización en una misma consulta SQL. Esto es así, sin embargo existe una excepción: usted puede acompañar un campo de tabla con funciones de totalización si su propósito es agrupar totales por ese campo, en cuyo caso deberá construir una consulta SQL para ese fin. El número de grupos resultantes dependerá de los distintos valores que existen para ese campo en el grupo de registros seleccionado.

Cláusula GROUP BY

Tenemos nuestra ya conocida tabla EMPLEADOS:

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
2	Elena	Rubio Cuestas	1978-09-25	M	Secretaria	1300
3	José	Calvo Sisman	1990-11-12	H	Mozo	1400
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

Construyamos una consulta que nos devuelva el **total de empleados por sexo**. Esto se consigue con una nueva cláusula: GROUP BY, en consecuencia debemos añadir una cuarta pregunta a las preguntas de construcción:

- ¿Qué datos nos piden?
  El número de empleados.
- ¿Dónde están los datos?
  En la tabla empleados
- ¿Qué requisitos deben cumplir los registros?
  Ninguno, necesitamos que intervengan todos los registros.
- ¿Cómo debemos agrupar los datos?
  Por sexo.

Consulta SQL:

CÓDIGO:
```
select SEXO , count(*) as EMPLEADOS
  from EMPLEADOS
 group by SEXO
```

Resultado:

SEXO	EMPLEADOS
H	2
M	2

“

Observe que el resultado de la consulta devuelve dos filas, una para los hombres y otra para las mujeres, cada fila indica el número de empleados de su grupo. Advierta como los grupos resultantes son dos porque los distintos valores del campo SEXO en los registros seleccionados son dos: "H" y "M".

En general, cuando acompañamos uno o más campos de tabla con funciones de totalización, estos campos deberán formar parte de la cláusula GROUP BY. Un campo por el que agrupamos puede omitirse en la cláusula SELECT, aunque entonces, como puede apreciarse en el próximo ejemplo, ignoramos a que grupo representa cada fila de resultado.

CÓDIGO:
```
select count(*) as EMPLEADOS
  from EMPLEADOS
 group by SEXO
```

Resultado:

EMPLEADOS
2
2

Pero **si un campo aparece en la clausula SELECT junto con funciones de totalización**, entonces debemos forzosamente agrupar por ese campo, o lo que es lo mismo, **debe formar parte de la clausula GROUP BY**.

*** * ***

La palabra clave DISTINCT

Este es un buen momento para hacer un paréntesis en la agrupación de datos y presentar la palabra clave: DISTINCT.
Con ella podemos eliminar filas redundantes de un resultado SQL, por lo que permite obtener los distintos valores de un campo existentes en una tabla o grupo de registros seleccionados.

Por ejemplo, ¿qué valores distintos existen en el campo SEXO de la tabla empleados?:

CÓDIGO:
```
select distinct SEXO
  from EMPLEADOS
```

Resultado:

SEXO
H
M

> “
>
> **sugerencia**: lleve la anterior consulta SQL al banco de pruebas, accesible desde la web deletesql.com, elimine la palabra clave distinct y ejecute la consulta.

Si al hacer DISTINCT en una consulta SQL sobre un campo el motor SQL devuelve por ejemplo dos filas, entonces al agrupar por ese campo un cálculo de totalización también devolverá dos filas, obviamente ambas consultas deben tener la misma cláusula WHERE, es decir, deben operar sobre el mismo grupo de registros.

En general pondremos la palabra clave DISTINCT delante de la lista de campos de la cláusula SELECT para eliminar las filas de resultados duplicadas o redundantes.

CÓDIGO:

```
select distinct campo_1 , campo_2 , ... , campo_n
  from tabla
```

*** * ***

Para seguir viendo el potentísimo recurso SQL que es la agrupación de datos, vamos a suponer que usted gestiona un centro de acogida de mascotas, a él llegan perros y gatos abandonadas o de gente que no puede hacerse cargo. Para cada nueva mascota que llega al centro creamos un nuevo registro en la tabla MASCOTAS. Cuando una mascota es acogida por alguien, damos el registro de baja para indicar que esa mascota ha abandonado el centro.

Diseño de la tabla MASCOTAS:

FIELD	TYPE	NULL	KEY	DEFAULT	EXTRA
ID_MASCOTA	int(11)	NO	PRI		
NOMBRE	varchar(30)	NO			
ESPECIE	varchar(1)	NO			
SEXO	varchar(1)	NO			
UBICACION	varchar(6)	NO			
ESTADO	varchar(1)	NO			

Descripción de los campos:

- **ID_MASCOTA**: Número o identificador de mascota.
- **NOMBRE**: Nombre de la mascota.
- **ESPECIE**: Campo codificado donde se guarda "P" para perro y "G" para gato.
- **SEXO**: Campo codificado donde se guarda "M" para macho y "H" para hembra.
- **UBICACION**: Jaula o estancia donde está ubicada la mascota.
- **ESTADO**: Campo codificado donde se guarda "A" para alta en el centro y "B" para baja en el centro.

Echemos un vistazo a la tabla MASCOTAS:

ID_MASCOTA	NOMBRE	ESPECIE	SEXO	UBICACION	ESTADO
1	Budy	P	M	E05	B
2	Pipo	P	M	E02	B
3	Nuna	P	H	E02	A
4	Bruts	P	M	E03	A
5	Americo	G	M	E04	A
6	Sombra	P	H	E05	A
7	Amaya	G	H	E04	A
8	Talia	G	H	E01	B
9	Trabis	P	M	E02	A
10	Tesa	G	H	E01	A
11	Titito	G	M	E04	B
12	Truca	P	H	E02	A
13	Zulay	P	H	E05	A
14	Dandi	G	M	E04	A
15	Ras	G	M	E01	A
16	Canela	P	H	E02	A

Planteemos la siguiente cuestión: ¿cuántos perros de cada sexo hay en total actualmente en el centro?

Para construir la consulta SQL nos ayudamos de las preguntas de construcción:

- ¿Qué datos nos piden?
  El número de perros.

- ¿Dónde están los datos?
  En la tabla mascotas

- ¿Qué requisitos deben cumplir los registros?
  Deben ser perros y estar de alta en el centro.

- ¿Cómo debemos agrupar los datos?
  Por sexo.

Consulta SQL:

CÓDIGO:
```
select SEXO,count(*) as PERROS_VIGENTES
  from MASCOTAS
 where ESPECIE = 'P' and ESTADO = 'A'
 group by SEXO
```

Resultado:

SEXO	PERROS_VIGENTES
H	5
M	2

El resultado son dos machos y cinco hembras.

*** * ***

Más ejemplos: ¿Cuantos ejemplares contiene actualmente cada jaula o ubicación?

- ¿Qué datos nos piden?
  El número de ejemplares.

- ¿Dónde están los datos?
  En la tabla mascotas

- ¿Qué requisitos deben cumplir los registros?
  las mascotas deben estar de alta en el centro.

- ¿Cómo debemos agrupar los datos?
  Por ubicación.

Consulta SQL:

CÓDIGO:
```
select UBICACION , count(*) as EJEMPLARES
  from MASCOTAS
 where ESTADO = 'A'
 group by UBICACION
```

Resultado:

UBICACION	EJEMPLARES
E01	2
E02	4
E03	1
E04	3
E05	2

“

Obsérvese como en este caso la consulta SQL devuelve cinco filas, o lo que es lo mismo, cinco grupos resultantes. Esto es debido a que el campo UBICACIÓN contiene cinco distintos valores de entre los registros seleccionados.

*** * ***

Veamos ahora un ejemplo donde se agrupa por dos campos. Supongamos la siguiente cuestión: ¿cuántos ejemplares de cada especie, y dentro de cada especie de cada sexo, hay actualmente en el centro?
Para construir la consulta SQL nos ayudamos de las preguntas de construcción:

- ¿Qué datos nos piden?
  El número de ejemplares.
- ¿Dónde están los datos?
  En la tabla mascotas
- ¿Qué requisitos deben cumplir los registros?
  Deben estar de alta en el centro.
- ¿Cómo debemos agrupar los datos?
  Por especie y por sexo.

Consulta SQL:

CÓDIGO:
```
select ESPECIE , SEXO , count(*) as EJEMPLARES_VIGENTES
  from MASCOTAS
 where ESTADO = 'A'
 group by ESPECIE , SEXO
```

Resultado:

ESPECIE	SEXO	EJEMPLARES_VIGENTES
G	H	2
G	M	3
P	H	5
P	M	2

El resultado son dos machos y cinco hembras para los perros, y tres machos y dos hembras para los gatos.

*** * ***

Pongamos otro ejemplo, pero esta vez planteémoslo al revés: ¿Qué devuelve la siguiente consulta SQL?:

Consulta SQL:

CÓDIGO:
```
select UBICACION , ESPECIE , SEXO , count(*) as EJEMPLARES_VIGENTES
  from MASCOTAS
 where ESTADO = 'A'
 group by UBICACION , ESPECIE , SEXO
```

Resultado:

UBICACION	ESPECIE	SEXO	EJEMPLARES_VIGENTES
E01	G	H	1
E01	G	M	1
E02	P	H	3
E02	P	M	1
E03	P	M	1
E04	G	H	1
E04	G	M	2
E05	P	H	2

Observamos que el resultado de la consulta anterior devuelve datos totalizados en tres grupos, responde al número de ejemplares por especie y sexo que hay en cada ubicación.

*** * ***

Resumen

La cláusula GROUP BY permite obtener totales, mediante las funciones de totalización SUM, COUNT, MAX..., por grupos.
Los grupos resultantes dependen de los distintos valores que contengan, de entre los registros seleccionados, el campo o campos por los que se está agrupando . Si por ejemplo estamos agrupando por SEXO, los grupos resultantes serán como máximo dos, a no ser que consideremos hermafrodita como un tercer sexo, en cuyo caso serán tres.

Cuando acompañamos un campo de tabla con funciones de totalización, se debe forzosamente agrupar por ese campo, de modo que el campo debe formar parte de la cláusula GROUP BY.

*** * ***

Ejercicio 1

Construya una consulta que devuelva el salario medio, máximo y mínimo de los empleados agrupado por sexo.

Ejercicio 2

Construya una consulta que devuelva cuantos perros y cuantos gatos han pasado por el centro y ya no están.

Ejercicio 3

Construya una consulta que devuelva cuantos perros macho hay actualmente en el centro agrupado por ubicación.

Ejercicio 4

Con ayuda del filtro DISTINCT, construya una consulta que devuelva las diferentes especies que hay actualmente en cada jaula o ubicación del centro.

Lección 8 - Filtrar cálculos de totalización (SQL HAVING)

Si se plantea la siguiente cuestión: *¿Qué ubicaciones del centro de mascotas tienen más de dos ejemplares?* Usted podría responder a la tercera pregunta de construcción: ¿Qué requisitos deben cumplir los registros?, lo siguiente: "que la ubicación tenga más de dos ejemplares"; y esa respuesta sería errónea. Esta pregunta nos la formulamos para construir la cláusula WHERE y aplicar filtros a los registros de la tabla, pero como el número de ejemplares de cada ubicación no lo tenemos en ninguna tabla, sino que debemos calcularlo, no podemos aplicar ese filtro en la cláusula WHERE. ¿Dónde entonces?, pues obviamente debemos filtrar las filas de resultados, es decir, de todas las filas resultantes ocultar las que no nos interesen y mostrar el resto. Puede verse como un filtro en segunda instancia, una vez el motor a resuelto la consulta y siempre ajeno a la tabla de datos. Para ello existe una nueva cláusula: HAVING, y en consecuencia una nueva pregunta de construcción: ¿qué requisitos deben cumplir los datos totalizados?

Cláusula HAVING

¿Qué ubicaciones del centro de mascotas tienen más de dos ejemplares?
Construyamos la consulta SQL que resuelve la cuestión planteada con ayuda da las preguntas de construcción:

- ¿Qué datos nos piden?
  Las ubicaciones.
- ¿Dónde están los datos?
  En la tabla mascotas.
- ¿Qué requisitos deben cumplir los registros?
  Ubicaciones que contengan mascotas de alta en el centro.
- ¿Cómo debemos agrupar los datos?
  Por ubicación.
- ¿Qué requisito han de cumplir los datos totalizados?
  Que el número de ejemplares de las ubicaciones sea mayor a dos.

Consulta SQL:

CÓDIGO:

```
select UBICACION , count(*) as EJEMPLARES
  from MASCOTAS
 where ESTADO = 'A'
 group by UBICACION
having count(*) > 2
```

Resultado:

UBICACION	EJEMPLARES
E02	4
E04	3

De las cinco ubicaciones que existen en el centro, solo dos cumplen la condición de la cláusula HAVING. Esta cláusula es de hecho como la cláusula WHERE, pero en lugar de filtrar registros de la tabla, filtra filas de resultado en función de las condiciones que establezcamos sobre las columnas de resultado. En realidad este recurso se usa casi exclusivamente para establecer condiciones sobre las columnas de datos totalizados, puesto que los demás valores, los que están en la tabla, los debemos filtrar en la clausula WHERE.

A fin de ser prácticos consideraremos la clausula HAVING como una cláusula WHERE para los cálculos de totalización. De modo que lo que filtraremos aquí serán cosa del estilo: que la suma sea inferior a..., que la media sea igual a..., que el máximo sea superior a...., o como en el ejemplo: que el recuento de registros sea superior a dos. Siempre sobre cálculos de totalización. Por lo tanto si no hay cláusula GROUP BY, tampoco habrá cláusula HAVING.

Diferencia entre WHERE y HAVING

Veamos con más detalle la diferencia entre una y otra cláusula. Cuando el motor SQL recorre la tabla para obtener el resultado, ignora los registros que no satisfacen la cláusula WHERE, en el caso anterior ignora los registros que el campo ESTADO no contenga una 'A', y estos registros no son considerados para desarrollar el cálculo. Una vez el motor SQL a recorrido toda la tabla y ha finalizado el cálculo, de las filas resultantes ocultará las que no satisfacen la cláusula HAVING, por lo que en primer lugar: no se ahorra hacer el cálculo para las filas de resultados no mostradas, de lo contrario no podría saber si cumplen o no la condición de la cláusula HAVING, y en segundo lugar, este filtro se aplica en la fase final del proceso que ejecuta el motor SQL, y siempre sobre las filas de resultados escrutando los datos totalizados (COUNT, SUM, MAX, ...), limitándose a mostrar o no una fila de resultado en función de las condiciones establecidas en dicha cláusula.

Todo lo expuesto sobre lógica booleana en la lección 5 es aplicable a la cláusula HAVING, teniendo en cuenta que lo correcto es establecer condiciones sobre las columnas de totalización, y carece de sentido establecer condiciones que podríamos perfectamente establecer en la cláusula WHERE, puesto que en ese caso estaremos haciendo trabajar al motor SQL en vano, es decir, le estaremos obligando a considerar registros que se podría ahorrar ya que finalmente se ocultará la fila o cálculo referente a ese grupo de registros. Por ejemplo:

La siguiente consulta SQL cuenta los ejemplares de alta de la ubicaciones E02 y E03.

CÓDIGO:
```
select UBICACION , count(*) as EJEMPLARES
  from MASCOTAS
 where ESTADO = 'A' and (UBICACION = 'E02' or UBICACION = 'E03')
 group by UBICACION
```

Pero esta otra consulta SQL cuenta los ejemplares de alta en todas las ubicación y finalmente oculta los que la ubicación no es E02 o E03. Por lo que este el método no es eficiente.

CÓDIGO:
```
select UBICACION , count(*) as EJEMPLARES
  from MASCOTAS
 where ESTADO = 'A'
 group by UBICACION
having UBICACION = 'E02' or UBICACION = 'E03'
```

El resultado de ambas consultas SQL es el mismo, pero hacer lo segundo es no entender el propósito de cada cláusula. Para no caer en este error basta con filtrar siempre las filas de resultado únicamente condicionando columnas de totalización en la cláusula HAVING.

*** * ***

No queda mucho más que añadir para esta cláusula, veamos otro ejemplo antes de pasar a los ejercicios:

¿Qué ubicaciones del centro de mascotas tienen tan solo un ejemplar?

La consulta que resuelve esta cuestión es casi idéntica a la primera consulta de esta lección. Ahora en lugar de ser el número de ejemplares mayor a dos, tan solo debe haber un ejemplar.

Consulta SQL:

CÓDIGO:
```
select UBICACION , count(*) as EJEMPLARES
  from MASCOTAS
 where ESTADO = 'A'
 group by UBICACION
having count(*) = 1
```

Resultado:

UBICACION	EJEMPLARES
E03	1

De hecho, como en este caso estamos forzando a que sólo haya un ejemplar, podríamos ocultar la columna de recuento omitiéndola en la cláusula SELECT del siguiente modo:

Consulta SQL:

CÓDIGO:

```
select UBICACION as UBICACIONES_CON_UN _EJEMPLAR
  from MASCOTAS
 where ESTADO = 'A'
 group by UBICACION
having count(*) = 1
```

Resultado:

UBICACIONES_CON_UN_EJEMPLAR
E03

*** * ***

Resumen

La cláusula HAVING permite establecer filtros sobre los cálculos de una consulta SQL que realizan las funciones (SUM, COUNT, etc...)

En la cláusula HAVING solo deben condicionarse columnas de cálculo, de modo que si en una consulta SQL no existe la cláusula GROUP BY, tampoco existirá cláusula HAVING.

*** * ***

Ejercicio

Usando el operador BETWEEN que vimos en las lecciones 3 y 5, construye una consulta que devuelva las ubicaciones del centro de mascotas que tienen entre 2 y 3 ejemplares.

Lección 9 - Ordenación del resultado (SQL ORDER BY)

Y llegamos a la última cláusula de una consulta SQL: ORDER BY, que permite ordenar las filas de resultado por una o más columnas. Esta cláusula no se presenta en última instancia por casualidad, sino porque siempre irá al final de una consulta y el motor SQL también será la última cosa que haga, a efectos lógicos, antes de devolver el resultado.

Una última cláusula implica una última pregunta de construcción: *¿Cómo deben ordenarse los datos resultantes?*

Supongamos que queremos obtener una lista ordenada de los empleados por sueldo, de modo que primero este situado el de menor salario y por último el de mayor:

CÓDIGO:
```
select NOMBRE,APELLIDOS,SALARIO
  from EMPLEADOS
 order by SALARIO
```

NOMBRE	APELLIDOS	SALARIO
Elena	Rubio Cuestas	1300
Margarita	Rodríguez Garcés	1325.5
José	Calvo Sisman	1400
Carlos	Jiménez Clarín	1500

Observamos como introduciendo la cláusula ORDER BY e indicando la columna por la que ordenar, el resultado viene ordenado de forma ascendente (ASC), es decir, de menor a mayor. ¿Y si queremos ordenar a la inversa, de mayor a menor? Bien, en ese caso se debe indicar que la ordenación es descendente (DESC). Veamos esto tomando como patrón la consulta anterior:

CÓDIGO:
```
select NOMBRE,APELLIDOS,SALARIO
  from EMPLEADOS
 order by SALARIO desc
```

NOMBRE	APELLIDOS	SALARIO
Carlos	Jiménez Clarín	1500
José	Calvo Sisman	1400
Margarita	Rodríguez Garcés	1325.5
Elena	Rubio Cuestas	1300

Por tanto si seguido del campo por el que queremos ordenar indicamos ASC, o bien no indicamos nada, la ordenación se hará de forma ascendente, mientras que si indicamos DESC, se hará de forma descendente.

* * *

Veamos un ejemplo donde se ordena por más de un campo. Tomemos por ejemplo la tabla MASCOTAS, y obtengamos una lista de los perros que han pasado por el centro, de modo que primero aparezcan las bajas, y al final las altas, o perros que siguen en el centro. Además queremos que en segundo término la lista esté ordenada por nombre:

CÓDIGO:
```
select *
  from MASCOTAS
 where ESPECIE = 'P'
 order by ESTADO desc, NOMBRE asc
```

ID_MASCOTA	NOMBRE	ESPECIE	SEXO	UBICACION	ESTADO
1	Budy	P	M	E05	B
2	Pipo	P	M	E02	B
4	Bruts	P	M	E03	A
16	Canela	P	H	E02	A
3	Nuna	P	H	E02	A
6	Sombra	P	H	E05	A
9	Trabis	P	M	E02	A
12	Truca	P	H	E02	A
13	Zulay	P	H	E05	A

* * *

Veamos un poco como resuelve esto el motor SQL. No vamos a entrar en los algoritmos de ordenación que usa, entre otras cosas porque tampoco los conozco, pero si quiero trazar, como hemos venido haciendo a lo largo del curso, lo que el motor SQL hace. En la consulta anterior el motor SQL recorre la tabla MASCOTAS y selecciona aquellos registros en que el campo ESPECIE contiene una "P", ignorando el resto. De los registros que satisfacen la cláusula WHERE tomará todos los campos, puesto que se ha indicado un asterisco en la cláusula SELECT, y una vez ha recorrido toda la tabla y tiene el resultado, lo ordenará según se indica en la cláusula ORDER BY.

Lo que debe quedar claro es que la ordenación, a efectos lógicos, se realiza siempre al final de todo, sobre las filas de resultado, al margen de la tabla, y siempre lo hará así por muy extensa y compleja que sea una consulta. La ordenación es lo último de lo último que realiza el motor SQL. Y como la ordenación se realiza sobre las filas de resultado, existen otras formas de indicar que columnas van a establecer la ordenación. Podemos por ejemplo hacer referencia a la columna por el orden que ocupa en la cláusula SELECT, por ejemplo:

En esta consulta estamos indicando que ordene por el tercer campo de la clausula SELECT:

CÓDIGO:
```
select ID_EMPLEADO , NOMBRE , APELLIDOS
  from EMPLEADOS
 order by 3
```

ID_EMPLEADO	NOMBRE	APELLIDOS
3	José	Calvo Sisman
1	Carlos	Jiménez Clarín
4	Margarita	Rodríguez Garcés
2	Elena	Rubio Cuestas

Para ordenar se puede indicar indistintamente el alias con el que se ha rebautizado la columna, o el campo de tabla tenga o no tenga alias.

CÓDIGO:
```
select NOMBRE as NAME, APELLIDOS as SURNAMES, SALARIO as SALARY
  from EMPLEADOS
 order by SURNAMES
```

Resultado:

NAME	SURNAMES	SALARY
José	Calvo Sisman	1400
Carlos	Jiménez Clarín	1500
Margarita	Rodríguez Garcés	1325.5
Elena	Rubio Cuestas	1300

De todos modos se recomienda usar para indicar la columna por la que se quiere ordenar, los nombres de tabla, por dos razones.

1. Si usamos alias y este cambia, se debe modificar la cláusula ORDER BY
2. Si usamos posición de columna y se añaden o se eliminan campos de la cláusula SELECT, es posible que se deba modificar la cláusula ORDER BY

*** * ***

Resumen

La cláusula ORDER BY permite establecer el orden de las filas de resultado en función de las columnas que se indiquen en dicha cláusula:

CÓDIGO:

```
order by CAMPO_1 , CAMPO_2 , ... , CAMPO_N
```

Para ordenar en forma descendente por una columna debemos indicar a continuación del nombre de la columna la palabra clave DESC. Para hacerlo de forma ascendente no hace falta indicar nada, si se quiere enfatizar se usa la palabra clave ASC.

CÓDIGO:

```
order by CAMPO_1 desc , CAMPO_2 desc , ... , CAMPO_N asc
```

Para hacer referencia a una columna en la clausula ORDER BY, es indiferente usar el alias de una columna, que el orden de la columna en la clausula SELECT, que el nombre de campo de la tabla. Sin embargo se recomienda hacer esto último para minimizar fuentes de error.

*** * ***

Ejercicio 1

Obtenga una lista de las personas de la tabla PERSONAS, donde primero aparezcan las rubias, después las altas, y finalmente las que llevan gafas. De manera que la primera persona de la lista, si la hay, será rubia alta y sin gafas, y la última, si la hay, no será rubia ni alta y llevará gafas.

Ejercicio 2

Obtenga el número actual de ejemplares de cada ubicación del centro de mascotas, que tengan dos o más ejemplares ordenado de mayor a menor por número de ejemplares y en segundo término por ubicación.

Lección 10 - El operador LIKE / El valor NULL

Antes de finalizar la primera parte del curso deben tratarse dos aspectos relevantes del lenguaje que se han quedado en el tintero: el operador LIKE, y el valor NULL.

El operador LIKE

Este operador se aplica a datos de tipo cadena y se usa para buscar registros, es capaz de hallar coincidencias dentro de una cadena bajo un patrón dado, por ejemplo:

¿Qué empleados su primer apellido comienza por "R"?
Veamos primero la consulta SQL que responde a esto:

CÓDIGO:
```
select *
  from EMPLEADOS
 where APELLIDOS like 'R%'
```

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
2	Elena	Rubio Cuestas	1978-09-25	M	Secretaria	1300
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

El interés de la anterior consulta se centra en la expresión: APELLIDOS like 'R%'

donde "like" es el operador, APELLIDOS es el operando variable que toma valores para cada registro de la tabla EMPLEADOS, y el operando constante: "R%", es un patrón de búsqueda donde el "%" representa un comodín que junto con el operador LIKE tiene el cometido de reemplazar a cualquier cadena de texto, incluso la cadena vacía, para evaluar la expresión booleana. De modo que cualquier valor que haya en el campo APELLIDOS que empiece por una "R" seguida de cualquier cosa(%) dará cierto para la expresión: APELLIDOS like 'R%'.

Veamos otro ejemplo: *¿Qué empleados su segundo apellido termina en "N"?*

En este caso interesa que el campo APELLIDOS empiece por cualquier cosa y acabe con una "N", por lo tanto la expresión que nos filtrará adecuadamente esto es: APELLIDOS like '%N'

CÓDIGO:
```
select *
  from EMPLEADOS
 where APELLIDOS like '%N'
```

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
3	José	Calvo Sisman	1990-11-12	H	Mozo	1400

> “
>
> En MySQL la comparación de cadenas por defecto no es sensible a mayúsculas, de ahí que aun indicando una "N" mayúscula encuentre los apellidos acabados en "n" minúscula.

Obsérvese como en este caso el "%" debe aparecer antes que la "N" en el patrón de búsqueda, puesto que queremos que los apellidos acaben en "N" y no que comiencen por "N".

Veamos una última aplicación de este recurso. ¿Qué devuelve esta consulta?:

CÓDIGO:
```
select *
  from EMPLEADOS
 where APELLIDOS like '%AR%'
```

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
1	Carlos	Jiménez Clarín	1985-05-03	H	Mozo	1500
4	Margarita	Rodríguez Garcés	1992-05-16	M	Secretaria	1325.5

Pues está devolviendo aquellos registros que el campo APELLIDOS contiene la cadena: "AR", ya sea al principio, al final, o en cualquier posición intermedia. De ahí que en el patrón de búsqueda encontremos la cadena "AR" acompañada de comodines a ambos lados.

Este recurso resulta muy útil para buscar coincidencias en campos sin necesidad de buscar el valor exacto. Si se nos pide buscar al empleado José Calvo, podemos limitarnos a buscar cualquier valor que contenga la cadena "Calvo" en el campo APELLIDOS para localizar el registro.

*** * ***

El valor NULL

Cuando se diseña una tabla en la base de datos, una de las propiedades que se establece para los campos de la tabla es si pueden contener o no un valor nulo. Por ejemplo, supongamos que tenemos una flota de vehículos. En la tabla VEHICULOS se guardan los datos de cada unidad, datos como el modelo, que obviamente no puede ser nulo puesto

que todo vehículo pertenece a un modelo, pero también por ejemplo la fecha de la última revisión obligatoria, cuyo valor sí puede ser nulo, especialmente si el vehículo es nuevo y todavía nunca se ha sometido a dicha revisión. Por tanto ya se ve que hay campos que no pueden ser nulos y otros sí, dependiendo de qué información se guarda.

Para ilustrar las particularidades del valor NULL tomemos la tabla VEHICULOS:

ID_VEHICULO	MARCA	MODELO	PROX_ITV	ULTI_ITV
1	Alfa Romeo	Brera	2011-10-20	
3	BMW	X3	2010-07-18	
5	Ford	Fiesta	2011-04-22	
2	Seat	Panda	2009-12-01	2008-12-01
4	Citroën	C2	2010-08-24	2009-08-24

En los Datos se observa como tres de las cinco unidades nunca han pasado la revisión obligatoria, puesto que el valor para el campo ULTI_ITV (última inspección técnica del vehículo) es nulo.

El operador IS NULL

Este operador permite establecer en la cláusula WHERE de una consulta SQL condiciones para filtrar por campos de valor nulo, por ejemplo: ¿Qué vehículos nunca han pasado la ITV?

CÓDIGO:
```
select *
  from VEHICULOS
 where ULTI_ITV is null
```

ID_VEHICULO	MARCA	MODELO	PROX_ITV	ULTI_ITV
1	Alfa Romeo	Brera	2011-10-20	
3	BMW	X3	2010-07-18	
5	Ford	Fiesta	2011-04-22	

Los vehículos que han pasado como mínimo una vez la ITV serán aquellos que el campo ULTI_ITV no contenga un valor nulo, para conocer estos datos debemos establecer la siguiente condición:

CÓDIGO:
```
select *
  from VEHICULOS
 where ULTI_ITV is not null
```

ID_VEHICULO	MARCA	MODELO	PROX_ITV	ULTI_ITV
2	Seat	Panda	2009-12-01	2008-12-01
4	Citroën	C2	2010-08-24	2009-08-24

Por tanto ya se ve que el valor nulo es un poco especial, en realidad es un valor indeterminado, una muestra de ello es la excepción que se da a la afirmación que se hizo en la lección 5 sobre operadores: "si negamos la cláusula WHERE de una consulta SQL con el operador NOT, se obtienen los registros que antes se ignoraban y se ignoran los que antes se seleccionaban".

Veamos una muestra de ello. La siguiente consulta SQL devuelve los vehículos que pasaron la ITV durante el 2008:

CÓDIGO:
```
select *
  from VEHICULOS
 where ULTI_ITV between '20080101' and '20081231'
```

ID_VEHICULO	MARCA	MODELO	PROX_ITV	ULTI_ITV
2	Seat	Panda	2009-12-01	2008-12-01

Es de esperar entonces que al negar la cláusula WHERE obtengamos todos los registros menos el Seat Panda:

CÓDIGO:
```
select *
  from VEHICULOS
 where not (ULTI_ITV between '20080101' and '20081231')
```

ID_VEHICULO	MARCA	MODELO	PROX_ITV	ULTI_ITV
4	Citroën	C2	2010-08-24	2009-08-24

Sin embargo no ocurre así; la consulta ha devuelto los vehículos que NO pasaron la revisión durante el 2008, pero los registros con valor nulo en el campo ULTI_ITV han vuelto a ser ignorados. Esto nos obliga a extremar el cuidado con estos campos sabiendo que: cuando el motor SQL evalúa un dato nulo en una expresión de la cláusula WHERE, no sabe resolver la operación y considera que el resultado de dicha expresión es falso. Pero en el caso de usar IS NULL, o bien IS NOT NULL, el motor SQL sí la sabe resolver. De modo que si anteriormente se quería obtener todos los vehículos que NO pasaron la ITV durante el 2008, debe plantearse si se incluyen los vehículos que NO la han pasado nunca, y si se decide que sí, debe especificarse en la cláusula WHERE:

CÓDIGO:
```
select *
  from VEHICULOS
 where not (ULTI_ITV between '20080101' and '20081231')
    or ULTI_ITV is null
```

ID_VEHICULO	MARCA	MODELO	PROX_ITV	ULTI_ITV
3	BMW	X3	2010-07-18	
4	Citroën	C2	2010-08-24	2009-08-24
5	Ford	Fiesta	2011-04-22	
1	Alfa Romeo	Brera	2011-10-20	

Para saber que campos de una tabla pueden tomar un valor nulo, se puede pedir al SGDB una descripción de la tabla:

CÓDIGO:
```
desc VEHICULOS
```

FIELD	TYPE	NULL	KEY	DEFAULT	EXTRA
ID_VEHICULO	int(11)	NO	PRI		auto_increment
MARCA	varchar(30)	NO			
MODELO	varchar(30)	NO			
PROX_ITV	date	NO			
ULTI_ITV	date	YES			

En la columna Null se informa para cada campo si permite valores nulos.

*** * ***

Para finalizar la lección retomemos algo que quedó pendiente, me refiero de la función de recuento COUNT aplicada a un campo concreto. Hasta ahora solo habíamos usado COUNT(*), fíjese en la consulta siguiente:

CÓDIGO:
```
select count(*) , count(ID_VEHICULO) , count(ULTI_ITV)
  from VEHICULOS
```

count(*)	count(ID_VEHICULO)	count(ULTI_ITV)
5	5	2

¿Que está devolviendo? Bueno en la primera columna lo que ya se trató en la lección 5, el recuento de registros de toda la tabla puesto que se ha omitido la cláusula WHERE. En la segunda columna, donde se hace un recuento del campo ID_VEHICULO parece que lo mismo, el número de registros de toda la tabla. Pero en la tercera columna, donde se hace

el recuento del campo ULTI_ITV, el valor del recuento es dos. En realidad está contando registros cuyo valor en el campo ULTI_ITV no es nulo, dicho de otro modo, la función de recuento COUNT aplicada a un campo, ignora los registros donde el valor de ese campo es nulo.

Esto es extensible a las otras funciones de totalización: SUM, AVG, MAX y MIN, los valores nulos no se pueden comparar ni sumar, no pueden intervenir en un promedio, no son valores máximos ni mínimos, son simplemente valores nulos.

*** * ***

Resumen

El operador LIKE permite, junto a un patrón de búsqueda, hallar coincidencias dentro de una cadena. En general:

- CADENA like 'hola%' -> devuelve cierto si el valor del campo CADENA empieza por "hola"
- CADENA like '%hola' -> devuelve cierto si el valor del campo CADENA termina por "hola"
- CADENA like '%hola%' -> devuelve cierto si el valor del campo CADENA contiene la cadena "hola"

Dependiendo del diseño de una tabla, algunos campos pueden tomar valores nulos. Cuando estos campos se condicionan en la cláusula WHERE y el motor SQL evalúa la expresión para un valor nulo, el resultado será siempre falso salvo que estemos usando el operado IS NULL o bien IS NOT NULL, en cuyo caso dependerá del caso concreto que se esté evaluando.

Las funciones de totalización ignoran el valor nulo para desarrollar los cálculos.

*** * ***

Ejercicio 1

¿Qué empleados se apellidan Calvo?

Ejercicio 2

Considerando que en la tabla VEHICULOS el campo PROX_ITV guarda la fecha de la próxima ITV que ha de pasar cada vehículo:

¿Qué vehículos que nunca han pasado la ITV deben pasar la primera revisión durante el año 2011?

Lección 11 - Síntesis de la primera parte

Durante las diez lecciones anteriores se ha trabajado con diversas consultas SQL que accedían a una sola tabla de la BD. El propósito de trabajar con una sola tabla es el de mostrar sobre el caso más simple (sólo interviene una tabla), la funcionalidad de todas las cláusulas SQL que una consulta puede incorporar. Con ello se consigue que en todos los ejemplos el motor SQL deba recorrer la tabla una sola vez, siendo más fácil entender que papel juega cada cláusula SQL y como deben combinarse para explotar los datos de una tabla. Aprovechado los ejemplos se han ido mostrando sobre la marcha distintos recursos y funcionalidades que el lenguaje SQL proporciona. Se ha pretendido mostrar estos recursos bajo un contexto, para acompañarlos de una aplicación práctica que ayude a entender su cometido.

*** * ***

A lo largo de estas lecciones se han estudiado dos tipos de consultas SQL que se diferencian claramente por el resultado que se obtiene:

Consultas SQL llanas:

Son las consultas que seleccionan los campos de una tabla indicados en la cláusula SELECT, de los registros filtrados por la cláusula WHERE.

Consultas SQL de cálculo:

Son las consultas que calculan los totales de una tabla indicados en la cláusula SELECT, mediante funciones de totalización (SUM, MAX,...), sobre los registros filtrados por la cláusula WHERE. Estos totales se pueden agrupar por los campos indicados en la cláusula GROUP BY, en cuyo caso, pueden ser filtrados por las condiciones establecidas en la clausula HAVING.

*** * ***

Cada cláusula SQL está relacionada o responde a una pregunta de construcción:

1. **SELECT**: ¿Qué datos nos piden?
2. **FROM**: ¿Dónde están los datos?
3. **WHERE**: ¿Qué requisitos deben cumplir los registros?
4. **GROUP BY**: ¿Cómo deben agruparse los datos?
5. **HAVING**: ¿Qué requisitos deben cumplir los cálculos de totalización?
6. **ORDER BY**: ¿Por qué columnas debe ordenarse el resultado?

Para construir una consulta como mínimo deben intervenir las cláusulas SELECT y FROM, el resto dependerá de lo que se quiera obtener, pero en cualquier caso el orden en que cada cláusula aparece en una consulta SQL no puede cambiarse, la cláusula SELECT siempre será la primera, y ORDER BY, si interviene, siempre será la última.

En general una consulta SQL tiene esta forma:

CÓDIGO:

```
select CAMPO_S1, ... , CAMPO_SN
  from TABLA_1, ... , TABLA_N
 where CONDICIONES_WHERE
 group by CAMPO_G1, ... , CAMPO_GN
having CONDICIONES_HAVING
 order by CAMPO_O1, ... , CAMPO_ON
```

En la cláusula **SELECT** se establecen las lista de campos que mostrará el resultado de la consulta, poniendo DISTINCT precediendo a la lista de campos se eliminan posibles registros duplicados que el resultado pueda devolver. Así mismo podemos rebautizar las columnas de resultado asignando un alias a cada campo.

En la cláusula **FROM** se establece la fuente de los datos, las tablas que intervienen en la consulta.

En la cláusula **GROUP BY** se establece los grupos de datos por los que se quiere obtener totales, no tendría sentido usar esta cláusula si no se indican cálculos de totalización (SUM , MAX, AVG , ...) en la cláusula SELECT.

La condiciones establecidas en la cláusula **WHERE** tienen como propósito filtrar registros de la tabla, mientras que las condiciones de la cláusula **HAVING** filtran filas de resultados condicionando únicamente los datos calculados por las funciones de totalización (SUM , MAX, AVG , ...). A efectos prácticos para este curso se considera que la cláusula HAVING sólo puede intervenir en una consulta SQL si lo hace la cláusula GROUP BY, de lo contrario no tiene sentido.

La cláusula **ORDER BY** permite finalmente ordenar el resultado por diferentes columnas, y en sentido ascendente o descendente, antes de ser presentado.

*** * ***

Si usted ha llegado hasta aquí le animo a que continué con el curso. En las siguientes lecciones se va a abordar lo que quizá es el mayor potencial de las BD relacionales: la reunión de datos. Usted aprenderá a trabajar con más de una tabla en una misma consulta SQL.

*** * ***

Ejercicio

Supongamos que usted tiene un amigo que es jugador de póquer, el pobre no sabe si sus ganancias en el juego son positivas o negativas porque no lleva un control sobre ello, por lo que usted se ofrece a gestionarle las ganancias. Le dice a su amigo que cuando acabe una sesión de juego le comunique a usted el dinero que ha ganado o perdido, entendiendo perdida como una ganancia o número en negativo.

Diseñe una tabla, es decir, los campos y tipo de dato de cada campo, para poder registrar la información que su amigo le facilita, y mediante SQL pueda responder en cualquier momento a las siguientes preguntas:

¿Cuáles son las ganancias actuales?
¿Cuánto dinero se ganó durante el mes de marzo de 2009?

Una vez diseñada la tabla construya las consultas SQL que responden a cada una de estas preguntas.

Lección 12 - El producto cartesiano (SQL FROM)

Si usted durante su infancia jugó a vestir muñecas, recordará que el juego consistía en un conjunto de camisetas y otro de pantalones de manera que, combinando un elemento de cada conjunto, podían confeccionarse distintas mudas con las que vestir la muñeca.

Es más que probable conociendo a los niños que, si es el caso, usted realizara el producto cartesiano de estos dos conjuntos, es decir, el producto cartesiano de las camisetas con los pantalones. Para ello debió vestir la muñeca, al menos una vez, con todas y cada una de las combinaciones posibles, de modo que todas las camisetas se combinaron con todos los pantalones, y por ende, todos los pantalones con las camisetas. El conjunto resultante de pares de elementos posibles es el producto cartesiano de ambos conjuntos.
Si suponemos que el número de camisetas era de cinco elementos y el de pantalones de cuatro, entonces las combinaciones posibles son veinte (5 x 4), ya que para cada camiseta tenemos cuatro mudas distintas, una con cada pantalón. Esos veinte pares de elementos son los veinte elementos del conjunto resultante. Por tanto el producto cartesiano entre dos conjuntos da como resultado un nuevo conjunto con tantos elementos como pares posibles de elementos puedan combinarse.

*** * ***

Si trasladamos esto al ámbito que nos ocupa de las bases de datos, una tabla es en realidad un conjunto de registros, y al realizar una consulta como la siguiente:

CÓDIGO:
```
select *
  from TABLA1 , TABLA2
```

el motor SQL realiza el producto cartesiano de ambos conjuntos, combinando todos los elementos o registros de la TABLA1 con todos los registros de la TABLA2, de manera que cada fila de resultado es una de las combinaciones posibles. Por tanto el número de filas resultantes será igual al número de registros de la TABLA1 multiplicado por el número de registros de la TABLA2.

Para ilustrar esto vamos a considerar estas dos tablas:

Tabla CAMISAS:

ID_CAMISA	CAMISA	PESO_GR
1	lino blanca	210
2	algodon naranja	290
3	seda negra	260

Tabla PANTALONES:

ID_PANTALON	PANTALON	PESO_GR
1	tela azul marino	470
2	pana marron claro	730

En estas tablas se guardan el vestuario de camisas y pantalones por separado, cada prenda tiene un número que la identifica y un peso expresado en gramos.
Si ahora nos preguntamos lo siguiente: ¿Qué combinaciones ofrece este vestuario? La respuesta es el producto cartesiano de ambas tablas:

CÓDIGO:
```
select *
  from CAMISAS , PANTALONES
```

ID_CAMISA	CAMISA	PESO_GR	ID_PANTALON	PANTALON	PESO_GR
1	lino blanca	210	1	tela azul marino	470
1	lino blanca	210	2	pana marron claro	730
2	algodon naranja	290	1	tela azul marino	470
2	algodon naranja	290	2	pana marron claro	730
3	seda negra	260	1	tela azul marino	470
3	seda negra	260	2	pana marron claro	730

“

Al intervenir dos tablas en una consulta SQL, en la cláusula SELECT se pueden solicitar datos de cualquiera de las dos tablas. En el ejemplo anterior se indica un asterisco, de modo que el motor SQL devuelve todos los campos de la primera tabla, seguido de todos los campos de la segunda tabla.

Veamos un poco como ejecuta esta consulta el motor SQL: primero escoge la tabla CAMISAS y toma el primer registro, con él recorre toda la tabla PANTALONES asociándole todos los registros de la tabla PANTALONES. Acto seguido tomará el segundo registro de la tabla CAMISAS y repetirá la operación asociando a este todos los registros de la tabla PANTALONES. Y así irá repitiendo esta operación con todos los registros de la tabla CAMISAS hasta llegar al último. Por tanto recorrerá la tabla PANTALONES tantas veces como registros hay en la tabla CAMISAS.

Para eliminar las columnas que no interesan construimos la siguiente consulta SQL:

CÓDIGO:
```
select CAMISA , PANTALON
  from CAMISAS , PANTALONES
```

CAMISA	PANTALON
lino blanca	tela azul marino
lino blanca	pana marron claro
algodon naranja	tela azul marino
algodon naranja	pana marron claro
seda negra	tela azul marino
seda negra	pana marron claro

Campo ambiguo

La ambigüedad se da cuando en una consulta SQL de por ejemplo dos tablas, en ambas existen uno o más campos con el mismo nombre, y estos campos aparecen en cualquier cláusula de la consulta. Ningún SGBD es hasta ahora adivino, por lo que si no le indicamos a cuál de las tablas pertenece el campo al que hacemos mención, devolverá un error.

Para ilustrar lo que es un campo ambiguo en una consulta SQL, planteamos la siguiente cuestión, ¿qué mudas pueden confeccionarse con este vestuario y que pesa en conjunto cada muda, es decir, pantalón más camisa?

La solución pasa por sumar el peso de la camisa más el del pantalón, ese dato se guarda en un campo que se denomina igual en ambas tablas: PESO_GR, por lo que debe indicarse a que tabla pertenece cada campo PESO_GR que aparezca en la consulta. Esto se consigue precediendo al campo por el nombre de la tabla, separando la tabla del campo por un punto:

CÓDIGO:
```
select CAMISA , PANTALON ,
       CAMISAS.PESO_GR + PANTALONES.PESO_GR as PESO_MUDA
  from CAMISAS , PANTALONES
```

CAMISA	PANTALON	PESO_MUDA
lino blanca	tela azul marino	680
lino blanca	pana marron claro	940
algodon naranja	tela azul marino	760
algodon naranja	pana marron claro	1020
seda negra	tela azul marino	730
seda negra	pana marron claro	990

> “
>
> **Sugerencia**: Lleve la anterior consulta al banco de pruebas, accesible desde la web deletesql.com, y elimine la tabla y el punto que preceden a cualquiera de los dos campos PESO_GR que intervienen.

En realidad esto se debería abordar al revés. En general en una consulta SQL con más de una tabla debe indicarse siempre a que tabla pertenece cada campo, pudiendo no hacerse en el caso de que no exista ambigüedad, aunque no se recomienda. Las tablas pueden cambiar en un futuro, y donde no existe ambigüedad hoy, puede no ser así mañana. Por tanto es más prudente para evitar fuentes de errores futuras construir la anterior consulta así:

CÓDIGO:
```
Select CAMISAS.CAMISA, PANTALONES.PANTALON,
       CAMISAS.PESO_GR + PANTALONES.PESO_GR as PESO_MUDA
  from CAMISAS , PANTALONES
```

Alias de tabla

Al igual que el SQL permite rebautizar columnas de la cláusula SELECT, también permite rebautizar tablas de la cláusula FROM. Para ello se emplea de igual modo la palabra clave AS. Se consigue así que las consultas sean menos laboriosas de construir, menos tupidas y más simplificadas a la vista del desarrollador. Por ejemplo:

CÓDIGO:
```
select C.CAMISA , P.PANTALON ,
       C.PESO_GR + P.PESO_GR as PESO_MUDA
  from CAMISAS as C, PANTALONES as P
```

CAMISA	PANTALON	PESO_MUDA
lino blanca	tela azul marino	680
lino blanca	pana marron claro	940
algodon naranja	tela azul marino	760
algodon naranja	pana marron claro	1020
seda negra	tela azul marino	730
seda negra	pana marron claro	990

> “
>
> Al igual que con los alias de campo, no es necesario indicar la palabra clave AS para establecer un alias, si la omitimos el SGBD entiende de igual modo que la palabra que sigue a la tabla o al campo es un alias. Lleve la anterior consulta al banco de pruebas, accesible desde la web deletesql.com, y elimine las palabras clave AS tanto de la cláusula SELECT como de la cláusula FROM, dejando un espacio entre la tabla y su alias, observará que el resultado es el mismo.

*** * ***

¿Qué ocurre si cruzamos tres tablas en una misma consulta SQL?

Si intuitivamente usted cree que el motor SQL realizará el producto cartesiano de tres tablas esta en lo cierto. Tomemos la tabla CALZADOS:

ID_CALZADO	CALZADO	PESO_GR
1	deportivas	675
2	mocasines	800
3	botas	1050

Es de esperar que si a la consulta SQL de las camisas con los pantalones le añadimos esta tabla, el motor SQL realice el producto cartesiano sobre estos dos conjuntos, es decir, sobre el conjunto de las camisas-pantalones(6 elementos) y el conjunto de los calzados:

CÓDIGO:
```
select *
  from CAMISAS , PANTALONES , CALZADOS
```

ID_CAMISA	CAMISA	PESO_GR	ID_PANTALON	PANTALON	PESO_GR	ID_CALZADO	CALZADO	PESO_GR
1	lino blanca	210	1	tela azul marino	470	1	deportivas	675
1	lino blanca	210	1	tela azul marino	470	2	mocasines	800

ID_CAMISA	CAMISA	PESO_GR	ID_PANTALON	PANTALON	PESO_GR	ID_CALZADO	CALZADO	PESO_GR
1	lino blanca	210	1	tela azul marino	470	3	botas	1050
1	lino blanca	210	2	pana marron claro	730	1	deportivas	675
1	lino blanca	210	2	pana marron claro	730	2	mocasines	800
1	lino blanca	210	2	pana marron claro	730	3	botas	1050
2	algodon naranja	290	1	tela azul marino	470	1	deportivas	675
2	algodon naranja	290	1	tela azul marino	470	2	mocasines	800
2	algodon naranja	290	1	tela azul marino	470	3	botas	1050
2	algodon naranja	290	2	pana marron claro	730	1	deportivas	675
2	algodon naranja	290	2	pana marron claro	730	2	mocasines	800
2	algodon naranja	290	2	pana marron claro	730	3	botas	1050
3	seda negra	260	1	tela azul marino	470	1	deportivas	675
3	seda negra	260	1	tela azul marino	470	2	mocasines	800
3	seda negra	260	1	tela azul marino	470	3	botas	1050
3	seda negra	260	2	pana marron claro	730	1	deportivas	675
3	seda negra	260	2	pana marron claro	730	2	mocasines	800
3	seda negra	260	2	pana marron claro	730	3	botas	1050

Las combinaciones o registros resultantes es igual a la multiplicación del número de camisas por el número de pantalones por el número de calzados: (3 x 2) x 3 = 6 x 3 = 18

Realicemos la consulta que devolvía las distintas mudas con su peso, considerando también la tabla CALZADOS:

CÓDIGO:

```
select C.CAMISA , P.PANTALON , Z.CALZADO ,
       C.PESO_GR + P.PESO_GR + Z.PESO_GR as PESO_MUDA
  from CAMISAS C , PANTALONES P , CALZADOS Z
 order by C.ID_CAMISA , P.ID_PANTALON , Z.ID_CALZADO
```

CAMISA	PANTALON	CALZADO	PESO_MUDA
lino blanca	tela azul marino	deportivas	1355
lino blanca	tela azul marino	mocasines	1480
lino blanca	tela azul marino	botas	1730
lino blanca	pana marron claro	deportivas	1615
lino blanca	pana marron claro	mocasines	1740
lino blanca	pana marron claro	botas	1990
algodon naranja	tela azul marino	deportivas	1435
algodon naranja	tela azul marino	mocasines	1560
algodon naranja	tela azul marino	botas	1810
algodon naranja	pana marron claro	deportivas	1695
algodon naranja	pana marron claro	mocasines	1820
algodon naranja	pana marron claro	botas	2070
seda negra	tela azul marino	deportivas	1405
seda negra	tela azul marino	mocasines	1530
seda negra	tela azul marino	botas	1780
seda negra	pana marron claro	deportivas	1665
seda negra	pana marron claro	mocasines	1790
seda negra	pana marron claro	botas	2040

*** * ***

Para acabar esta lección y a modo de adelanto, fíjese en las filas resultantes de la anterior consulta SQL, usted que es el informático no tendrá problemas en saber que el resultado son en realidad las distintas mudas que se pueden confeccionar con el vestuario disponible y el peso de cada una. Pero créame, su jefe se lo agradecerá si le muestra el resultado así:

CÓDIGO:

```
select concat('Camisa de ' , C.CAMISA ,
              ' con pantalón de ' , P.PANTALON ,
              ' y ' , Z.CALZADO) as MUDA ,
             C.PESO_GR + P.PESO_GR + Z.PESO_GR PESO_MUDA
  from CAMISAS C , PANTALONES P , CALZADOS Z
 order by C.ID_CAMISA , P.ID_PANTALON , Z.ID_CALZADO
```

MUDA	PESO_MUDA
Camisa de lino blanca con pantalón de tela azul marino y deportivas	1355
Camisa de lino blanca con pantalón de tela azul marino y mocasines	1480
Camisa de lino blanca con pantalón de tela azul marino y botas	1730
Camisa de lino blanca con pantalón de pana marron claro y deportivas	1615
Camisa de lino blanca con pantalón de pana marron claro y mocasines	1740
Camisa de lino blanca con pantalón de pana marron claro y botas	1990
Camisa de algodon naranja con pantalón de tela azul marino y deportivas	1435
Camisa de algodon naranja con pantalón de tela azul marino y mocasines	1560
Camisa de algodon naranja con pantalón de tela azul marino y botas	1810
Camisa de algodon naranja con pantalón de pana marron claro y deportivas	1695
Camisa de algodon naranja con pantalón de pana marron claro y mocasines	1820
Camisa de algodon naranja con pantalón de pana marron claro y botas	2070
Camisa de seda negra con pantalón de tela azul marino y deportivas	1405
Camisa de seda negra con pantalón de tela azul marino y mocasines	1530
Camisa de seda negra con pantalón de tela azul marino y botas	1780
Camisa de seda negra con pantalón de pana marron claro y deportivas	1665
Camisa de seda negra con pantalón de pana marron claro y mocasines	1790
Camisa de seda negra con pantalón de pana marron claro y botas	2040

CONCAT es una función que concatena datos de tipo cadena dando como resultado una única cadena. No debe confundirse esta función con las funciones de totalización (SUM , AVG ....). Las funciones se tratarán más adelante, por el momento no se preocupe por ello y quédese con la idea. Añadir también que la función CONCAT es un recurso particular del SGBD MySQL fuera del estándar SQL, por ejemplo en Oracle el modo de concatenar cadenas aplicado a este ejemplo es el siguiente:

CÓDIGO:

```
select 'Camisa de ' || C.CAMISA || ' con pantalón de '
       || P.PANTALON as MUDA
  from CAMISAS C, PANTALONES P
```

* * *

Resumen

En las consultas SQL de dos o más tablas:

- Sin cláusula WHERE el motor SQL realiza el producto cartesiano de todas las tablas.

- Para evitar la ambigüedad de campos y por tanto asegurar el funcionamiento de la consulta SQL, debe indicarse para cualquier campo que aparezca en la consulta y en cualquiera de sus cláusulas, la tabla a la que pertenece del siguiente modo: TABLA.CAMPO

- Es posible establecer alias o sobrenombres de tabla, con intención de agilizar la construcción de la consulta, y usar ese alias para indicar a que tabla pertenece cada campo.

* * *

Ejercicio 1

Realice una consulta que devuelva las combinaciones posibles entre los pantalones y los calzados, sin más columnas que la descripción de cada prenda. Use alias de tabla para indicar a que tabla pertenece cada campo de la cláusula SELECT.

Ejercicio 2

Si en una BD existe una tabla T1 con 4 campos y 12 registros, y una tabla T2 con 7 campos y 10 registros, ¿cuántas filas y columnas devolvería la siguiente consulta?

CÓDIGO:

```
select *
  from T1 , T2
```

Lección 13 - Consultas III (SQL SELECT FROM WHERE)

Ahora que usted ya conoce el producto cartesiano entre dos o más tablas, y conoce también como trabajar con consultas SQL de una sola tabla, en esta lección descubrirá que, a nivel lógico, hay muy poca diferencia entre las consultas de una sola tabla y las de dos o más tablas.
Una tabla está formada por un conjunto de registros con un cierto número de campos. Podemos afirmar también que el producto cartesiano entre dos tablas está formado por un conjunto de filas de datos con un cierto número de columnas. Si se abstraen estos dos conceptos, ¿no se está hablando de lo mismo?

Véase la tabla MUDAS:

ID_CAMISA	CAMISA	PESO_CAMISA	ID_PANTALON	PANTALON	PESO_PANTALON
1	lino blanca	210	1	tela azul marino	470
1	lino blanca	210	2	pana marron claro	730
2	algodon naranja	290	1	tela azul marino	470
2	algodon naranja	290	2	pana marron claro	730
3	seda negra	260	1	tela azul marino	470
3	seda negra	260	2	pana marron claro	730

La siguiente consulta SQL:

CÓDIGO:
```
select *
  from MUDAS
 where ID_CAMISA = 1
```

mostraría únicamente las mudas con la camisa de lino blanca, es decir, la camisa con identificador igual a 1, si no fuese porque la tabla MUDAS no existe en la BD. Lo que aparenta ser una tabla es en realidad el resultado de la siguiente consulta SQL:

CÓDIGO:
```
select C.ID_CAMISA , C.CAMISA , C.PESO_GR as PESO_CAMISA ,
       P.ID_PANTALON , P.PANTALON , P.PESO_GR as PESO_PANTALON
  from CAMISAS C , PANTALONES P
```

Por tanto diremos que cuando se combinan varias tablas en una consulta SQL, **a efectos lógicos**, **el producto cartesiano** de dichas tablas **se puede considerar como una nueva tabla** con tantos registros y campos como filas y columnas resuelva la operación, **siendo válido** sobre estos registros y campos **todo lo expuesto en la primera parte del curso**, donde se trabajó únicamente con una sola tabla.

La consulta SQL que devuelve las mudas de la camisa con identificador 1 quedaría de la siguiente manera:

CÓDIGO:

```
select C.ID_CAMISA , C.CAMISA , C.PESO_GR as PESO_CAMISA ,
       P.ID_PANTALON , P.PANTALON , P.PESO_GR as PESO_PANTALON
  from CAMISAS C , PANTALONES P
 where C.ID_CAMISA = 1
```

ID_CAMISA	CAMISA	PESO_CAMISA	ID_PANTALON	PANTALON	PESO_PANTALON
1	lino blanca	210	1	tela azul marino	470
1	lino blanca	210	2	pana marron claro	730

Versión simplificada:

CÓDIGO:

```
select *
  from CAMISAS , PANTALONES
 where ID_CAMISA = 1
```

ID_CAMISA	CAMISA	PESO_GR	ID_PANTALON	PANTALON	PESO_GR
1	lino blanca	210	1	tela azul marino	470
1	lino blanca	210	2	pana marron claro	730

Si usted tiene dificultades en el futuro para entender o construir consultas complejas, donde intervienen varias tablas y cláusulas, siempre puede considerar de forma lógica la consulta como una sola tabla. Quizás eso le ayude en su propósito.

*** * ***

Funciones de totalización

Con la anterior premisa es fácil intuir como se comportarán las funciones de totalización. Si antes recorrían la tabla realizando un cálculo determinado, ahora recorrerán el conjunto de filas resultantes de reunir varias tablas:

¿Cuántas mudas se pueden confeccionar con las camisas y pantalones?

CÓDIGO:

```
select count(*) as COMBINACIONES
  from CAMISAS , PANTALONES
```

COMBINACIONES
6

Un recurso de la función COUNT es la de contar sobre un campo: los distintos valores que contiene, en lugar de contar todos los valores que contiene la columna. Por ejemplo:

CÓDIGO:
```
select count(*) , count(CAMISA) , count(distinct CAMISA)
  from CAMISAS , PANTALONES
```

COUNT(*)	COUNT(CAMISA)	COUNT(DISTINCT CAMISA)
6	6	3

COUNT(*) cuenta filas resultantes, COUNT(camisa) cuenta los datos de la columna CAMISA que no son nulos, en este caso coincide con el número de filas resultantes; y COUNT(DISTINCT camisa) cuenta los distintos valores que presenta la columna CAMISA, como sólo hay tres camisas distintas, el resultado de esta totalización es tres.

*** * ***

Cláusula WHERE

En todos los filtros que se han establecido en las cláusulas WHERE de las consultas SQL en este curso, hasta ahora, siempre se han condicionado campos con constantes. La potencia del SQL va más allá, pudiendo si interesa comparar o condicionar campos de un mismo registro, o fila resultante de un producto cartesiano, entre sí. Siguiendo con el conjunto resultante de combinar las camisas con los pantalones, supongamos que interesa seleccionar aquellas mudas que el pantalón y la camisa son del mismo color. Al establecer esta condición se está reduciendo el número de elementos resultantes, puesto que ahora de todas las mudas solo se seleccionarán aquellas que ambas prendas sean del mismo color. Esto se consigue con ayuda de la cláusula WHERE. Volviendo al símil del archivo, a nuestro ayudante le pediríamos para este propósito lo siguiente:

Selecciona todas las mudas confeccionables
del archivo PANTALONES y CAMISAS
donde el COLOR del PANTALON sea el mismo que el COLOR de la CAMISA

Como en las tablas no tenemos guardado el color, para ilustrar esto supongamos que las camisas tiene un orden o prioridad que viene dado por su identificador (1 , 2 , ...), para los pantalones consideramos lo mismo:

¿Qué mudas o combinaciones son aquellas que la primera camisa se combina con el primer pantalón, la segunda camisa con el segundo pantalón, y así sucesivamente?

Para ello se tiene que dar que el identificador de la camisa sea el mismo que el del pantalón, por tanto:

CÓDIGO:
```
select *
  from CAMISAS C, PANTALONES P
 where C.ID_CAMISA = P.ID_PANTALON
```

ID_CAMISA	CAMISA	PESO_GR	ID_PANTALON	PANTALON	PESO_GR
1	lino blanca	210	1	tela azul marino	470
2	algodon naranja	290	2	pana marron claro	730

Esto puede ser visto como un filtro, puesto que de las seis filas resultantes de cruzar estas dos tablas, solo tomamos dos, las que cumplen la condición, y esto sería así si en realidad se tratara de una sola tabla, pero son dos tablas, por lo que más que un filtro, que lo es, se debe hablar de reunión. Con este tipo de condiciones se están reuniendo registros entre tablas relacionadas, o si usted quiere, entre ellas guardan una relación y de este modo obtenemos los registros relacionados. En este caso existe una relación de orden. Este no es un buen ejemplo de reunión entre dos tablas, pero como vamos a hablar largo y tendido de ello en las próximas lecciones, sirva el ejemplo de introducción.

Un ejemplo de filtro que no relaciona tablas pero sí compara campos entre sí de una misma tabla es el siguiente:

CÓDIGO:
```
select *
  from PERSONAS
 where RUBIA = ALTA
```

ID_PERSONA	NOMBRE	RUBIA	ALTA	GAFAS
1	Manuel	S	S	N
2	Maria	N	N	S
4	José	S	S	S

La anterior consulta SQL devuelve los registros de la tabla PERSONAS que contienen el mismo valor en el campo RUBIA Y ALTA. En este caso concreto devuelve las personas que son altas y rubias, o bien, no son altas ni rubias. Por tanto en este, y sólo en este caso, equivale a la siguiente consulta:

CÓDIGO:
```
select *
  from PERSONAS
 where (RUBIA = 'S' and ALTA = 'S')
    or (RUBIA = 'N' and ALTA = 'N')
```

ID_PERSONA	NOMBRE	RUBIA	ALTA	GAFAS
1	Manuel	S	S	N
2	María	N	N	S
4	José	S	S	S

*** * ***

Uniones (UNION ALL)

Antes de entrar de lleno en la operación de reunión, vamos a presentar otro recurso muy potente del SQL. Se ha visto como al cruzar dos tablas, el SGBD reúne los registros mediante el producto cartesiano colocando los registros de la segunda tabla al lado de los registros de la primera tabla. Ahora vamos a ver el modo de colocar los registros de una tabla debajo los registros de otra, es decir, vamos a unir dos consultas. Para ello planteamos la siguiente pregunta:

¿Qué prendas contiene una maleta con todas las camisas y pantalones?

Con lo visto en este curso usted debería saber responder a esto, para ello necesita dos consultas, una para seleccionar todas las camisas, y otra para seleccionar todos los pantalones. Existe un modo de unir los resultados de dos o más consultas colocando entre ellas la palabra clave: UNION ALL.

Prendas de la maleta:

CÓDIGO:

```
   select concat('Camisa de ',CAMISA) as PRENDA
     from CAMISAS
union all
   select concat('Pantalón de ',PANTALON)
     from PANTALONES
```

PRENDA
Camisa de lino blanca
Camisa de algodon naranja
Camisa de seda negra
Pantalón de tela azul marino
Pantalón de pana marron claro

> “
>
> Obsérvese como para realizar la **operación de unión**, es necesario que **ambas consultas devuelvan el mismo número de columnas**, de lo contrario la consulta SQL es en conjunto errónea y SGBD no sabrá resolverla.

Una variante de UNION ALL es indicar para este mismo propósito UNION a secas, esta opción eliminará del resultado filas duplicadas, es decir, si de entre las consultas implicadas existen filas repetidas, al realizar la unión solo se quedará con una.

*** * ***

Resumen

A efectos lógicos el producto cartesiano entre varias tablas se puede considerar como una nueva tabla siendo válido todo lo expuesto sobre consultas de una sola tabla, es decir, se considera que el motor SQL primero generará el producto cartesiano sobre las tablas de la cláusula FROM, para después ejecutar el resto de la consulta sobre este resultado.

En la clausula WHERE se pueden establecer condiciones comparando campos de un mismo registro entre sí. Cuando los campos son de tablas distintas se estarán reuniendo registros entre ambas tablas bajo un concepto o propósito. El caso más amplio de relación es el producto cartesiano, donde se reúnen todos con todos.

Mediante UNION ALL o bien UNION, es posible unir dos o más consultas SQL, el resultado es la unión de resultados, por lo que todas las consultas que intervienen en la unión deben devolver el mismo número de columnas.

Obsérvese que si tomamos la tabla CAMISAS y PANTALONES y realizamos la operación de reunión se obtiene la lista de todas las mudas confeccionables, mientras que si realizamos la operación de unión, se obtiene la lista de las prendas disponibles.

*** * ***

Ejercicio 1

Construya una consulta SQL que devuelva el peso medio de todas las mudas confeccionables entre camisas y pantalones. Modifique la consulta para obtener el mismo resultado entre camisas, pantalones y calzados.

Ejercicio 2

Construya una consulta SQL que devuelva el peso medio de todas las mudas confeccionables entre camisas y pantalones agrupado por camisa. Modifique la consulta de manera que devuelva el mismo resultado pero de los grupos que el peso medio es superior a 850 gramos.

Ejercicio 3

Construya una consulta SQL que devuelva las combinaciones de las camisas con los pantalones de manera que: la primera camisa se combine con todos los pantalones menos con el primero, la segunda camisa se combine con todos los pantalones menos con el segundo, y así sucesivamente.

Ejercicio 4

Construye una consulta que devuelva la lista de prendas de una maleta que contiene todos las camisas, pantalones y calzados.

Lección 14 - Relaciones, claves primarias y foráneas

Las Relaciones es lo que, aparte de dar el nombre a las BD relacionales, hacen de este modelo una potente herramienta de reunión de datos. Para abordar las relaciones debemos tratar primero el concepto de clave primaria y clave foránea, puesto que son estas claves las que establecen las relaciones en una BD, y realizan la reunión de datos mediante consultas SQL.

Clave primaria

Si usted recuerda como su profesor de enseñanza básica pasaba lista a la clase, recordará que nombraba a los alumnos bien por su apellido, bien por su nombre, dependiendo del caso, e incluso por nombre y apellido si era necesario evitar ambigüedades. El propósito era dejar claro a quien se estaba haciendo mención y no dar lugar a dudas entre dos alumnos de igual nombre o apellido. Podríamos decir que el profesor asignaba una clave primaria a cada alumno, y con ello todo el mundo sabía que clave identificar como propia y responder: presente, al oírla.

Podemos considerar que una tabla es como una clase, y el conjunto de registros que contiene son los alumnos de esta clase. Para identificar cada registro es necesario establecer, de igual modo que hace el profesor, una clave primaria, con el propósito de identificar cada registro de forma única, por lo que el valor, o valores que ejercen de clave en un registro no se pueden repetir en el resto de registros de la tabla, ni en futuros registros que puedan existir. De esto ya se encarga el SGBD al especificarle que campos de la tabla forman la clave primaria, devolviendo un error cuando se intenta duplicar una clave primaria al insertar un nuevo registro en la tabla.

Un error común, a mi entender, al establecer la clave primaria de una tabla es intentar aprovechar algún campo de datos para que ejerza de clave, por ejemplo el DNI(documento nacional de identidad) de una persona. Aparentemente es un campo que no se puede repetir y, por tanto, es un buen candidato para ejercer de clave primaria en, por ejemplo, una tabla de empleados o de alumnos. Sin embargo no tenemos control sobre él, es decir, no podemos garantizar que no se repita. En ocasiones se asigna un DNI que perteneció a una persona ya fallecida, a un nuevo ciudadano de modo que aunque sea una posibilidad remota, puede dar problemas. Eso sin considerar que en ocasiones pueda resultar una clave poco práctica de manejar.

Otro error común es pretender que el campo clave guarde información implícita en la propia clave. Por ejemplo, a un vehículo de nuestra flota le asignamos la clave 1100456, donde el 11 está indicando que es marca SEAT y el 00456 es el resto de la clave.

Mi consejo es que no se empeñe, ni en aprovechar datos para que ejerzan de clave, ni en aprovechar claves para que implícitamente contengan información relevante. Las claves son claves y deben diseñarse únicamente para identificar registros. Los números naturales (1, 2 , 3 , etc...) son excelentes candidatos para ejercer de clave, se pueden ordenar (el SGBD creará índices sobre los campos clave que agilizarán las consultas) y son infinitos (siempre dispondremos de un valor para no repetir claves).

Clave foránea

La clave o claves foráneas de una tabla son referencias a registros de otra tabla, formándose entre ambas tablas una relación. Una registro de la tabla que tiene la clave foránea, llamémoslo registro hijo, apunta a un solo registro de la tabla a la que hace referencia, llamémoslo registro padre. Por tanto, una clave foránea apuntará siempre a la clave primaria de otra tabla.
De hecho el nombre ya nos indica que es una clave externa, es decir, el valor que contiene un registro en el campo, o campos, que ejercen de clave foránea, deberá contenerlo algún registro(uno solo) en el campo, o campos, que ejercen de clave primaria en la tabla a la que hace referencia dicha clave foránea.

Es también el SGBD quien garantiza esto, no dejando armar una clave foránea si pretendemos montarla sobre el campo, o campos, que no son clave primaria en la tabla con la que se pretende relacionar.
Tampoco permitirá, devolviendo un error, insertar valores que no existen como clave primaria en la tabla padre, o tabla a la que se hace referencia. A esto se le llama integridad referencial. El SGBD no permite incoherencias referenciales, de modo que si por ejemplo se intenta eliminar un registro padre el cual dejaría hijos huérfanos en otras tablas, es decir, tiene referencias o claves foráneas de él, el SGBD devuelve un error y no se realiza la operación.

Relaciones

El modo de relacionar registros entre tablas es por tanto mediante referencias, para lo cual se usan los identificadores definidos como claves primarias y foráneas.

Supongamos una academia donde se imparten clases, en consecuencia habrá cursos, profesores y alumnos. En nuestra base de datos diseñamos una tabla para cada entidad, es decir, para alumnos, profesores y cursos. Veamos como se relacionan entre si estas tres entidades y como se establecen estas relaciones en la base de datos.

Intuitivamente usted puede resolver la siguiente relación: La academia oferta cursos que imparten los profesores a los alumnos matriculados, y está en lo cierto, pero para relacionar esto en una BD debemos conocer en que medida se relacionan entre si estas tres entidades, es lo que se llama cardinalidad de una relación. Veamos primero el diseño de las tablas, los datos que contienen, y que campo, o campos, juegan el papel de identificador o clave primaria.
Los campos clave se han bautizado con el prefijo ID, abreviación de identificador.

TABLA CURSOS

ID_CURSO	TITULO
1	Programación PHP
2	Modelos abstracto de datos
3	SQL desde cero
4	Dibujo técnico
5	SQL avanzado

TABLA PROFESORES

ID_PROFE	NOMBRE	APELLIDOS	F_NACIMIENTO
1	Federico	Gasco Daza	1975-04-23
2	Ana	Saura Trenzo	1969-08-02
3	Rosa	Honrosa Pérez	1980-09-05

TABLA ALUMNOS

ID_ALUMNO	NOMBRE	APELLIDOS	F_NACIMIENTO
1	Pablo	Hernandaz Mata	1995-03-14
2	Jeremias	Santo Lote	1993-07-12
3	Teresa	Lomas Trillo	1989-06-19
4	Marta	Fuego García	1992-11-23
5	Sergio	Ot Dirmet	1991-04-21
6	Carmen	Dilma Perna	1987-12-04

A estas tablas se las llama "maestros", dado que contienen información relevante y concreta de cada entidad, así hablaremos del maestro de profesores o del maestro de alumnos. Bien, para establecer las relaciones entre estas tres tablas necesitamos conocer con algo más de detalle la actividad en la academia, de modo que después de investigar un poco sacamos las siguientes conclusiones:

- Cada curso lo imparte un único profesor, sin embargo algún profesor imparte más de un curso.

- Cada curso tiene varios alumnos, y algunos alumnos cursan dos o más cursos.

*** * ***

Relación de cardinalidad 1 a N

Establezcamos la siguiente relación:

Cada curso lo imparte un único profesor, sin embargo algún profesor imparte más de un curso.

Para ello basta con crear un campo en la tabla CURSOS que informe que profesor lo imparte. Este dato es una clave primaria de la tabla PROFESORES alojada en la tabla CURSOS, de ahí lo de clave foránea, por tanto el campo que ejercerá de clave foránea en la tabla CURSOS debe ser forzosamente una referencia a la clave primaria de la tabla PROFESORES.
Este tipo de relación se denomina de uno a varios, también denominada de 1 a N: un profesor imparte varios cursos, pero un curso es impartido por un único profesor. En estos casos siempre se diseña una clave foránea en la tabla hijo(CURSOS) que apunta a la tabla padre(PROFESORES).

Debemos diseñar entonces una clave foránea en la tabla CURSOS para alojar valores que son clave primaria de la tabla PROFESORES. En este caso diseñaremos un campo que llamaremos ID_PROFE, aunque se podría llamar de cualquier otro modo, que contendrá el identificador de profesor que imparte el curso que representa cada registro. Veamos como queda la tabla CURSOS:

ID_CURSO	TITULO	ID_PROFE
1	Programación PHP	3
2	Modelos abstracto de datos	3
3	SQL desde cero	1
4	Dibujo técnico	2
5	SQL avanzado	

Observando los datos de la tabla se aprecia como efectivamente cada curso lo imparte un único profesor, y que algún profesor imparte más de un curso. También se observa como uno de los curso no se le ha asignado profesor, dado que el campo ID_PROFE está a nulo. Por lo tanto una clave foránea apuntará a un solo registro de la tabla padre o no apuntará a ninguno, en cuyo caso guardará un valor indeterminado o nulo, pero jamás contendrá un valor que no exista en la tabla padre.

A usted se le puede ocurrir que es mucho más práctico y simple guardar para cada curso el nombre del profesor en lugar de claves que apenas nos dicen nada a simple vista. Esto sería transgredir la filosofía de las BD relacionales, que defienden la no duplicidad de información. El nombre de un profesor debe estar en el maestro de profesores, y cualquier referencia a ellos debe hacerse mediante su identificador. Con ello conseguimos tres cosas destacables:

- No se duplica información en la BD.

- Cualquier cambio o corrección de esa información solo debe realizarse en un único lugar.

- Evitamos la ambigüedad al no llamar la misma cosas de mil formas distintas en mil ubicaciones posibles.

Veamos la consulta que reúne el nombre de cada profesor junto al curso que imparte.

CÓDIGO:
```
select *
  from CURSOS C, PROFESORES P
 where C.ID_PROFE = P.ID_PROFE
```

ID_CURSO	TITULO	ID_PROFE	ID_PROFE	NOMBRE	APELLIDOS	F_NACIMIENTO
1	Programación PHP	3	3	Rosa	Honrosa Pérez	1980-09-05
2	Modelos abstracto de datos	3	3	Rosa	Honrosa Pérez	1980-09-05
3	SQL desde cero	1	1	Federico	Gasco Daza	1975-04-23
4	Dibujo técnico	2	2	Ana	Saura Trenzo	1969-08-02

Observe como si omitimos la cláusula WHERE de la anterior consulta, el SGBD realizaría el producto cartesiano entre los cursos y los profesores, es decir, asociaría todos los profesores con todos los cursos. El hecho de disponer de un indicador por cada curso que informa que profesor lo imparte, permite filtrar el producto cartesiano y solicitar aquellas filas que la columna ID_PROFE procedente de la tabla cursos es igual a la columna ID_PROFE procedente de la tabla PROFESORES, discriminando así las filas del producto cartesiano que carecen de sentido y obteniendo aquellas que guardan una relación.

Una lista de esto mismo mejor presentada:

CÓDIGO:
```
select concat('Curso de ',C.TITULO,', impartido por ',
       P.NOMBRE,' ',P.APELLIDOS) CURSOS
  from CURSOS C, PROFESORES P
 where C.ID_PROFE = P.ID_PROFE
```

CURSOS
Curso de Programación PHP, impartido por Rosa Honrosa Pérez
Curso de Modelos abstracto de datos, impartido por Rosa Honrosa Pérez
Curso de SQL desde cero, impartido por Federico Gasco Daza
Curso de Dibujo técnico, impartido por Ana Saura Trenzo

Las relaciones de 1 a N son quizás las más comunes en una BD y pueden verse como un padre, tabla referenciada, con muchos hijos, tabla que hace referencia a este padre. En el caso que se acaba de tratar el padre es el profesor y los hijos son los cursos que imparte dicho profesor. Todo hijo tiene forzosamente un padre, a no ser que la clave foránea pueda contener valores nulos, mientras que un padre puede tener de cero a muchos hijos.

*** * ***

Relación de cardinalidad N a M

Establezcamos la siguiente relación:

Cada curso tiene varios alumnos, y algunos alumnos cursan dos o más cursos.

Esta relación es un poco más laboriosa de establecer en la base de datos, puesto que un alumno cursa varios cursos, y a su vez, un curso es cursado por varios alumnos. Este tipo de relación se denomina de varios a varios, o bien, de N a M. Necesitamos crear una nueva tabla denominada tabla de relación, y que tiene como propósito definir la relación de N a M. La nueva tabla: ALUMNOS_CURSO, contendrá como mínimo las claves primarias de ambas tablas: ID_ALUMNO e ID_CURSO. La clave primaria de la nueva tabla la formaran ambos campos conjuntamente, y a su vez cada uno de ellos por separado será clave foránea de la tabla ALUMNOS y CURSOS respectivamente.

Echemos un vistazo a la tabla ALUMNOS_CURSO:

ID_ALUMNO	ID_CURSO
1	1
3	1
5	1
4	2
1	3
5	3
2	4
6	4

Fíjese que esta tabla contiene únicamente referencias. Cada registro establece una relación, está relacionando un registro de la tabla CURSOS con un registro de la tabla ALUMNOS.

Veamos la consulta que realiza la reunión de los alumnos con los cursos que cursa cada uno:

CÓDIGO:
```
select *
  from ALUMNOS_CURSOS AC, ALUMNOS A, CURSOS C
 where AC.ID_ALUMNO = A.ID_ALUMNO
   and AC.ID_CURSO = C.ID_CURSO
```

ID_ALUMNO	ID_CURSO	ID_ALUMNO	NOMBRE	APELLIDOS	F_NACIMIENTO	ID_TUTOR	ID_CURSO	TITULO	ID_PROFE
1	1	1	Pablo	Hernandaz Mata	1995-03-14		1	Programación PHP	3
3	1	3	Teresa	Lomas Trillo	1989-06-19		1	Programación PHP	3
5	1	5	Sergio	Ot Dirmet	1991-04-21		1	Programación PHP	3
4	2	4	Marta	Fuego García	1992-11-23		2	Modelos abstracto de datos	3
1	3	1	Pablo	Hernandaz Mata	1995-03-14		3	SQL desde cero	1
5	3	5	Sergio	Ot Dirmet	1991-04-21		3	SQL desde cero	1
2	4	2	Jeremias	Santo Lote	1993-07-12		4	Dibujo técnico	2
6	4	6	Carmen	Dilma Perna	1987-12-04		4	Dibujo técnico	2

Una lista de esto mismo mejor presentada:

CÓDIGO:
```
select C.TITULO CURSO ,
       concat(A.APELLIDOS,', ',A.NOMBRE ) ALUMNO
  from ALUMNOS_CURSOS AC, ALUMNOS A, CURSOS C
 where AC.ID_ALUMNO = A.ID_ALUMNO
   and AC.ID_CURSO = C.ID_CURSO
 order by C.TITULO , A.NOMBRE , A.APELLIDOS
```

CURSO	ALUMNO
Dibujo técnico	Dilma Perna, Carmen
Dibujo técnico	Santo Lote, Jeremias
Modelos abstracto de datos	Fuego García, Marta
Programación PHP	Hernandaz Mata, Pablo
Programación PHP	Ot Dirmet, Sergio
Programación PHP	Lomas Trillo, Teresa
SQL desde cero	Hernandaz Mata, Pablo
SQL desde cero	Ot Dirmet, Sergio

> “
>
> En este caso también podemos hacer el ejercicio de considerar el producto cartesiano entre estas tres tablas y como la cláusula WHERE permite ignorar aquellos registros del producto cartesiano que carecen de sentido y filtrar aquellos que guardan una relación.

La tabla de relación puede contener más información si es necesario, siempre y cuando la información sea vinculante tanto para el curso como para el alumno del registro en cuestión. No se tercia guardar aquí datos referentes al alumno que no tengan que ver con el curso, o datos del curso que no tengan que ver con el alumno. Por tanto registrar aquí cosas como la fecha de matrícula de un alumno en un curso, o la nota que el alumno ha sacado en un curso, tiene sentido, mientras que no lo tiene guardar aquí la fecha en que empieza un curso que nada tiene que ver con un alumno, o la veteranía del alumno en la academia que nada tiene que ver con un curso.

*** * ***

Relación de cardinalidad 1 a 1

No vamos a extendernos en esta tipo de relación puesto que no suelen darse mucho. En cualquier caso estas relaciones pueden verse como una relación 1 a N donde la N vale uno, es decir como una relación padre hijos donde el hijo es hijo único. En estos casos, cuando sólo se espera un hijo por registro padre, podemos montar la clave foránea en cualquiera de las dos tablas, aunque lo más correcto es establecerla en la tabla que NO es maestro. A efectos prácticos lo mismo da que el padre apunte al hijo que, a la inversa, es decir, que el hijo apunte al padre, o si usted quiere, cual de las dos tablas juega el papel de padre y cual de hijo. Lo importante es saber como se ha establecido la relación para atacarla mediante SQL al construir las consultas, pero siempre es preferible que la tabla maestro juegue el papel de padre.

*** * ***

Resumen

La clave primaria de una tabla permiten identificar registros de forma única, estas pueden ser simples: de un solo campo, o bien compuestas: formadas por dos o más campos. En cualquier caso los valores que toman estas claves no se pueden repetir en dos o más registros de la tabla, puesto que se perdería la funcionalidad de identificar un registro de forma única. De esto se encarga el SGBD si se ha especificado debidamente que campos son la clave primaria de la tabla.

Las claves foráneas de una tabla permiten establecer relaciones con otras tablas, puesto que contienen valores que encontramos como clave primaria en la tabla con la que se relaciona. Una clave foránea será simple o compuesta dependiendo de si lo es la clave primaria de la tabla a la que apunta o hace referencia.

Si al diseñar una tabla el campo o campos que forman una clave foránea pueden contener valores nulos, entonces el registro hijo puede no tener registro padre asociado. Esto es muy común que ocurra cuando un registro se ha creando en previsión y será en un futuro, después de que ocurra alguna cosa, que se le asignará un padre. Por ejemplo el curso sin profesor definido de la tabla CURSOS. El curso está previsto que se imparta, pero no se ha decidido o no se conoce aun que profesor lo impartirá, de ahí que el campo ID_PROFE de dicho registro contenga un valor nulo.

Las relaciones de 1 a N son quizás las que más se dan en una BD, en estos casos siempre encontraremos la clave foránea en el registro hijo apuntando al registro padre.

Las relaciones de N a M, entre por ejemplo dos tablas maestras, siempre necesitará una estructura auxiliar para establecer la relación. Esta tabla auxiliar se denomina tabla de relación, y contendrá como mínimo los campos que son clave primaria en ambos maestros. La clave primaria de la nueva tabla será siempre compuesta y estará formada por todos estos campos que son clave primaria en los maestros. A su vez estos campos por separado serán clave foránea de sus respectivos maestros. Por tanto los registros hijos se hallarán en la tabla de relación.

El modo de obtener la reunión de tablas relacionadas es mediante filtros sobre el producto cartesiano de dichas tablas, excluyendo con ayuda de la cláusula WHERE aquellos registros del producto cartesiano que carecen de sentido y obteniendo los que guardan una relación. Para ello debemos igualar la clave primaria de la tabla padre con la clave foránea de la tabla hijo.

*** * ***

Ejercicio 1

Construya una consulta que devuelva los cursos en que se ha matriculado el alumno con identificador 1.

Modifique la anterior consulta para que devuelva los nombres y apellidos de los alumnos, y los cursos en que se han matriculado, tales que el nombre de pila del alumno contenga un E.

Ejercicio 2

¿Cuántos cursos imparte cada profesor? Construya una consulta que responda a esta cuestión de modo que el resultado muestre el nombre completo del profesor acompañado del número de cursos que imparte.

Ejercicio 3

¿Cuántos alumnos hay matriculados en cada uno de los cursos? Construya una consulta que responda a esta cuestión de modo que el resultado muestre el titulo del curso acompañado de el número de alumnos matriculados.

Modifique la anterior consulta de modo que muestre aquellos cursos que el número de alumnos matriculados sea exactamente de dos alumnos.

Ejercicio 4

Si ahora a usted le pidiesen que adaptara la BD, que consta de las tres tablas presentadas en esta lección, a la siguiente necesidad: A todo alumno se le asignara un profesor que lo tutele. ¿Qué cambios realizaría en la BD?

Lección 15 - Reunión interna y externa

En la lección anterior se trató la operación de reunión entre tablas que guardan una relación. Existe una sintaxis más concreta para realizar la operación de reunión, donde la cláusula WHERE se usa únicamente para filtrar registros y no para reunir registros.

Reunión interna - cláusulas inner join / on

Esta cláusula está diseñada precisamente para reunir registros de varias tablas, en ella intervienen las claves primarias y foráneas, y no intervienen, o lo hacen en la cláusula WHERE, los filtros propiamente dichos. Veamos una de las consultas que se expuso en la lección anterior usando esta sintaxis.

Consulta que realiza la reunión entre los profesores y los cursos que imparte cada uno usando INNER JOIN / ON:

CÓDIGO:
```
select *
  from CURSOS C inner join PROFESORES P
    on C.ID_PROFE = P.ID_PROFE
```

ID_CURSO	TITULO	ID_PROFE	ID_PROFE	NOMBRE	APELLIDOS	F_NACIMIENTO
1	Programación PHP	3	3	Rosa	Honrosa Pérez	1980-09-05
2	Modelos abstracto de datos	3	3	Rosa	Honrosa Pérez	1980-09-05
3	SQL desde cero	1	1	Federico	Gasco Daza	1975-04-23
4	Dibujo técnico	2	2	Ana	Saura Trenzo	1969-08-02

Si antes se dijo que el SGBD realiza el producto cartesiano entre dos tablas y posteriormente mediante la cláusula WHERE ignora aquellos registros que carecen de sentido y muestra los que guardan una relación, ahora podemos verlo del siguiente modo: el SGBD recorrerá la tabla hijo(CURSOS) y para cada uno asociará el registro de la tabla padre(PROFESORES) que satisface la cláusula ON. Para asociar el profesor no es necesario realizar, para cada curso, un recorrido secuencial sobre la tabla PROFESORES hasta encontrarlo, puesto que en la cláusula ON estamos indicando su clave primaria, por lo que el motor SQL usará el índice que la clave lleva implícito para localizar un profesor de forma mucho más eficiente. Igual que haría usted para localizar un capítulo concreto de un libro, usando el índice.

Algunos puristas afirman que este es el modo correcto de construir las consultas, porque el motor SQL trabaja de un modo más eficiente, a otros sin embargo les resulta incomodo o simplemente menos atractivo. Lo ideal sería que todo esto fuese transparente al desarrollador. El motor SQL debe interpretar la consulta y devolver el resultado de la forma más eficiente, obviamente realizando productos cartesianos y posteriormente filtrando los registros no es un método eficiente, pero esto es algo que consideramos en este curso como herramienta de aprendizaje o método de comprensión, y que en realidad los SGBD no hacen a no ser que no les quede más remedio, ya sea porque se lo pidamos

explícitamente omitiendo la cláusula WHERE, no se hayan establecido las relaciones, o sean poco "inteligentes". Dicho de otro modo, yo espero de un SGBD que sea eficaz no solo al ejecutar la consulta, sino al interpretarla y al elaborar un plan de ejecución adecuado y eficaz, y poder usar la sintaxis que más cómoda me resulte, sin tener que pensar si la consulta es costosa o no para el motor SQL. Desafortunadamente no siempre se podrá pasar esto por alto, en ocasiones se deberá optimizar la consulta para ayudar al SGBD a ser más eficiente. En cualquier caso la optimización queda fuera del alcance de este curso. Por ahora basta con que usted sepa que es importante crear las claves primarias y foráneas debidamente, tanto por una cuestión de eficiencia como de integridad referencial.

Veamos otro ejemplo de la lección anterior usando esta cláusula, concretamente del apartado de ejercicios, donde se pedía los cursos en que se ha matriculado el alumno con identificador 1:

CÓDIGO:

```
select C.TITULO CURSO
  from ALUMNOS_CURSOS AC inner join CURSOS C
    on AC.ID_CURSO = C.ID_CURSO
 where AC.ID_ALUMNO = 1
```

CURSO
Programación PHP
SQL desde cero

Observe como en la cláusula WHERE se establece un filtro propiamente dicho, y en la cláusula ON se establece la condición de reunión que el motor debe aplicar entre las tablas a ambos lados de la cláusula INNER JOIN.

Veamos un último ejemplo de reunión interna en la que aparezcan tres tablas, para ello tomemos otro ejemplo de la lección anterior, la reunión de los alumnos con los cursos que cursa cada uno. Tomando ejemplos equivalentes construidos únicamente con la cláusula WHERE se pueden observar mejor las diferencias.

CÓDIGO:

```
  select C.TITULO CURSO ,
         concat(A.APELLIDOS,', ',A.NOMBRE ) ALUMNO
    from ALUMNOS_CURSOS AC inner join ALUMNOS A
      on AC.ID_ALUMNO = A.ID_ALUMNO inner join CURSOS C
      on AC.ID_CURSO  = C.ID_CURSO
order by C.TITULO , A.NOMBRE , A.APELLIDOS
```

CURSO	ALUMNO
Dibujo técnico	Dilma Perna, Carmen
Dibujo técnico	Santo Lote, Jeremias
Modelos abstracto de datos	Fuego García, Marta

CURSO	ALUMNO
Programación PHP	Hernandaz Mata, Pablo
Programación PHP	Ot Dirmet, Sergio
Programación PHP	Lomas Trillo, Teresa
SQL desde cero	Hernandaz Mata, Pablo
SQL desde cero	Ot Dirmet, Sergio

Si ahora sobre este consulta se quisiera reducir el resultado a un curso o un alumno en concreto, se añadiría la pertinente cláusula WHERE con el filtro deseado justo antes de la cláusula ORDER BY.

*** * ***

Reunión externa - left outer join / right outer join

La reunión externa puede verse en este caso como una reunión interna donde no es necesario que el registro hijo tenga informada la clave foránea para ser mostrado, por ejemplo, cuando se mostraban los cursos junto a los profesores que los imparten, como uno de los cursos no tiene padre, es decir, no tiene un profesor asignado, o lo que es lo mismo, el campo ID_PROFE de la tabla CURSOS está a nulo, este curso no se muestra dado que no satisface la cláusula ON. Bien, este recurso nos ofrece la posibilidad de mostrar estos registros con los campos del registro padre a nulo.

La reunión externa siempre se realizará por la izquierda o por la derecha, una de las dos. De este modo expresamos el deseo de considerar todos los registros de la tabla a la izquierda o a la derecha de la cláusula OUTER JOIN, aunque no se hallen coincidencias con la otra tabla según la cláusula ON. Veamos la consulta que muestra los cursos y sus profesores aunque el curso no tenga profesor asignado:

CÓDIGO:
```
select *
  from CURSOS C left outer join PROFESORES P
    on C.ID_PROFE = P.ID_PROFE
```

ID_CURSO	TITULO	ID_PROFE	ID_PROFE	NOMBRE	APELLIDOS	F_NACIMIENTO
1	Programación PHP	3	3	Rosa	Honrosa Pérez	1980-09-05
2	Modelos abstracto de datos	3	3	Rosa	Honrosa Pérez	1980-09-05
3	SQL desde cero	1	1	Federico	Gasco Daza	1975-04-23
4	Dibujo técnico	2	2	Ana	Saura Trenzo	1969-08-02
5	SQL avanzado					

Como en este caso usamos LEFT OUTER JOIN, la tabla que de la izquierda, es decir, la tabla CURSOS, será considerada por completo aunque no tenga éxito la cláusula ON, en cuyo caso los campos de la tabla situada a la derecha de la cláusula se mostrarán a nulo.

Si invertimos el orden de las tablas y usamos RIGHT OUTER JOIN, o simplemente RIGHT JOIN, expresión equivalente simplificada aplicable también a LEFT JOIN, el resultado es el mismo.

CÓDIGO:
```
select *
  from PROFESORES P right join CURSOS C
    on C.ID_PROFE = P.ID_PROFE
```

ID_PROFE	NOMBRE	APELLIDOS	F_NACIMIENTO	ID_CURSO	TITULO	ID_PROFE
3	Rosa	Honrosa Pérez	1980-09-05	1	Programación PHP	3
3	Rosa	Honrosa Pérez	1980-09-05	2	Modelos abstracto de datos	3
1	Federico	Gasco Daza	1975-04-23	3	SQL desde cero	1
2	Ana	Saura Trenzo	1969-08-02	4	Dibujo técnico	2
				5	SQL avanzado	

En la consulta anterior se están considerando todos los cursos aunque estos no tengan un profesor definido, si ahora usted quisiera obtener esto mismo pero añadiendo un filtro sobre la tabla PROFESORES, por ejemplo que el apellido del profesor contenga una "E", cabe esperar hacerlo en la cláusula WHERE, sin embargo también es posible aplicar el filtro en la cláusula ON. En realidad elegiremos una u otra cláusula en función de lo que deseemos obtener. Si lo hacemos en la clausula ON de un OUTER JOIN se estarán obteniendo todos los cursos con los campos de la tabla PROFESORES a nulo si la condición establecida en la cláusula ON no tiene éxito. Si se hace en la cláusula WHERE se estará forzando a que se cumpla dicha cláusula y por tanto la reunión externa se rompe. Veamos esto con un ejemplo:

Consulta que muestra todos los cursos acompañados del profesor que lo imparte. Si el curso no tiene profesor definido o bien el campo APELLIDOS no contiene una "E", los campos de la tabla PROFESORES se mostrarán a nulo:

CÓDIGO:
```
select *
  from PROFESORES P right join CURSOS C
    on C.ID_PROFE = P.ID_PROFE
   and P.APELLIDOS like '%E%'
```

ID_PROFE	NOMBRE	APELLIDOS	F_NACIMIENTO	ID_CURSO	TITULO	ID_PROFE
3	Rosa	Honrosa Pérez	1980-09-05	1	Programación PHP	3
3	Rosa	Honrosa Pérez	1980-09-05	2	Modelos abstracto de datos	3
				3	SQL desde cero	1
2	Ana	Saura Trenzo	1969-08-02	4	Dibujo técnico	2
				5	SQL avanzado	

El resultado presenta para el curso 3 los campos de la tabla PROFESORES a nulo porque el campo APELLIDOS del profesor que lo imparte no contiene un "E". Para el curso 5 ocurre lo mismo pero en este caso el motivo es además que no tiene profesor definido, con que mucho menos podrá ser cierta la otra condición.

Ahora aplicamos el filtro del apellido en la cláusula WHERE:

CÓDIGO:
```
select *
  from PROFESORES P right join CURSOS C
    on C.ID_PROFE = P.ID_PROFE
 where P.APELLIDOS like '%E%'
```

ID_PROFE	NOMBRE	APELLIDOS	F_NACIMIENTO	ID_CURSO	TITULO	ID_PROFE
3	Rosa	Honrosa Pérez	1980-09-05	1	Programación PHP	3
3	Rosa	Honrosa Pérez	1980-09-05	2	Modelos abstracto de datos	3
2	Ana	Saura Trenzo	1969-08-02	4	Dibujo técnico	2

Observamos como la reunión externa se rompe puesto que la cláusula WHERE exige que el apellido del profesor contenga una "E", dado que los cursos que no tienen profesor definido la consulta devuelve el apellido a nulo, esta cláusula no se satisface por lo que oculta el registro y la reunión externa carece de sentido, o si usted quiere, la clausula WHERE es aplicable a la tabla CURSOS pero no a la tabla PROFESORES, puesto que en este caso no tiene sentido realizar una reunión externa para que luego un filtro en la cláusula WHERE la anule.

*** * ***

Vamos ahora a ver los recuentos sobre reuniones externas, por ejemplo los alumnos que hay matriculados en cada curso. Esta consulta se presentó en la lección anterior, en el apartado de ejercicios, sin embargo los cursos sin alumnos matriculados eran ignorados en lugar de aparecer con un cero como sería de esperar. Esto es debido a que no satisfacen la clausula ON de una reunión interna, por lo que se debe usar la reunión externa para este propósito, pero cuidado, ahora no nos sirve el recuento de registros, puesto que pueden venir cursos sin alumnos, o lo que es lo mismo, cursos con los datos del alumno a nulo, de modo que si contamos registros los datos no serán verídicos, deben contarse alumnos. Revise si lo cree conveniente la lección 10 donde se trataron las particularidades del valor NULO.

Alumnos matriculados en cada curso, aunque estos sean cero:

CÓDIGO:

```
  select C.TITULO CURSO,
         count(AC.ID_ALUMNO) ALUMNOS,
         count(1) REGISTROS
    from ALUMNOS_CURSOS AC right join CURSOS C
      on AC.ID_CURSO = C.ID_CURSO
   group by C.TITULO
```

CURSO	ALUMNOS	REGISTROS
Dibujo técnico	2	2
Modelos abstracto de datos	1	1
Programación PHP	3	3
SQL avanzado	0	1
SQL desde cero	2	2

En la anterior consulta se han contado tanto alumnos como registros para poder observar la diferencia. La única fila en que estos dos valores difieren es para el curso de SQL avanzado. Dado que la reunión externa devuelve la fila con los datos del alumno a nulo para los cursos sin alumnos, al realizar un recuento de registros el valor es uno, el registro existe, pero al realizar el recuento del campo ID_ALUMNO este es ignorado por la función COUNT por ser nulo.

Observe que en este caso la tabla que interesa tratar por completo mostrando todos sus registros es la tabla padre(CURSOS), y la tabla donde no importa que haya aciertos es la tabla hijos(ALUMNOS_CURSOS). Es decir, la consulta devuelve todos los registros de la tabla CURSOS aunque para ellos no existan hijos en la tabla ALUMNOS_CURSOS. En los ejemplos anteriores a este último, también interesaba tratar por completo la tabla CURSOS, pero esta ejercía de hijo y no de padre, y los campos de la tabla PROFESORES podían venir a nulo no porque no existiera el registro en la tabla PROFESORES que también, sino como consecuencia de que el campo ID_PROFE de la tabla CURSOS contenía un valor nulo.

*** * ***

Por último comentar que la reunión externa no es posible hacerla usando únicamente la cláusula WHERE, debemos forzosamente usar la cláusula OUTER JOIN. Esto es así en MySQL, sin embargo en Oracle, que originalmente solo se podían construir consultas con las sintaxis basada en cláusula WHERE, si es posible realizar reuniones externas con esta sintaxis, para ello indicamos el símbolo"(+)" tras los campos de la clausula WHERE pertenecientes a la tabla que devolverá los campos a nulo en el caso de no cumplirse la condición, por ejemplo la consulta anterior en Oracle se construiría de la siguiente manera.

Consulta Oracle:

CÓDIGO:
```
  select C.TITULO CURSO,
         count(AC.ID_ALUMNO) ALUMNOS, count(1) REGISTROS
    from ALUMNOS_CURSOS AC ,CURSOS C
   where AC.ID_CURSO(+) = C.ID_CURSO
group by C.TITULO
```

Con ello estamos indicando el SGBD Oracle que aunque no encuentre el registro en la tabla ALUMNOS_CURSOS, devuelva el registro de la tabla CURSOS con los datos de la tabla ALUMNOS_CURSOS a nulo, es decir, se está realizando una reunión externa. Por ejemplo, en la consulta en la que se devolvía todos los cursos con el profesor que imparte cada curso y además incluíamos el filtro de que el apellido del profesor tuviese una "E" se construiría de la siguiente manera en Oracle:

Consulta Oracle:

CÓDIGO:
```
select *
  from PROFESORES P , CURSOS C
 where C.ID_PROFE = P.ID_PROFE(+)
   and P.APELLIDOS(+) like '%E%'
```

En general en una reunión externa debemos tratar siempre la tabla cuyos campos pueden venir a nulo, si es en Oracle, aunque actualmente es posible usar la sintaxis OUTER JOIN, pondremos el símbolo "(+)" a cada campo de dicha tabla que aparezca en la cláusula WHERE. Si usamos OUTER JOIN, pondremos todos los campos que establecen condiciones de dicha tabla en la cláusula ON, si lo hacemos en la cláusula WHERE como filtro corriente la reunión externa se rompe y carece de sentido.

*** * ***

Resumen

La reunión interna permite reunir registros de tablas relacionadas ignorando los registros que no satisfacen la condición de reunión especificada en la cláusula WHERE o bien en la cláusula ON en el caso de usar la sintaxis INNER JOIN.

La reunión externa permite reunir registros de tablas relacionadas considerando todos los registros de una primera tabla aunque ninguno de los registros de una segunda tabla presente aciertos contra la primera, obviamente en ese caso los campos de esta última tabla vendrán a nulo.

Existen dos sintaxis para realizar las operaciones de reunión ya sea externa o interna, dependiendo del SGBD. Basada en cláusula WHERE o bien basada en cláusula INNER JOIN / OUTER JOIN. Lo ideal sería dejar a criterio del desarrollador el uso de cualquiera de ellas siempre y cuando el SGBD lo soporte.

En una reunión externa debemos tratar siempre la tabla cuyos campos pueden venir a nulo poniendo todos los campos que establecen condiciones de dicha tabla en la cláusula ON del OUTER JOIN, si lo hacemos en la cláusula WHERE como filtro corriente la reunión externa se rompe y carece de sentido.

*** * ***

Ejercicio 1

Construya una consulta que resuelva el número de cursos que imparte cada profesor usando la cláusula INNER JOIN.

Ejercicio 2

Realice una consulta entre las tablas CURSOS, ALUMNOS y ALUMNOS_CURSOS de modo que aparezcan los alumnos matriculados en cada curso pero mostrando todos los cursos aunque no tengan alumnos matriculados.

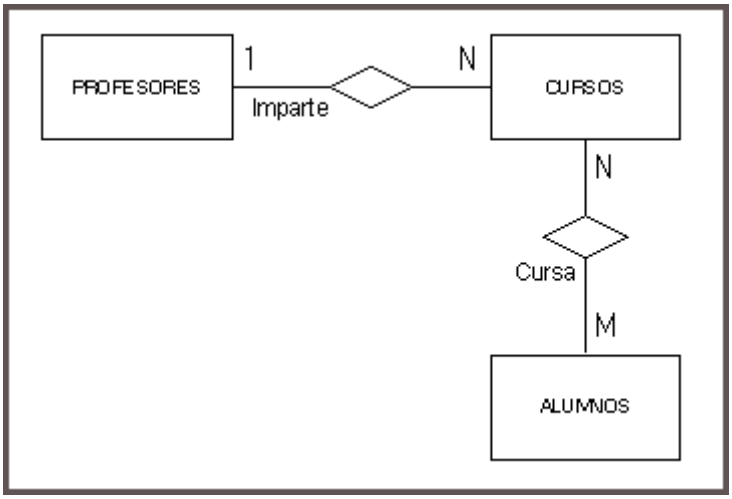
Lección 16 - El modelo entidad-relación

El modelo entidad-relación es una herramienta para generar el modelo de datos que describe la estructura y relaciones de una BD. Estos modelos al mismo tiempo están describiendo una situación real, con elementos reales que se relacionan entre sí. Por ejemplo: La actividad de un almacén de fruta, o sin ir más lejos, la actividad de este mismo foro. Obviamente no se está describiendo la actividad concreta de, por ejemplo, cargar un camión de fruta. Pero sí se está describiendo que en esta realidad (el almacén de fruta) hay una entidad llamada REPARTIDORES, que está relacionada con otro entidad llamada PEDIDOS, donde estos últimos serán adquiridos, y por tanto se relacionan, por otro entidad llamada CLIENTES, etc... Al igual que, en el caso de este foro, no se está describiendo como publicar un mensaje, pero sí que hay una entidad llamada MENSAJES, que se relaciona con otro entidad llamada USUARIOS, que a su vez se relacionan con otra entidad llamada VISITAS, etc...

El modelo entidad-relación es un diagrama que ayuda a generar la estructura de datos con la que gestionar un problema o actividad real. Una vez este modelo se ha convertido en una estructura dentro la BD, es decir, las tablas con sus claves primarias y foráneas, mediante SQL es posible tanto mantener el funcionamiento de la actividad alimentando la base de datos, como analizar los datos en beneficio de la actividad. Por ejemplo, en el caso del almacén de fruta, la estructura de datos debería permitir registrar pedidos de los clientes, pero también y en consecuencia, obtener las ventas por cliente en un periodo determinado. En el caso de este foro, la estructura de datos permite registrar nuevos usuarios, pero también conocer cuantos usuarios hay registrados hasta la fecha, o cuantos de ellos están online en un momento dado.

Este curso NO tiene el propósito directo de que usted aprenda a diseñar modelos entidad-relación. En este sentido la formación puede ayudar pero en cualquier caso, esto es algo que se adquiere con el tiempo, después de pelearse mucho diseñando modelos para actividades diversas, y equivocarse en su empeño una y otra vez. Diseñar un modelo de datos es sin lugar a dudas un ejercicio de alta creatividad, donde dependiendo de como interprete los requerimientos el analista, y su grado de imaginación, dará como fruto resultados distintos, pudiendo ser todos ellos válidos. En esencia se trata de plasmar una realidad en forma de entidades relacionadas entre sí que posteriormente será traducido a tablas dentro de una BD con sus claves primarias y foráneas.

Veamos por ejemplo el modelo entidad-relación simplificado (sin los atributos o campos de cada entidad) que describe la BD de la academia que se ha usado en las dos lecciones anteriores:



Observamos que existen tres entidades: CURSOS, PROFESORES y ALUMNOS, también se observa la cardinalidad de las relaciones mediante los indicadores a ambos lados de las mismas, junto a las entidades que se están relacionando. Para establecer la cardinalidad de relaciones debemos formularnos las preguntas que responden a dicha cuestión, por ejemplo, tomemos la relación CURSOS - PROFESORES y veamos como se establece la cardinalidad de dicha relación:

- Un profesor puede impartir **varios cursos**. Lo que implica anotar una N en el lado de la entidad CURSOS de dicha relación.

- Un curso es impartido por **un solo profesor**. Lo que implica anotar un UNO en el lado de la entidad PROFESORES de dicha relación.

Como ya se dijo con anterioridad este tipo de relación implica añadir una clave foránea de la tabla PROFESORES en la tabla CURSOS. Es decir, el campo ID_PROFE de la tabla CURSOS.

Tomemos ahora la relación CURSOS - ALUMNOS:

- En un curso se matriculan **varios alumnos**. Lo que implica anotar una N en el lado de la entidad ALUMNOS de dicha relación.

- Un alumno puede asistir a **varios cursos**. Lo que implica anotar una M en el lado de la entidad CURSOS de dicha relación.

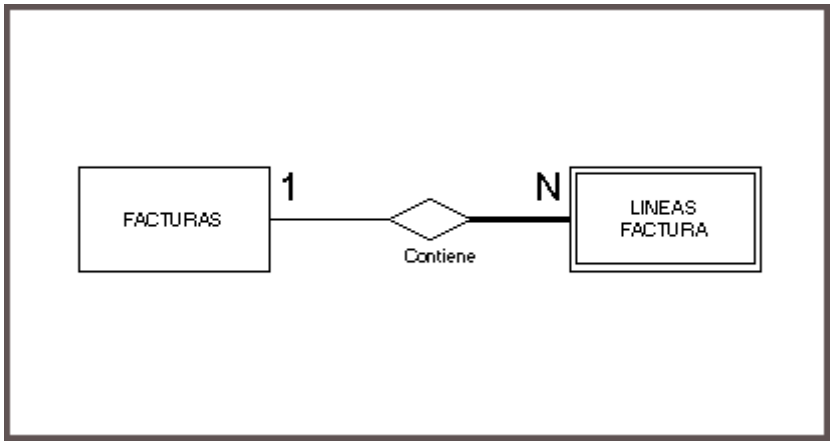
> “
>
> Obsérvese que se anota M porque la N ya se usa en el otro extremo de la relación, con esto se indica que es una relación de varios a varios pudiendo ser N y M de distinto valor para un curso y alumno dados.

Como ya se dijo con anterioridad este tipo de relación implica crear en la BD una tabla auxiliar llamada tabla de relación.

*** * ***

Entidades fuertes y débiles

Existen dos tipos de entidades, las fuertes, en ocasiones llamadas maestros, que de forma independiente identifican sus registros con un clave propia, y las débiles que dependen de una entidad fuerte para identificar sus registros, o si usted quiere, no tiene sentido su existencia sin una entidad fuerte donde apoyarse. Un ejemplo típico de entidad débil es la entidad LINEAS_FACTURA que depende del maestro de FACTURAS para identificar sus registros. La cardinalidad de esta relación es de 1 a N, puesto que una factura puede tener varias líneas mientras que una línea solo puede pertenecer a una factura. Pues bien, en la entidad débil LINEAS_FACTURA la clave primaria será compuesta y en ella formará parte el campo ID_FACTURA que a su vez será clave foránea de la tabla FACTURAS. El otro campo que formará la clave primaria será por ejemplo ID_LINEA, de modo que para identificar un registro de la entidad LINEAS_FACTURAS se necesita de la clave de su maestro o entidad fuerte además de ID_LNEA. Ejemplo: factura: 92054 línea: 3 identifica la línea 3 de la factura 92054. La cardinalidad de la relación de una entidad débil con su maestro o entidad fuerte siempre será de 1 a N. Las entidades débiles se representan en el diagrama entidad-relación con un doble rectángulo:



Lo cierto es que no afecta al funcionamiento de la gestión errar y hacer débil una entidad que en realidad es fuerte o viceversa. Sin embargo una vez adviertes que aquella parte está mal construida, el evolucionarla o simplemente explotarla se hace más incomoda al

acarrear claves compuestas cuando no deberían serlo, o el tener claves propias cuando en realidad deberían ser compuestas y estar sujetas a la entidad fuerte de la que dependen.

En ocasiones puede ser un verdadero dilema decidir en tiempo de diseño si una entidad es fuerte o débil. Si usted tiene dudas sobre que naturaleza aplicar a una entidad, puede serle de ayuda las siguientes premisas:

- Si para la entidad que se estudia su naturaleza los registros pueden cambiar de padre en un futuro, con toda seguridad es una entidad fuerte.

- Si la entidad padre de la entidad que se está estudiando su naturaleza simplemente agrupa registros siendo en ocasiones dudoso que padre asociarle a un registro hijo, o si usted quiere existen varios candidatos igual de válidos, probablemente se trate de una entidad fuerte.

- Si para la entidad que se estudia su naturaleza no se esperan demasiados registros para un mismo padre, es decir, tendrá un número de registros relativamente pequeño para un padre dado, y aparte de su posible maestro no se relaciona con apenas con otras entidades, entonces probablemente se una entidad débil.

- Si la entidad que se estudiando su naturaleza se relaciona con otras muchas entidades de modo que deberemos crear en todas ellas claves foráneas a la entidad que se está analizando, entonces aunque sea una entidad débil quizás sea conveniente valorar el identificar sus registros con una clave propia y hacerla fuerte. De otro modo deberemos acarrear la clave compuesta hacia todas estas entidades relacionadas para crear las claves foráneas.

*** * ***

El propósito de este curso no es profundizar con los modelos entidad-relación. Para acabar diremos que una vez se tiene desarrollado un modelo entidad-relación completo, no simplificado como se ha mostrado en esta lección, existe un procedimiento o protocolo para traducirlo en forma de tablas y relaciones en una base de datos. Por ejemplo, cada entidad será una tabla en la BD, las relaciones N - M obligarán a crear nuevas tablas que relacionarán las tablas que representan a cada entidad de la relación. Para las relaciones 1 - N se creará una clave foránea de la tabla de cardinalidad 1 en la tabla con cardinalidad N.

*** * ***

Resumen

El modelo entidad-relación es una herramienta en forma de diagrama que ayuda a generar la estructura de datos con la que gestionar un problema o actividad real, es decir las tablas con sus claves en una BD relacional.

Cuanto mayor sea el grado de conocimiento de la actividad a gestionar tanto mejor para desarrollar el modelo entidad-relación. Esta es un ejercicio creativo donde la teoría al respecto ayuda pero no enseña a desempeñarlo con soltura, solo se adquiere con la práctica y experiencia.

Una vez se tiene un modelo desarrollado la traducción a objetos de BD es directa aplicando una serie de pasos.

En un modelo entidad-relación encontraremos esencialmente relaciones de dos tipos: de 1 a N y, de N a M. También encontraremos dos tipos de entidades: fuertes y débiles. Una entidad débil necesitará la clave de la entidad fuerte para identificar sus registros. La cardinalidad de una relación de una entidad débil con su maestro o entidad fuerte siempre será de 1 a N.

*** * ***

Ejercicio

Modifique el modelo entidad-relación presentado en esta lección para que considere la siguiente premisa: Todo alumno tendrá un profesor que lo tutele.

Lección 17 - Funciones

Las funciones se pueden ver como cajas negras a las que les pasamos unos parámetros de entrada y tras procesarlos devuelven un único resultado o dato de salida.

Con anterioridad en este curso apareció la función CONCAT, que realiza la concatenación de dos o más cadenas de texto. Los parámetros de entrada en este caso son tantas cadenas como deseemos separadas por comas, y el resultado que devuelve es una única cadena con la concatenación de todas las cadenas de entrada.

CÓDIGO:

```
select concat('Esto ','es ','un ','ejemplo ',
              'de ','concatenación ', 'de ', 'cadenas ',
              'de ','texto.') as EJEMPLO_CONCAT
```

EJEMPLO_CONCAT
Esto es un ejemplo de concatenación de cadenas de texto.

En realidad existen infinidad de funciones con propósitos y utilidades múltiples. Las funciones están fuera del estándar SQL, cada SGBD tiene las suyas aunque existen funcionalidades presentes en todos ellos pudiendo tener diferente nombre. En esta lección veremos algunos ejemplos de funciones que por razones obvias solo podrán aplicarse al SGBD MySQL. Encontrará por Internet numerosas páginas donde documentarse sobre las funciones para este y otros SGBD.

En general una función recibe como parámetro valores, y en función de estos devuelven un resultado que es el que se considera al llamar a la función desde la cláusula SELECT de una consulta, o desde la cláusula WHERE, o desde cualquier lugar aplicable.

Alguna funciones no precisan parámetros, por ejemplo la función LOCALTIME y CURRENT_DATE. La primera devuelve la fecha y la hora del servidor de BD, la segunda solo la fecha, por tanto si usted quiere saber la fecha y la hora del servidor cuando se elaboraba este libro, aquí tiene la respuesta:

CÓDIGO:

```
select localtime , current_date
```

LOCALTIME	CURRENT_DATE
2012-04-05 11:05:57	2012-04-05

Funciones para fechas

Las funciones más usadas son quizás las de tratamiento de fechas y cadenas alfanuméricas. Veamos un ejemplo de formateo de fecha: supongamos por ejemplo que: de la fecha y hora actual solo nos interesa mostrar el mes y el año, para ello se usa la función DATE_FORMAT. Esta función precisa dos parámetros, en primer lugar el dato de tipo fecha que se quiere formatear, y seguidamente la máscara que determina el formato. Para el mes y año una mascará posible es la siguiente: '%m-%Y', por tanto la llamada a la función DATE_FORMAT para formatear una fecha con mes y año se realiza del siguiente modo:

CÓDIGO:
```
select date_format(localtime,'%m-%Y')
```

DATE_FORMAT(LOCALTIME,'%m-%Y')
04-2012

Y para que las fechas aparezcan en un formato más normal del que devuelve por defecto MySQL, al menos por lo que respecta a algunos países, la máscara es la siguiente: '%d-%m-%Y'. Así para mostrar por ejemplo los datos de la tabla EMPLEADOS con este formato para el campo F_NACIMIENTO podríamos construir la siguiente consulta:

CÓDIGO:
```
select ID_EMPLEADO, NOMBRE, APELLIDOS,
       date_format(F_NACIMIENTO,'%d-%m-%Y') F_NACIMIENTO
  from EMPLEADOS
```

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO
1	Carlos	Jiménez Clarín	03-05-1985
2	Elena	Rubio Cuestas	25-09-1978
3	José	Calvo Sisman	12-11-1990
4	Margarita	Ródriguez Garcés	16-05-1992

Si se quieren usar barras en lugar de guiones para separar el día mes y año en una dato de tipo fecha, tan solo deberá indicarlo en la máscara como se muestra en el siguiente ejemplo:

CÓDIGO:
```
select ID_EMPLEADO, NOMBRE, APELLIDOS,
       date_format(F_NACIMIENTO,'%d/%m/%Y') F_NACIMIENTO
  from EMPLEADOS
```

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO
1	Carlos	Jiménez Clarín	03/05/1985
2	Elena	Rubio Cuestas	25/09/1978
3	José	Calvo Sisman	12/11/1990
4	Margarita	Ródriguez Garcés	16/05/1992

Para conocer todas las posibilidades que ofrece MySQL o cualquier otro SGBD en lo que a máscaras de formato de fechas se refiere deberá consultar la documentación de cada sistema en particular. Encontrará numerosas páginas en Internet con información al respecto.

Veamos ahora una función que opera con fechas de modo que permite, por ejemplo, sumar días a una fecha obteniendo como resultado una nueva fecha. Para ello usamos la función de MySQL DATE_ADD.

CÓDIGO:
```
select date_add(current_date, INTERVAL 30  DAY)
       as FECHA_ACTUAL_MAS_TREINTA_DIAS,
       date_add(current_date, INTERVAL 6 MONTH)
       as FECHA_ACTUAL_MAS_SEIS_MESES
```

FECHA_ACTUAL_MAS_TREINTA_DIAS	FECHA_ACTUAL_MAS_SEIS_MESES
2012-05-05	2012-10-05

El valor que devuelve la función DATE_ADD es un dato de tipo fecha, de modo que es posible usar la llamada a DATE_ADD como parámetro en la función DATE_FORMAT para darle formato al resultado que devuelve DATE_ADD:

CÓDIGO:
```
select date_format(date_add(current_date, INTERVAL 30
       DAY) , '%d-%m-%Y') as FECHA_ACTUAL_MAS_TREINTA_DIAS,
       date_format(date_add(current_date, INTERVAL 6
       MONTH) , '%d-%m-%Y') as FECHA_ACTUAL_MAS_SEIS_MESES
```

FECHA_ACTUAL_MAS_TREINTA_DIAS	FECHA_ACTUAL_MAS_SEIS_MESES
05-05-2012	05-10-2012

Por último en lo que a funciones de fecha se refiere, aunque existen muchas más, veremos un ejemplo de la función DATEDIFF, que devuelve los días de diferencias entre dos fechas. Si usted recuerda la tabla VEHICULOS, donde se guardaba para cada unidad la fecha de la próxima revisión, se preguntará quizás como realizar una consulta que informe de los vehículos que deben pasar la revisión en los próximos 30 días, para ello supondremos que hoy es 15 de noviembre de 2009:

CÓDIGO:

```
select *
  from vehiculos
 where datediff(PROX_ITV,'2009-11-15') < 31
```

ID_VEHICULO	MARCA	MODELO	PROX_ITV	ULTI_ITV
2	Seat	Panda	2009-12-01	2008-12-01

*** * ***

Funciones para cadenas

Otro tipo de funciones son las de tratamiento de cadenas. Ya se ha visto al principio de esta lección la función CONCAT, que permite concatenar cadenas. Con estas funciones podemos obtener subcadenas de una cadena dada, por ejemplo los cuatro primeros caracteres. Para ello usaremos SUBSTR abreviación de substring, es decir, subcadena. Como parámetros recibe el dato de tipo cadena a tratar en primer lugar, seguido de la posición dentro de la cadena donde se quiere obtener la subcadena, y por último la longitud o número de caracteres de esta. Ejemplos:

CÓDIGO:

```
select substr('ABCDEFGHIJ',1,4) LOS_CUATRO_PRIMEROS_CARACTERES
```

LOS_CUATRO_PRIMEROS_CARACTERES
ABCD

CÓDIGO:

```
select substr('ABCDEFGHIJ',4,3) LOS_TRES_CARACTERES_CENTRALES
```

LOS_TRES_CARACTERES_CENTRALES
DEF

CÓDIGO:

```
select substr('ABCDEFGHIJ',3)
       as LA_CADENA_IGNORANDO_LOS_DOS_CARACTERES_INICALES
```

LA_CADENA_IGNORANDO_LOS_DOS_CARACTERES_INICALES
CDEFGHIJ

CÓDIGO:
```
select substr('ABCDEFGHIJ',-2) LOS_DOS_CARACTERES_FINALES
```

LOS_DOS_CARACTERES_FINALES
IJ

Con la función LENGTH se obtiene la longitud de una cadena:

CÓDIGO:
```
select length('ABCDEFGHIJ')
```

LENGTH('ABCDEFGHIJ')
10

La función REPLACE es de gran utilidad, remplaza en una cadena un texto por otro. Por ejemplo, imagine que usted vende manteles de varios colores y en función de un campo de tabla que vendrá de un filtro seleccionado por el usuario le viene el valor naranja, aunque el usuario podría haber seleccionado otros colores disponibles. Mediante la siguiente consulta podría establecer un texto fijo con una subcadena a remplazar por el color, que es variable y depende de lo que el usuario selecciona. La idea del siguiente ejemplo es que el literal 'naranja' seria en realidad un campo de tabla con el valor 'naranja'.

CÓDIGO:
```
select REPLACE('Mantel de color &','&','naranja') PRODUCTO
```

PRODUCTO
Mantel de color naranja

*** * ***

La función IF

Hablemos ahora de una función un poco particular pero de suma utilidad, la función IF de MySQL. En Oracle se llama DECODE y funciona de forma un poco distinta. Permite condicionar el valor que devuelve en función de si se cumple una condición que se establece. Si usted recuerda la tabla personas, donde se guardaba una 'S' para indicar Sí, y una 'N' para indicar No, con la función IF podemos dar una salida de resultados más humana decodificando esta codificación:

CÓDIGO:
```
select NOMBRE , if(RUBIA='S','Sí','No') RUBIA
  from PERSONAS
```

NOMBRE	RUBIA
Manuel	Sí
Maria	No
Carmen	Sí
José	Sí
Pedro	No

La función IF en este caso interroga si el campo RUBIA contiene un 'S', si es así devuelve 'Sí', en caso contrario devuelve 'No'

* * *

Funciones numéricas

Por último comentaremos algunas funciones para trabajar con números. Por ejemplo la función ROUND, que permite redondear un número a por ejemplo dos decimales, con lo que evitamos largas ristras de números en los resultados. O TRUNCATE que trunca un número por la parte decimal que se le indique, pudiendo así considerar únicamente la parte entera:

CÓDIGO:
```
select round(7.64739836953 , 2) , truncate(7.64739836953 , 0)
```

ROUND (7.64739836953 , 2)	TRUNCATE (7.64739836953 , 0)
7.65	7

Consulte la documentación del SGBD que le ocupe para obtener más información al respecto.

* * *

Resumen

Las funciones esperan parámetros de un tipo de dato determinado y devuelven un valor de un tipo de dato determinado. El número de parámetros y el tipo de dato de cada parámetro depende de la especificación de cada función, al igual que el tipo de dato que devuelve cada una y de su funcionalidad.

Las funciones permiten obtener valores en función de los parámetros que se le pasa para mostrarlos u operar con ellos. Los parámetros pueden ser constantes, campos de tabla, o

bien llamadas a una función. En este último caso el valor que devuelve la función ejerce de parámetro y por tanto será el dato que considerará la función que lo toma como parámetro.

Si la llamada se realiza en la cláusula SELECT el valor que devuelve la función se mostrará como un campo más de tabla, es apropiado entonces rebautizar la columna con un alias. Si se usa en la clausula WHERE el valor que devuelve formará parte de una condición que se evaluará como un campo más de tabla para mostrar o ignorar el registro. También es posible hacer la llamada a una función en la cláusula GROUP BY, si también se ha hecho en la cláusula SELECT y se pretende agrupar por esa columna.

No deben confundirse este tipo de funciones con las funciones de totalización (SUM, AVG, etc...) estas últimas operan con todos los registros seleccionados de una consulta, mientras que las primeras operan únicamente con valores de un solo registro, o si usted quiere, se llama a la función tantas veces como registros devuelve la consulta, y el resultado de la función forma parte de la fila resultante de cada registro.

*** * ***

Ejercicio 1

Realice una consulta que devuelva la media de salarios de la tabla EMPLEADOS agrupado por sexo. Redondee la media de salarios a un solo decimal y decodifique la columna sexo para que aparezca el literal HOMBRES y MUJERES en lugar de H y M. No olvide rebautizar las columnas con un alias apropiado.

Ejercicio 2

Realice una consulta sobre la tabla EMPLEADOS que devuelva el nombre, los apellidos, la fecha de nacimiento y la edad actual en años de cada empleado. Para aquellos empleados con 18 años o más.
Nota: la edad de un empleado en años es el número de días transcurridos desde el nacimiento dividido entre los 365 días que tiene un año.

Ejercicio 3

Realice una consulta sobre la tabla vehículos que devuelva el número de vehículos que deben pasar la revisión agrupado por el año en que deben pasarla.

Lección 18 - INSERT, UPDATE, DELETE

En esta lección vamos a ver como modificar la BD. Los registros de una tabla pueden ser modificados de tres modos: Crear nuevos registros, modificarlos o bien eliminarlos. No se va a profundizar en este sentido, esto lo dejaremos para un posible curso avanzado de SQL.
Por razones obvias no podrá usar el banco de pruebas, accesible desde la web deletesql.com, para probar estas instrucciones, de modo que para ello deberá usar su propia BD.

Insert SQL

La instrucción INSERT permite crear o insertar nuevos registros en una tabla, veamos su sintaxis con un ejemplo práctico, la inserción de un registro en la tabla ALUMNOS:

CÓDIGO:
```
insert
  into ALUMNOS (ID_ALUMNO , NOMBRE , APELLIDOS , F_NACIMIENTO)
values (1 , 'Pablo' , 'Hernandaz Mata' , '1995-03-14')
```

Observe como todo lo que se explicó en referencia a los tipos de datos es válido para la instrucción INSERT. Los datos de tipo numérico no se entrecomillan, a diferencia de los datos de tipo cadena y fecha.

En general la sintaxis de la instrucción INSERT es la siguiente:

CÓDIGO:
```
INSERT INTO nombre_tabla (lista de campos separados por comas)
VALUES (lista de datos separados por comas)
```

Donde cada dato de la lista VALUES se corresponde y se asigna a cada campo de la tabla en el mismo orden de aparición de la sentencia INSERT. Cabe mencionar que si la clave primaria que identifica el registro que se pretende insertar ya la usa un registro existente el SGBD rechazaría la operación y devolvería un error de clave primaria duplicada.

Así que cuando usted rellena un formulario en Internet por ejemplo, y los datos son almacenados en una BD, en algún momento del proceso se realizará una instrucción INSERT con los datos que usted a cumplimentado.

*** * ***

Update SQL

La instrucción UPDATE permite actualizar registros de una tabla. Debemos por lo tanto indicar que registros se quiere actualizar mediante la cláusula WHERE, y que campos mediante la cláusula SET, además se deberá indicar que nuevo dato va a guardar cada campo.
Así por ejemplo supongamos que para el curso que carecía de profesor finalmente ya se ha decidido quien lo va a impartir, la sintaxis que permite actualizar el profesor que va a impartir un curso sería la siguiente:

CÓDIGO:
```
update CURSOS
   set ID_PROFE = 2
 where ID_CURSO = 5
```

Todo lo expuesto sobre lógica booleana es válido para la clausula WHERE de la instrucción UPDATE, en todo caso dicha cláusula se comporta igual que en una consulta, solo que ahora en lugar de seleccionar registros para mostrarnos algunos o todos los campos, seleccionará registros para modificar algunos o todos sus campos. Por lo tanto omitir la cláusula WHERE en una instrucción UPDATE implica aplicar la actualización a todos los registros de la tabla.

La instrucción anterior asignará un 2 en el campo ID_PROFE de la tabla CURSOS en los registros cuyo valor en el campo ID_CURSO sea 5. Como sabemos que el campo ID_CURSO es la clave primaria de la tabla, tan solo se modificará un solo registro si es que existe. Obviamente en este caso, dado que el campo que se pretende actualizar es clave foránea de la tabla PROFESORES, si no existe un registro en dicha tabla con identificador 2 el SGBD devolverá un error de clave no encontrada.

Veamos otro ejemplo, esta vez se modificarán varios campos y registros con una sola instrucción. Recordemos la tabla EMPLEADOS, en ella se guardan los datos de cada empleado, el sueldo y supongamos que también se guarda en el campo PRECIO_HORA el precio de la hora extra que cobra cada empleado en el caso que las trabaje. Bien, con el cambio de ejercicio se deben subir los sueldos y el precio por hora extra trabajada, digamos que un 2% el sueldo y un 1 % el precio de la hora extra. Sin embargo la política de empresa congela el salario a directivos que cobran 3000 euros o más. ¿Qué instrucción actualizaría estos importes según estas premisas? :

CÓDIGO:
```
update EMPLEADOS
   set SALARIO = SALARIO * 1.02
       PRECIO_HORA = PRECIO_HORA * 1.01
 where SALARIO < 3000
```

Por lo tanto solo se está actualizando el salario y el precio de la hora extra de aquellos empleados que su salario es inferior a 3000 euros.

En general la sintaxis de la instrucción UPDATE es la siguiente:

CÓDIGO:

```
UPDATE nombre_tabla
   SET campo1 = valor1,
       campo2 = valor2,
       ...,
       campoN = valorM
 WHERE condicines
```

*** * ***

Delete SQL

La instrucción DELETE permite eliminar registros de una tabla, su sintaxis es simple, puesto que solo debemos indicar que registros deseamos eliminar mediante la cláusula WHERE. La siguiente consulta elimina todos los registros de la tabla mascotas que están de baja:

CÓDIGO:

```
delete from MACOTAS
 where ESTADO = 'B'
```

Del mismo modo que ocurre con la instrucción UPDATE, para la instrucción DELETE es válido todo lo expuesto sobe la cláusula WHERE para consultas.

La siguiente instrucción elimina todos los registros de la tabla VEHICULOS:

CÓDIGO:

```
delete
  from VEHICULOS
```

Al eliminar registros de una tabla estos no deben figurar como clave foránea en otra tabla, de lo contrario el SGBD devolverá un error de violación de integridad referencial, puesto que si se permitiese quedarían registros huérfanos.

En general la sintaxis de la instrucción DELETE es la siguiente:

CÓDIGO:

```
DELETE
  FROM nombre_tabla
 WHERE condiniones
```

*** * ***

Resumen

Con las instrucciones INSERT, DELETE y UPDATE el SGBD permite crear eliminar o modificar registros.

La cláusula WHERE de las instrucciones DELETE y UPDATE se comporta igual que en las consultas y permite descartar o considerar registros mediante condiciones por la instrucción de actualización o de borrado. Omitir la cláusula WHERE implica aplicar la operación a todos los registros de la tabla.

Al insertar eliminar o actualizar datos, deben respetarse las restricciones. Si estas están montadas en la BD, cosa por otro lado muy recomendable, podemos tener errores de tres tipos:

- Clave primaria duplicada (Al insertar o modificar un registro).
- Violación de integridad referencial (se pretende dejar huérfanos registros que apuntan al registro padre al intentarlo eliminar o modificar).
- Clave padre no encontrada (al actualizar o insertar una clave foránea que no existe en la tabla padre a la que apunta)

*** * ***

Ejercicio 1

Construya una instrucción de inserción en la tabla CURSOS para guardar un nuevo curso de pintura y asígnele una clave que no entre en conflicto con la existentes, posteriormente construya la instrucción para eliminar de la tabla el registro que acaba de crear.

Ejercicio 2

En esta lección se puso como ejemplo la actualización del salario de los empleados donde este se incrementaba un 2% para empleados con un sueldo inferior a 3000 euros. Sin embargo no parece muy justo que un empleado con un sueldo de 3000 Euros no reciba incremento alguno, y otros que rozan los 3000 euros pero no llegan reciban el incremento superando el importe de corte una vez aplicado dicho incremento. Construya una instrucción de actualización, que se debería ejecutar previamente, de modo que evite que para estos empleados el resultado del incremento sea superior a 3000 euros. Para ello esta instrucción debe actualizar el salario de los empleados afectados a 3000 euros, para que cuando se realice el incremento no se les aplique la subida puesto que su sueldo será entones de 3000 euros justos.

Lección 19 - Síntesis de la segunda parte

Reunión

En esta segunda parte se ha profundizado sobre la operación de reunión. El producto cartesiano es el caso más general, donde se combinan todos los registros de una primera tabla con todos los registros de las otras tablas que intervienen en la operación de reunión, esto ocurre cuando se omite la cláusula WHERE de la consulta SQL.

Podemos usar una sintaxis concreta para la operación de reunión mediante las cláusulas FROM INNER JOIN, donde se establecen a ambos lados de la expresión las tablas que van a intervenir en la operación, más la cláusula ON, donde se establecen las condiciones que ha de satisfacer la operación. La alternativa a esta sintaxis es usar las cláusulas FROM WHERE con idéntico propósito.

Totalización

Cuando realizamos una operación de reunión, a efectos lógicos, podemos considerar que el motor SQL realiza el producto cartesiano entre las tablas involucradas y posteriormente aplica sobre el resultado los filtros que establece la cláusula WHERE. Para las funciones de totalización, más comúnmente conocidas como funciones de agregado, podemos considerar lo mismo, es decir, que el cálculo se realiza sobre los registros resultantes del producto cartesiano tras ser filtrado por la cláusula WHERE.

Reunión externa

La reunión externa permite obtener datos de una tabla aunque no tenga éxito la condición que estable la operación, o lo que es lo mismo, en el resultado de la reunión externa las columnas referentes al registro padre vendrán a nulo porque la clave foránea del registro hijo contiene un valor nulo. Esto es así cuando la tabla que manda es la tabla hijo. Puede darse también a la inversa, es decir, que la consulta se construya de modo que mande la tabla padre, de modo que los registros padre que no tengan hijos se muestren con los campos de la tabla hijo a nulo en el caso de no existir ningún registro hijo para ese padre. La cláusula que permite realizar esta operación es OUTER JOIN. Que mande una u otra tabla dependerá de si aplicamos la reunión externa por la izquierda o por la derecha, es decir, de si aplicamos entre ellas LEFT OUTER JOIN o RIGHT OUTER JOIN, manteniendo las tablas que interviene fijas a ambos lados del OUTER JOIN.

*** * ***

Claves primarias y foráneas

La clave primaria de una tabla permite identificar de forma única cada registro de una tabla mediante los propios datos que contienen los campos que forman la clave primaria. Esta puede ser simple o compuesta, es decir, formada por uno o por varios campos. Los datos

que contiene un registro en los campos que establecen la clave primaria en una tabla no se pueden repetir para ningún otro registro de la tabla, garantizándose de este modo la exclusividad y la identificación unívoca del registro.

La clave foránea de una tabla relaciona el registro que contiene la clave foránea con el registro de la tabla padre que contiene el mismo valor en su clave primaria. Dicho de otro modo, la clave foránea de una tabla siempre existirá como clave primaria en la tabla a la que apunta y con la que se relaciona. En el caso de que el campo o campos que forman la clave foránea pueden contener valores nulos, el registro puede no estar asociado a ningún registro padre.

Relaciones y cardinalidad

Existen principalmente dos tipos de relación entre tablas de una BD según la cardinalidad de la relación.

- Relación de 1 a N
- Relación de N a M

La relación de cardinalidad 1 a N entre dos tablas establece que por cada registro de la tabla padre se esperan varios registros de la tabla hijos, pudiendo no tener ninguno. También el registro hijo puede no tener padre definido, en cuyo caso el campo o campos que apuntan al padre contendrán un valor nulo.

La relación de cardinalidad N a M entre dos tablas precisa de una tabla auxiliar o tabla de relación para establecer o definir la relación. Si dos registros concretos de dos tablas que guardan una relación N a M NO se relacionan entre sí, significa que NO aparecerá registro alguno en la tabla de relación que apunten a estos dos registros. En una tabla de relación la clave primaria será compuesta y estará formada por los campos que son clave primaria en las tablas o maestros de los que deriva. Estos campos, los que forman la clave primaria en cada maestro, de la tabla de relación serán a su vez por separado claves foráneas de sus respectivos maestros.

*** * ***

Modelo entidad-relación

Es un diagrama donde se representan entidades de una realidad o actividad y sus relaciones. Es de gran ayuda para posteriormente definir la estructura de tablas de una BD con sus relaciones.

Desarrollar modelos relacionales precisa un análisis e investigación previa de la actividad que se pretende gestionar. Cuantos más datos se tengan mejor y con menos errores se podrá desarrollar el modelo entidad-relación. Por otro lado desarrollar un modelo es una ejercicio creativo que precisa de una dilatada experiencia y práctica para dominarlo.

Una vez se obtiene el diagrama del modelo entidad-relación, se obtiene de forma directa la estructura de BD aplicando un protocolo que se rige por una serie de pasos, estos son los más significativos:

- Cada entidad será una tabla de la base de datos.
- Las relaciones de 1 a N implican crear una clave foránea en la tabla que deriva de la entidad con cardinalidad N, que apunte a la tabla que deriva de la entidad con cardinalidad 1.
- Las relaciones de N a M exigen crear una nueva tabla de relación en la base de datos.

En el modelo entidad relación encontraremos entidades fuertes y débiles. Las entidades fuertes tienen claves propias para identificar sus registros, mientras que las débiles deben apoyarse en una entidad fuerte para identificarlos. La clave primaria de una entidad débil siempre será compuesta y en ella intervendrá la clave foránea que apunta a la entidad fuerte de la que depende. La relación entre ambas entidades siempre será de cardinalidad 1 a N.

*** * ***

DML (data manipulation language)

Las instrucciones DML, lenguaje de manipulación de datos, son:

- **SELECT**: esta instrucción permite obtener datos de una BD, reunirlos, unirlos, calcular, etc...
- **INSERT**: instrucción que permite insertar nuevo registros en una tabla de la BD.
- **DELETE**: permite eliminar registros de una tabla.
- **UPDATE**: esta instrucción posibilita la modificación de datos de una tabla, es equivalente a eliminar primero el registro mediante la instrucción DELETE y volverlo a crear con la instrucción INSERT. En ocasiones la integridad referencial hará imposible eliminarlo y volverlo a crear directamente debiendo usar forzosamente la instrucción UPDATE.

Al usar estas instrucciones, salvo la instrucción SELECT, se deben tener en cuenta las restricciones de la BD. Así pues el SGBD devolverá los siguientes errores si se pretende violar la **integridad referencial**.

- **Clave padre no encontrada**: al insertar o modificar un registro donde se está indicando una clave foránea que no existe en la tabla padre.
- **Clave primaria duplicada**: al insertar o modificar un registro donde se está indicando una clave primaria que ya existe en otro registro de la misma tabla.
- **Registros dependientes encontrados**: al intentar eliminar un registro al que uno o más registros hijos de otras tablas hacen referencia, dicho de otro modo, al intentar dejar huérfanos registros hijos del registro padre que se pretende eliminar.

* * *

Funciones

Las funciones permiten formatear fechas, manipular cadenas, redondear números... y un sinfín de utilidades que son de gran ayuda sobretodo en las consultas SQL, aunque también es posible aplicarlas a cualquiera de las instrucciones DML.

Las funciones esperan parámetros con los que, en función de estos y aplicados a un algoritmo, retornar un valor como resultado. Este valor será considerado como un campo de tabla o un valor constante por la instrucción DML. El tipo de dato de los parámetros y el tipo de dato que retorna la función dependerá de la especificación de cada función.

Las funciones en ocasiones permiten agrupar por valores que de otro modo sería imposible, por ejemplo, agrupar datos por año cuando la fuente del año es un campo de tipo fecha. Otras veces nos simplificarán el trabajo pudiendo obtener el resultado en la propia consulta evitando así un tratamiento posterior por código, es decir, desde el programa donde se lanza la consulta, por ejemplo php o java.

* * *

Ejercicio 1

Supongamos que tenemos las siguientes entidades en un modelo relacional que gestiona la liga profesional de fútbol: EQUIPOS y JUGADORES. La cardinalidad de esta relación es 1 a N, puesto que un equipo tiene una plantilla de N jugadores mientras que un jugador milita en un solo equipo. ¿Es JUGADORES una entidad débil?

Ejercicio 2

Supongamos que tenemos las siguientes entidades en un modelo relacional que gestiona las reparaciones del alumbrado público de una urbanización: FAROLAS y REPARACIONES. La cardinalidad de esta relación es 1 a N, puesto que a una farola se le realizan N reparaciones mientras que una reparación se practica a una farola. ¿Es REPARACIONES una entidad débil?

Ejercicio 3

Supongamos que tenemos las siguientes entidades en un modelo relacional que gestiona la actividad de un almacén de distribución de género: ARTICULOS y FAMILIAS. La cardinalidad de esta relación es 1 a N, puesto que una familia agrupa N artículos mientras que un artículo pertenece a una sola familia. ¿Es ARTICULOS una entidad débil?

Lección 20 - Aplicación SQL

En esta última parte, y última lección del curso, vamos a desarrollar a modo de ejemplo una pequeña aplicación donde se pretende, además de ver como crear las tablas y claves en la BD, mostrar las fases a nivel de BD para el desarrollo de una aplicación.

Así pues vamos primero a esquematizar las fases de esta aplicación de ejemplo.

- Toma y análisis de requerimientos.
- Modelo entidad-relación.
- Creación de la estructura o modelo de datos.
- Creación de claves primarias y foráneas de las tablas.
- Inserción de registros en las tablas.
- Informes o explotación de datos.

Análisis de requerimientos.

Para analizar los requerimientos primero debemos conocer cuales son, así pues consideremos lo siguiente:

Requerimientos

Se necesita una aplicación que permita gestionar las apuestas de quinielas futbolísticas para un único apostante, es decir, para uno mismo. La aplicación deberá ser capaz de escrutar los pronósticos una vez se tiene el resultado de la jornada, e informar de cuantos aciertos se han logrado, no es necesario que gestione los gastos y premios ni apuestas múltiples(con dobles y/o triples). En general deberá permitir mantener los datos referentes a las jornadas de Liga y a las quinielas para explotar los datos referentes a los aciertos.

Análisis

Bien, dadas estas especificaciones debemos conocer esencialmente como funcionan las quinielas futbolísticas. En el mundo del fútbol podemos afirmar que se convocan jornadas en las que se organizan eventos, en estos eventos participan equipos, y sobre estos eventos se pronostican resultados a 1,X, 2. El conjunto de pronósticos sobre los eventos de una misma jornada es lo que se llama quiniela. Y el conjunto de resultados de los eventos de una misma jornada es lo que se llama combinación ganadora.

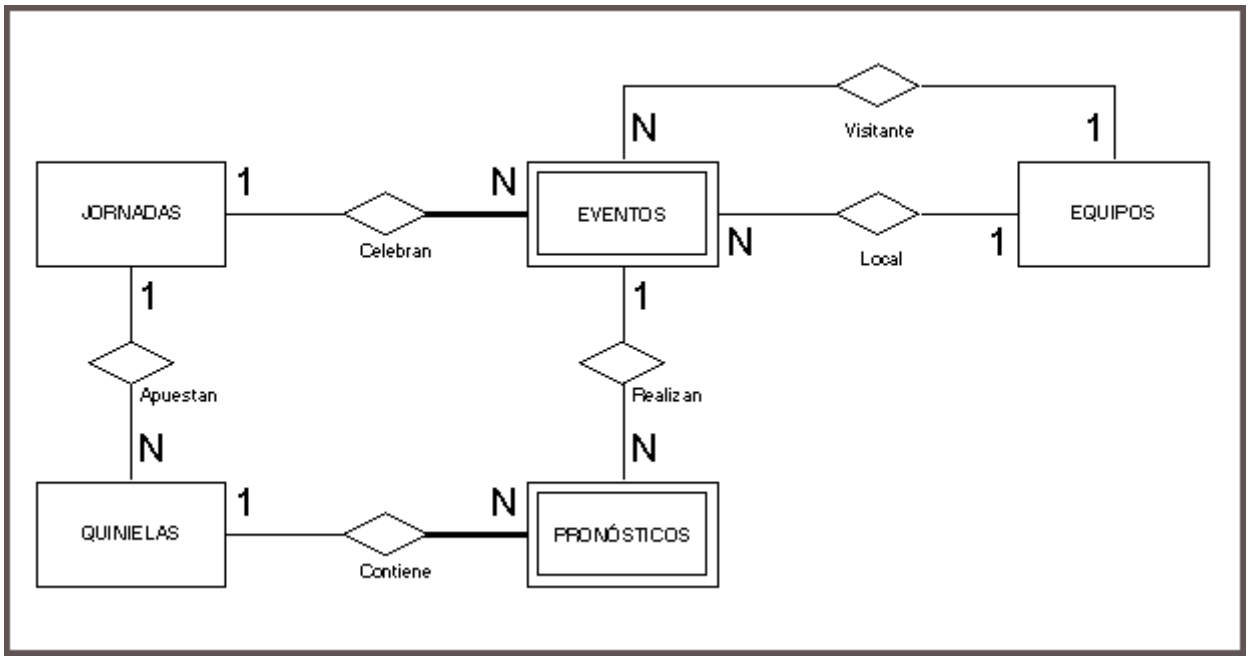
Entidades

Del análisis anterior proponemos las siguientes entidades:

- EQUIPOS
- JORNADAS
- EVENTOS
- QUINIELAS
- PRONOSTICOS

Modelo entidad-relación

Mostremos primero el diagrama resultante:



Y estos son los razonamientos que se hicieron mientras se elaboraba el diagrama de los cuales se obtiene la cardinalidad de las relaciones.

- En una jornada se organizan varios eventos mientras que un evento se celebra en una jornada. Eventos es una entidad débil dependiente de jornadas.
- Un equipo participa en varios eventos como local mientras que en un evento sola hay un equipo local.
- Un equipo participa en varios eventos como visitante mientras que en un evento sola hay un equipo visitante.
- En una jornada se sellan varias quinielas mientras que una quiniela solo es válida para una jornada.
- En una quiniela se realizan varios pronósticos mientras que un pronóstico pertenece a una quiniela concreta. Pronósticos es una entidad débil dependiente de quinielas.
- Sobre un evento se realizan varios pronósticos mientras que un pronóstico pertenece a un evento concreto.

Las entidades EVENTOS y PRONOSTICOS se han considerado débiles. Por lo tanto la clave primaria de estas entidades será compuesta interviniendo en ella la clave foránea de la entidad fuerte de la que dependen. EVENTOS depende de JORNADAS, y PRONOSTICOS depende de QUINIELAS.

Atributos de cada entidad

Este apartado se debería integrar en el modelo entidad-relación, en su lugar lo haremos aparte con la intención de no cargar el diagrama, de modo que el modelo presentado con anterioridad es un modelo simplificado. En la primera columna de la siguiente propuesta de atributos se especifica el nombre del atributo, en la segunda el tipo de dato, en la tercera si puede ser o no nulo, y en la cuarta columna, si se tercia, si es clave primaria, foránea, o los posibles valores si es un campo codificado.

EQUIPOS:

CAMPO	**TIPO**	**NULO?**	**COMENTARIO**
ID_EQUIPO	numérico	no nulo	clave primaria
EQUIPO	cadena(30)	no nulo	

JORNADAS:

CAMPO	**TIPO**	**NULO?**	**COMENTARIO**
ID_JORNADA	numérico	no nulo	clave primaria
NOMBRE	cadena(30)	no nulo	
FECHA	fecha	no nulo	
DISPUTADA	cadena(1)	no nulo	posibles valores: ('S' , 'N') S -> sí , N -> No

EVENTOS:

CAMPO	**TIPO**	**NULO?**	**COMENTARIO**
ID_JORNADA	numérico	no nulo	clave primaria
ID_EVENTO	numérico	no nulo	clave primaria
LOCAL	numérico	no nulo	clave foránea de la entidad EQUIPOS
VISITANTE	numérico	no nulo	clave foránea de la entidad EQUIPOS
RESULTADO	cadena(1)	nulo	posibles valores: ('1' , 'X' , '2')

QUINIELAS:

CAMPO	TIPO	NULO?	COMENTARIO
ID_QUINIELA	numérico	no nulo	clave primaria
NOMBRE	cadena(30)	nulo	
ESCRUTADA	cadena(1)	no nulo	posibles valores: ('S', 'N') S -> sí, N -> No
ACIERTOS	numérico	nulo	

PRONOSTICOS:

CAMPO	TIPO	NULO?	COMENTARIO
ID_QUINIELA	numérico	no nulo	clave primaria
ID_PRO	numérico	no nulo	clave primaria
ID_JORNADA	numérico	no nulo	clave foránea de la entidad EVENTOS junto con ID_EVENTO
ID_EVENTO	numérico	no nulo	clave foránea de la entidad EVENTOS junto con ID_JORNADA
PRONOSTICO	cadena(1)	no nulo	posibles valores: ('1' , 'X' , '2')

Creación de la estructura o modelo de datos

Las instrucciones DDL (Data Definition Language) lenguaje de definición de datos, permiten crear las tablas o estructuras de datos así como sus claves primarias y foráneas entre otras cosas. La sintaxis que permite crear las tablas y sus claves primarias según lo expuesto con anterioridad es la siguiente:

Tabla **EQUIPOS:**

CÓDIGO:

```
create table EQUIPOS (
  ID_EQUIPO int(10)     not null,
  EQUIPO    varchar(30) not null
);
```

Clave primaria de **EQUIPOS**

CÓDIGO:

```
alter table EQUIPOS
  add constraint primary key EQUIPOS_PK (ID_EQUIPO);
```

Tabla **EVENTOS**

CÓDIGO:
```
create table EVENTOS (
  ID_JORNADA int         not null,
  ID_EVENTO  int         not null,
  LOCAL      int         not null,
  VISITANTE  int         not null,
  RESULTADO  varchar(1) null
);
```

Clave primaria de **EVENTOS**

CÓDIGO:
```
alter table EVENTOS
add constraint primary key EVENTOS_PK (ID_JORNADA,ID_EVENTO);
```

Tabla **JORNADAS**

CÓDIGO:
```
create table JORNADAS (
  ID_JORNADA int          not null,
  NOMBRE     varchar(30) not null,
  FECHA      date         not null,
  DISPUTADA  varchar(1)  not null default 'N'
);
```

Clave primaria de **JORNADAS**

CÓDIGO:
```
alter table JORNADAS
  add constraint primary key JORNADAS_PK (ID_JORNADA);
```

Tabla **QUINIELAS**

CÓDIGO:
```
create table QUINIELAS (
  ID_QUINIELA int          not null,
  NOMBRE      varchar(30) null,
  ESCRUTADA   varchar(1)  not null default 'N',
  ACIERTOS    int          null
);
```

Clave primaria de **QUINIELAS**

CÓDIGO:
```
alter table QUINIELAS
  add constraint primary key QUINIELAS_PK (ID_QUINIELA);
```

Tabla **PRONOSTICOS**

CÓDIGO:
```
create table PRONOSTICOS (
  ID_QUINIELA int        not null,
  ID_PRO      int        not null,
  ID_JORNADA  int        not null,
  ID_EVENTO   int        not null,
  PRONOSTICO  varchar(1) not null
);
```

Clave primaria de **PRONOSTICOS**

CÓDIGO:
```
alter table PRONOSTICOS
add constraint primary key PRONOSTICOS_PK (ID_QUINIELA,ID_PRO);
```

*** * ***

Claves foráneas

A continuación se mostrarán las instrucciones que crean las claves foráneas según indica el modelo entidad-relación. Las claves foráneas es preferible crearlas una vez se han creado todas las tablas con sus claves primarias, de lo contrario puede que se esté intentando crear una clave foránea que hace referencia a una tabla que todavía no existe en la base de datos, en ese caso la instrucción fallará y devolverá un error.

Claves foráneas de **EVENTOS**

CÓDIGO:
```
alter table EVENTOS
  add constraint EVENTOS_ID_JORNADA_FK   foreign key
      (ID_JORNADA) references JORNADAS (ID_JORNADA),
  add constraint EVENTOS_ID_LOCAL_FK     foreign key
      (LOCAL) references EQUIPOS (ID_EQUIPO),
  add constraint EVENTOS_ID_VISITANTE_FK foreign key
      (VISITANTE) references EQUIPOS (ID_EQUIPO);
```

Clave foránea de **QUINIELAS**

CÓDIGO:

```
alter table QUINIELAS
  add constraint QUINIELAS_ID_JORNADA_FK foreign key
      (ID_JORNADA) references JORNADAS (ID_JORNADA);
```

Clave foránea de **PRONOSTICOS**

CÓDIGO:

```
alter table PRONOSTICOS
  add constraint PRONOSTICOS_ID_QUINIELA_FK foreign key
      (ID_QUINIELA) references QUINIELAS (ID_QUINIELA),
  add constraint PRONOSTICOS_ID_JOR_ID_EVEN_FK foreign key
      (ID_JORNADA,ID_EVENTO) references EVENTOS
      (ID_JORNADA,ID_EVENTO);
```

*** * ***

Inserción de registros en las tablas

Para alimentar la BD lo habitual es disponer de alguna pantalla, o interface de usuario, donde mediante un formulario permita realizar las inserciones. Como esto queda fuera del alcance de este curso los registros se han creado directamente con instrucciones de inserción directamente sobre la BD, aunque lo habitual hubiese sido que un usuario insertara los datos desde los formularios de entrada de datos y mantenimiento de la aplicación.

Para simplificar vamos a suponer que en la competición participan solo seis equipos y, por tanto, se celebrarán diez jornadas de Liga, cinco la primera vuelta y cinco más la segunda, con un total de tres eventos por jornada. Supondremos también que la competición se encuentra en momento tal que la jornada 8 todavía no se ha disputado, es decir, se han disputado ya 7 jornadas, por lo que en los registros de las tabla JORNADAS referentes a las jornadas 8, 9 y 10 el campo DISPUTADA contendrá una "N" y los registros de la tabla eventos referentes a las mismas jornadas el campo RESULTADO estará a nulo.

Inserts

Las inserciones que dejan la BD en esta situación son las siguientes:

Inserciones en la tabla **EQUIPOS**

CÓDIGO:
```
insert into EQUIPOS (ID_EQUIPO, EQUIPO) values (1, 'Las Palmas');
insert into EQUIPOS (ID_EQUIPO, EQUIPO) values (2, 'Xerez');
insert into EQUIPOS (ID_EQUIPO, EQUIPO) values (3, 'Getafe');
insert into EQUIPOS (ID_EQUIPO, EQUIPO) values (4, 'Nastic');
insert into EQUIPOS (ID_EQUIPO, EQUIPO) values (5, 'Celta');
insert into EQUIPOS (ID_EQUIPO, EQUIPO) values (6, 'Alcorcón');
```

Inserciones en la tabla **JORNADAS**

CÓDIGO:
```
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (1, 'J
ornada 1', '2010-01-10', 'S');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (2, 'J
ornada 2', '2010-01-17', 'S');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (3, 'J
ornada 3', '2010-01-24', 'S');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (4, 'J
ornada 4', '2010-02-07', 'S');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (5, 'J
ornada 5', '2010-02-14', 'S');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (6, 'J
ornada 6', '2010-02-21', 'S');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (7, 'J
ornada 7', '2010-03-07', 'S');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (8, 'J
ornada 8', '2010-03-21', 'N');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (9, 'J
ornada 9', '2010-04-04', 'N');
insert into JORNADAS (ID_JORNADA, NOMBRE, FECHA, DISPUTADA) values (10, '
Jornada 10', '2010-04-18', 'N');
```

Inserciones en la tabla **EVENTOS**

CÓDIGO:
```
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (1, 1, 5, 1, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (1, 2, 2, 3, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (1, 3, 4, 6, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (2, 1, 1, 2, '2');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (2, 2, 3, 4, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
```

```
values (2, 3, 6, 5, '2');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (3, 1, 1, 4, 'X');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (3, 2, 3, 5, '2');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (3, 3, 6, 2, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (4, 1, 3, 6, '2');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (4, 2, 2, 4, '2');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (4, 3, 1, 5, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (5, 1, 1, 3, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (5, 2, 5, 2, '2');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (5, 3, 6, 4, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (6, 1, 2, 1, 'X');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (6, 2, 4, 3, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (6, 3, 5, 6, '2');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (7, 1, 3, 2, 'X');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (7, 2, 4, 5, '1');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (7, 3, 6, 1, 'X');
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (8, 1, 3, 1, NULL);
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (8, 2, 2, 6, NULL);
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (8, 3, 5, 4, NULL);
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (9, 1, 1, 6, NULL);
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (9, 2, 5, 3, NULL);
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (9, 3, 4, 2, NULL);
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (10, 1, 6, 3, NULL);
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (10, 2, 4, 1, NULL);
insert into EVENTOS (ID_JORNADA, ID_EVENTO, LOCAL, VISITANTE, RESULTADO)
values (10, 3, 2, 5, NULL);
```

Inserciones en la tabla **QUINIELAS**

CÓDIGO:

```
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (1, 1, 'Quini 1.1', 'S', 0);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (2, 1, 'Quini 1.2', 'S', 1);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (3, 1, 'Quini 1.3', 'S', 0);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (4, 2, 'Quini 2.1', 'S', 1);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (5, 2, 'Quini 2.2', 'S', 1);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (6, 3, 'Quini 3.1', 'S', 1);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (7, 3, 'Quini 3.2', 'S', 1);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (8, 3, 'Quini 3.3', 'S', 3);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (9, 3, 'Quini 3.4', 'S', 2);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (10, 4, 'Quini 4.1', 'S', 2);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (11, 4, 'Quini 4.2', 'S', 0);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (12, 4, 'Quini 4.3', 'S', 0);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (13, 5, 'Quini 5.1', 'S', 2);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (14, 5, 'Quini 5.2', 'S', 2);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (15, 6, 'Quini 6.1', 'S', 0);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (16, 6, 'Quini 6.2', 'S', 1);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (17, 6, 'Quini 6.3', 'S', 0);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (18, 7, 'Quini 7.1', 'S', 1);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (19, 7, 'Quini 7.2', 'S', 1);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (20, 8, 'Quini 8.1', 'N', NULL);
insert into QUINIELAS (ID_QUINIELA, ID_JORNADA, NOMBRE, ESCRUTADA,
ACIERTOS) values (21, 8, 'Quini 8.2', 'N', NULL);
```

Inserciones en la tabla **PRONOSTICOS**

CÓDIGO:

```
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (1, 1, 1, 1, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (1, 2, 1, 2, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (1, 3, 1, 3, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (2, 1, 1, 1, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (2, 2, 1, 2, '1');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (2, 3, 1, 3, '2');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (3, 1, 1, 1, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (3, 2, 1, 2, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (3, 3, 1, 3, '2');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (4, 1, 2, 1, '1');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (4, 2, 2, 2, '2');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (4, 3, 2, 3, '2');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (5, 1, 2, 1, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (5, 2, 2, 2, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (5, 3, 2, 3, '2');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (6, 1, 3, 1, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (6, 2, 3, 2, '1');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (6, 3, 3, 3, 'X');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (7, 1, 3, 1, '2');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (7, 2, 3, 2, '2');
insert into PRONOSTICOS (ID_QUINIELA, ID_PRO, ID_JORNADA, ID_EVENTO, PRON
OSTICO) values (7, 3, 3, 3, '2');
...
```

*** * ***

Informes o explotación de datos

Al igual que para las inserciones, lo más normal de una aplicación es que disponga de alguna funcionalidad en las pantallas de gestión que permita obtener informes. Esto también está fuera del ámbito de este curso, así que lo haremos con consultas SQL directamente sobre la BD.

Usted debería estar capacitado para desarrollar esta parte de la aplicación, puesto que es lo que se ha estado trabajando durante todo el curso. No voy a quitarle el protagonismo que merece y dejaré que ponga en práctica lo aprendido. Los informes o consultas a desarrollar las encontrará a continuación, en el apartado de ejercicios. En esta lección, por ser la última, no se han publicado las soluciones, aunque sí los resultados de las consultas que se piden en los ejercicios para que puedan ser contrastados con los resultados que usted obtenga.

Sin embargo hay una cuestión que por la naturaleza de esta aplicación de ejemplo es preferible introducir. En los ejercicios se le pedirá un tipo de consultas que requiere que aparezca dos veces la tabla EQUIPOS en la cláusula FROM. Esto es debido a que en todo evento participan dos equipos, el local y el visitante. El modo de poder usar la misma tabla por duplicado en la cláusula FROM es mediante alias de tabla, con ello se romper la ambigüedad del mismo modo que se hace con los campos de igual nombre pero de tablas distintas. Veamos esto con un ejemplo:

Que calendario tiene en la competición el Xerez, equipo de identificador 2:

CÓDIGO:

```
  select J.ID_JORNADA,
         date_format(J.FECHA,'%d-%m-%Y') FECHA,
         L.EQUIPO LOCAL,
         V.EQUIPO  VISITANTE
    from JORNADAS J, EVENTOS E, EQUIPOS L, EQUIPOS V
   where J.ID_JORNADA  = E.ID_JORNADA
     and E.LOCAL       = L.ID_EQUIPO
     and E.VISITANTE   = V.ID_EQUIPO
     and (E.LOCAL = 2 or E.VISITANTE = 2)
   order by J.FECHA
```

ID_JORNADA	FECHA	LOCAL	VISITANTE
1	10-01-2010	Xerez	Getafe
2	17-01-2010	Las Palmas	Xerez
3	24-01-2010	Alcorcón	Xerez
4	07-02-2010	Xerez	Nastic
5	14-02-2010	Celta	Xerez
6	21-02-2010	Xerez	Las Palmas
7	07-03-2010	Getafe	Xerez
8	21-03-2010	Xerez	Alcorcón
9	04-04-2010	Nastic	Xerez
10	18-04-2010	Xerez	Celta

*** * ***

Ejercicio 1

Desarrolle un informe que muestre los pronósticos de una quiniela, pruebe su funcionamiento con la quiniela de identificador 4.

Resultado a obtener:

ID_QUINIELA	ID_EVENTO	LOCAL	VISITANTE	PRONOSTICO
4	1	Las Palmas	Xerez	1
4	2	Getafe	Nastic	2
4	3	Alcorcón	Celta	2

Ejercicio 2

Desarrolle un informe que muestre la combinación ganadora de una jornada, pruebe su funcionamiento con la jornada de identificador 3.

Resultado a obtener:

ID_JORNADA	ID_EVENTO	LOCAL	VISITANTE	RESULTADO
3	1	Las Palmas	Nastic	X
3	2	Getafe	Celta	2
3	3	Alcorcón	Xerez	1

Ejercicio 3.1

Desarrolle un informe que escrute una quiniela, es decir, que muestre los eventos en los que se acertó el resultado. Pruebe el funcionamiento con la quiniela de identificador 6.
Nota: Deberá usar la función IF para calcular la columna ACIERTO.

Resultado a obtener:

ID_QUINIELA	ID_JORNADA	ID_EVENTO	LOCAL	VISITANTE	RESULTADO	PRONOSTICO	ACIERTO

ID_QUINIELA	ID_JORNADA	ID_EVENTO	LOCAL	VISITANTE	RESULTADO	PRONOSTICO	ACIERTO
6	3	1	Las Palmas	Nastic	X	X	Sí
6	3	2	Getafe	Celta	2	1	No
6	3	3	Alcorcón	Xerez	1	X	No

Ejercicio 3.2

Tomando como patrón la consulta resultante del ejercicio 3.1, desarrolle una consulta que calcule los aciertos de las quinielas, es decir, escrute las quinielas. Agrupe los datos por quiniela. Si una quiniela no tiene ningún acierto no es necesario que aparezca en la lista resultante.

Resultado a obtener:

ID_QUINIELA	ACIERTOS
2	1
4	1
5	1
6	1
7	1
8	3
9	2
10	2
13	2
14	2
16	1
18	1
19	1

Añádale un filtro para poder calcular los aciertos de una quiniela concreta. Este dato es especialmente útil para que un usuario, o un proceso automático, pueda actualizar el campo ACIERTOS de la tabla QUINIELAS, que contiene un valor nulo hasta que se conozca la combinación ganadora y, en consecuencia, el dato a actualizar una vez escrutada la quiniela.

Ejercicio 3.3

Desarrolle una consulta que calcule los aciertos de las quinielas pero esta vez considerando las quinielas que no presentan ningún acierto.

Resultado a obtener:

ID_QUINIELA	ACIERTOS
1	0
2	1
3	0
4	1
5	1
6	1
7	1
8	3
9	2
10	2
11	0
12	0
13	2
14	2
15	0
16	1
17	0
18	1
19	1
20	0
21	0

Ejercicio 4

Desarrolle un informe que muestre la media de aciertos agrupado por jornada. No considere quinielas de jornadas no disputadas. No es necesario recalcular los aciertos de las quinielas, en su lugar use el campo ACIERTOS de la tabla QUINIELAS.

Resultado a obtener:

ID_JORNADA	QUINIELAS	MEDIA_ACIERTOS
1	3	0.33
2	2	1.00
3	4	1.75
4	3	0.67
5	2	2.00
6	3	0.33
7	2	1.00

Ejercicio 5

Desarrolle un informe que muestre la media de aciertos agrupado por meses. No considere quinielas de jornadas no disputadas, ni recalcule los aciertos de las quinielas.

Nota: deberá utilizar la siguiente función de MySQL para poder agrupar por mes-año

DATE_FORMAT(FECHA_A_FORMATEAR,'%m-%Y')

Resultado a obtener:

MES	QUINIELAS	MEDIA_ACIERTOS
01-2010	9	1.11
02-2010	8	0.88
03-2010	2	1.00

Soluciones

Lección 2 - Consultas I (SQL SELECT FROM WHERE)

Ejercicio 1

Intente hallar una consulta que devuelva el nombre, apellidos y la fecha de nacimiento de aquellos empleados que cobren más de 1350 euros.

CÓDIGO:
```
select NOMBRE , APELLIDOS , F_NACIMIENTO
  from EMPLEADOS
 where SALARIO > 1350
```

NOMBRE	APELLIDOS	F_NACIMIENTO
Carlos	Jiménez Clarín	1985-05-03
José	Calvo Sisman	1990-11-12

* * *

Lección 3 - Consultas II (SQL SELECT FROM WHERE)

Ejercicio 1

Intente hallar una consulta que devuelva el nombre y apellidos de los empleados que cobren menos de 1350 euros.

CÓDIGO:
```
select NOMBRE , APELLIDOS
  from EMPLEADOS
 where 1350 > SALARIO
```

O bien:

CÓDIGO:
```
select NOMBRE , APELLIDOS
  from EMPLEADOS
 where SALARIO < 1350
```

Y el resultado que nos devuelve el SGBD es:

NOMBRE	APELLIDOS
Elena	Rubio Cuestas
Margarita	Rodríguez Garcés

* * *

Lección 4 - Tipos de datos

Ejercicio 1

Defina de que tipo de dato crearía los campos, y su tamaño mínimo si se tercia, para albergar los siguientes datos:

- Hola mundo -> VARCHAR(10)
- 9.36 -> FLOAT
- 4564 -> INT
- Esto es un ejercicio de tipos de datos -> VARCHAR(38)
- 8 de enero de 1998 -> DATE

Ejercicio 2

Formatee en una cadena, según se ha visto en esta lección, las siguientes fechas.

- 23 de agosto de 1789 -> '17890823'
- 8 de enero de 1998 -> '19980108'

* * *

Lección 5 - Operadores (SQL WHERE)

Ejercicio 1

Cree una consulta SQL que devuelva las personas que son altas, o bien son rubias con gafas.

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where ALTA = 'S' or (RUBIA = 'S' and GAFAS = 'S')
```

NOMBRE
Manuel
Carmen
José
Pedro

Ejercicio 2

Cree una consulta SQL que devuelva los empleados que son mujer y cobran más de 1300 euros.

CÓDIGO:
```
select NOMBRE,APELLIDOS
  from EMPLEADOS
 where SEXO = 'M'
   and SALARIO > 1300
```

NOMBRE	APELLIDOS
Margarita	Ródriguez Garcés

Ejercicio 3

Usando solo expresiones (ALTA = 'S') , (RUBIA = 'S') , (GAFAS = 'S') combinadas con el operador NOT resuelva: ¿Quién es quién? Lleva gafas y no es alta ni rubia.

CÓDIGO:
```
select NOMBRE
  from PERSONAS
 where GAFAS = 'S'
   and not RUBIA = 'S'
   and not ALTA = 'S'
```

NOMBRE
Maria

Ejercicio 4

Suponiendo que A vale cierto y B vale falso, evalúe la siguiente expresión booleana:

C= ((A and B) and (A or (A or B))) or A

A = cierto
B = falso
C= ((A and B) and (A or (A or B))) or A

En este caso no hace falta evaluar toda la expresión, sabemos que lo que está entre paréntesis dará cierto o falso, de momento digámosle a este resultado X, luego:

C= X or A

sabemos que A es cierto, evaluemos la expresión para los dos posibles valores de X:

X = cierto -> C = cierto or cierto = cierto
X = falso -> C = falso or cierto = cierto

por lo tanto, valga lo que valga X el resultado es cierto.

C= X or A = X or cierto = **cierto**

*** * ***

Lección 6 - Totalizar datos / Alias de campos (SQL AS)

Ejercicio 1

En todos los ejemplos de esta lección se ha omitido la clausula WHERE, construya una consulta, donde necesitará establecer una condición en la clausula WHERE, que devuelva el salario medio de los empleados que son hombres. Renombre la cabecera del resultado con un título que deje claro que dato se está mostrando.

CÓDIGO:
```
select avg(SALARIO) as MEDIA_SALARIO_HOMBRES
  from EMPLEADOS
 where SEXO = 'H'
```

MEDIA_SALARIO_HOMBRES
1450

Ejercicio 2

Construya una consulta que devuelva en la misma fila el salario máximo y mínimo de entre todos los empleados. Renombre las cabeceras de resultados con un título que deje claro que datos se están mostrando.

CÓDIGO:
```
select min(SALARIO) as SALARIO_MINIMO,
       max(SALARIO) as SALARIO_MAXIMO
  from EMPLEADOS
```

SALARIO_MINIMO	SALARIO_MAXIMO
1300	1500

Ejercicio 3

Construya una consulta que responda a lo siguiente: ¿Que cuesta pagar a todas las mujeres en total? Renombre la cabecera del resultado con un título que deje claro que dato se está mostrando.

CÓDIGO:
```
select sum(SALARIO) as TOTAL_SALARIO_MUJERES
  from EMPLEADOS
 where SEXO = 'M'
```

TOTAL_SALARIO_MUJERES
2625.5

*** * ***

Lección 7 - Agrupación de datos (SQL GROUP BY)

Ejercicio 1

Construya una consulta que devuelva el salario medio, máximo y mínimo de los empleados agrupado por sexo.

CÓDIGO:
```
select SEXO,
       avg(SALARIO) as SALARIO_MEDIO ,
       min(SALARIO) as SALARIO_MINIMO ,
       max(SALARIO) as SALARIO_MAXIMO
  from EMPLEADOS
 group by SEXO
```

SEXO	SALARIO_MEDIO	SALARIO_MINIMO	SALARIO_MAXIMO
H	1450	1400	1500
M	1312.75	1300	1325.5

Ejercicio 2

Construya una consulta que devuelva cuantos perros y cuantos gatos han pasado por el centro y ya no están.

CÓDIGO:
```
select ESPECIE , count(*) as BAJAS
  from MASCOTAS
 where ESTADO = 'B'
 group by ESPECIE
```

ESPECIE	BAJAS
G	2
P	2

Ejercicio 3

Construya una consulta que devuelva cuantos perros macho hay actualmente en el centro agrupado por ubicación.

CÓDIGO:

```
select UBICACION , count(*) as PERROS_MACHO
  from MASCOTAS
 where ESTADO = 'A'
   and ESPECIE = 'P'
   and SEXO = 'M'
 group by UBICACION
```

UBICACION	PERROS_MACHO
E02	1
E03	1

Ejercicio 4

Con ayuda del filtro DISTINCT, construya una consulta que devuelva las diferentes especies que hay actualmente en cada jaula o ubicación del centro.

CÓDIGO:

```
select distinct UBICACION , ESPECIE
  from MASCOTAS
 where ESTADO = 'A'
```

UBICACION	ESPECIE
E02	P
E03	P
E04	G
E05	P
E01	G

*** * ***

Lección 8 - Filtrar cálculos de totalización (SQL HAVING)

Ejercicio 1

Usando el operador BETWEEN que vimos en las lecciones 3 y 5, construye una consulta que devuelva las ubicaciones del centro de mascotas que tiene entre 2 y 3 ejemplares.

CÓDIGO:
```
select UBICACION , count(*) as EJEMPLARES
  from MASCOTAS
 where ESTADO = 'A'
 group by UBICACION
having count(*) between 2 and 3
```

Resultado:

UBICACION	EJEMPLARES
E01	2
E04	3
E05	2

*** * ***

Lección 9 - Ordenación del resultado (SQL ORDER BY)

Ejercicio 1

Obtenga una lista de las personas de la tabla PERSONAS, donde primero aparezcan las rubias, después las altas, y finalmente las que llevan gafas. De manera que la primera persona de la lista, si la hay, será rubia alta y sin gafas, y la última, si la hay, no será rubia ni alta y llevará gafas.

CÓDIGO:
```
select *
  from PERSONAS
 order by RUBIA desc, ALTA desc, GAFAS
```

ID_PERSONA	NOMBRE	RUBIA	ALTA	GAFAS
1	Manuel	S	S	N
4	José	S	S	S
3	Carmen	S	N	S
5	Pedro	N	S	N
2	Maria	N	N	S

Ejercicio 2

Obtenga el número actual de ejemplares de cada ubicación del centro de mascotas, que tengan dos o más ejemplares ordenado de mayor a menor por número de ejemplares y en segundo término por ubicación.

CÓDIGO:

```
select UBICACION, count(*) as EJEMPLARES
  from MASCOTAS
 where ESTADO = 'A'
 group by UBICACION
having count(*) > 1
 order by count(*) desc, UBICACION
```

UBICACION	EJEMPLARES
E02	4
E04	3
E01	2
E05	2

*** * ***

Lección 10 - El operador LIKE / El valor NULL

Ejercicio 1

¿Qué empleados se apellidan Calvo?

CÓDIGO:

```
select *
  from EMPLEADOS
 where APELLIDOS like '%calvo%'
```

ID_EMPLEADO	NOMBRE	APELLIDOS	F_NACIMIENTO	SEXO	CARGO	SALARIO
3	José	Calvo Sisman	1990-11-12	H	Mozo	1400

Ejercicio 2

Considerando que en la tabla VEHICULOS el campo PROX_ITV guarda la fecha de la próxima ITV que ha de pasar cada vehículo: ¿Qué vehículos que nunca han pasado la ITV deben pasar la primera revisión durante el año 2011?

CÓDIGO:

```
select *
  from VEHICULOS
 where PROX_ITV between '20110101' and '20111231'
   and ULTI_ITV is null
```

ID_VEHICULO	MARCA	MODELO	PROX_ITV	ULTI_ITV
1	Alfa Romeo	Brera	2011-10-20	
5	Ford	Fiesta	2011-04-22	

*** * ***

Lección 11 - Síntesis de la primera parte

Ejercicio 1

Supongamos que usted tiene un amigo que es jugador de póquer, el pobre no sabe si sus ganancias en el juego son positivas o negativas porque no lleva un control sobre ello, por lo que usted se ofrece a gestionarle las ganancias. Le dice a su amigo que cuando acabe una sesión de juego le comunique a usted el dinero que ha ganado o perdido, entendiendo perdida como una ganancia o número en negativo.

Diseñe una tabla, es decir, los campos y tipo de dato de cada campo, para poder registrar la información que su amigo le facilita, y mediante SQL pueda responder en cualquier momento a las siguientes preguntas:

¿Cuáles son las ganancias actuales?

¿Cuánto dinero se ganó durante el mes de marzo de 2009? Una vez diseñada la tabla construya las consultas SQL que responden a cada una de estas preguntas.

Una solución posible es:

Tabla POKER:

ID_SESION ->INT
F_SESION -> DATE
GANANCIA -> FLOAT

¿Cuales son las ganancias actuales?

CÓDIGO:

```
select sum(GANANCIA)
  from POKER
```

¿Cuánto dinero se ganó durante el mes de marzo de 2009?

CÓDIGO:
```
select sum(GANANCIA)
  from POKER
 where F_SESION between '20090301' and '20090331'
```

***

Lección 12 - El producto cartesiano (SQL FROM)

Ejercicio 1

Realice una consulta que devuelva las combinaciones posibles entre los pantalones y los calzados, sin más columnas que la descripción de cada prenda. Use alias de tabla para indicar a que tabla pertenece cada campo de la cláusula SELECT.

CÓDIGO:
```
select P.PANTALON , Z.CALZADO
  from PANTALONES P , CALZADOS Z
```

PANTALON	CALZADO
tela azul marino	deportivas
pana marron claro	deportivas
tela azul marino	mocasines
pana marron claro	mocasines
tela azul marino	botas
pana marron claro	botas

Ejercicio 2

Si en una BD existe una tabla T1 con 4 campos y 12 registros, y una tabla T2 con 7 campos y 10 registros, ¿cuántas filas y columnas devolvería la siguiente consulta?

CÓDIGO:
```
select *
  from T1 , T2
```

filas = 12 x 10 = 120
columnas = 4 + 7 = 11

* * *

Lección 13 - Consultas III (SQL SELECT FROM WHERE)

Ejercicio 1

Construya una consulta SQL que devuelva el peso medio de todas las mudas confeccionables entre camisas y pantalones. Modifique la consulta para obtener el mismo resultado entre camisas, pantalones y calzados.

Peso medio de todas las mudas confeccionables entre camisas y pantalones:

CÓDIGO:
```
select avg( C.PESO_GR + P.PESO_GR) PESO_MEDIO_MUDAS
  from CAMISAS C , PANTALONES P
```

PESO_MEDIO_MUDAS
853.3333

Peso medio de todas las mudas confeccionables entre camisas, pantalones y calzados.

CÓDIGO:
```
select avg( C.PESO_GR + P.PESO_GR + Z.PESO_GR) PESO_MEDIO_MUDAS
  from CAMISAS C , PANTALONES P , CALZADOS Z
```

PESO_MEDIO_MUDAS
1695.0000

Ejercicio 2

Construya una consulta SQL que devuelva el peso medio de todas las mudas confeccionables entre camisas y pantalones agrupado por camisa. Modifique la consulta de manera que devuelva el mismo resultado pero de los grupos que el peso medio es superior a 850 gramos.

Peso medio de todas las mudas confeccionables entre camisas y pantalones agrupado por camisa:

CÓDIGO:

```
  select C.CAMISA , avg( C.PESO_GR + P.PESO_GR) PESO_MEDIO_MUDAS
    from CAMISAS C , PANTALONES P
group by C.CAMISA
```

CAMISA	PESO_MEDIO_MUDAS
algodon naranja	890.0000
lino blanca	810.0000
seda negra	860.0000

Peso medio de todas las mudas confeccionables entre camisas y pantalones agrupado por camisa con peso superior a 850 gramos:

CÓDIGO:

```
  select C.CAMISA , avg( C.PESO_GR + P.PESO_GR) PESO_MEDIO_MUDAS
    from CAMISAS C , PANTALONES P
group by C.CAMISA
having avg( C.PESO_GR + P.PESO_GR) > 850
```

CAMISA	PESO_MEDIO_MUDAS
algodon naranja	890.0000
seda negra	860.0000

Ejercicio 3

Construya una consulta SQL que devuelva las combinaciones de las camisas con los pantalones de manera que: la primera camisa se combine con todos los pantalones menos con el primero, la segunda camisa se combine con todos los pantalones menos con el segundo, y así sucesivamente.

CÓDIGO:

```
select *
  from CAMISAS C , PANTALONES P
 where C.ID_CAMISA != P.ID_PANTALON
```

ID_CAMISA	CAMISA	PESO_GR	ID_PANTALON	PANTALON	PESO_GR
1	lino blanca	210	2	pana marron claro	730
2	algodon naranja	290	1	tela azul marino	470
3	seda negra	260	1	tela azul marino	470
3	seda negra	260	2	pana marron claro	730

Ejercicio 4

Construye una consulta que devuelva la lista de prendas de una maleta que contiene todas las camisas, pantalones y calzados.

CÓDIGO:

```
   select concat('Camisa de ',CAMISA) as PRENDA
     from CAMISAS
union all
   select concat('Pantalón de ',PANTALON)
     from PANTALONES
union all
   select concat('Calzado: ',CALZADO)
     from CALZADOS
```

PRENDA
Camisa de lino blanca
Camisa de algodon naranja
Camisa de seda negra
Pantalón de tela azul marino
Pantalón de pana marron claro
Calzado: deportivas
Calzado: mocasines
Calzado: botas

*** * ***

Lección 14 - Relaciones, claves primarias y foráneas

Ejercicio 1

Construya una consulta que devuelva los cursos en que se ha matriculado el alumno con identificador 1. Modifique la anterior consulta para que devuelva los nombres y apellidos de los alumnos, y los cursos en que se han matriculado, tales que el nombre de pila del alumno contenga un E.

Cursos en que se ha matriculado el alumno con identificador 1:

CÓDIGO:
```
select C.TITULO CURSO
  from ALUMNOS_CURSOS AC, CURSOS C
 where AC.ID_CURSO = C.ID_CURSO
   and AC.ID_ALUMNO = 1
```

CURSO
Programación PHP
SQL desde cero

Nombres y apellidos de los alumnos, y los cursos en que se han matriculado, que el nombre de pila del alumno contenga un E:

CÓDIGO:
```
select A.NOMBRE,A.APELLIDOS,C.TITULO CURSO
  from ALUMNOS_CURSOS AC, CURSOS C, ALUMNOS A
 where AC.ID_CURSO = C.ID_CURSO
   and AC.ID_ALUMNO = A.ID_ALUMNO
   and A.NOMBRE like '%E%'
```

NOMBRE	APELLIDOS	CURSO
Teresa	Lomas Trillo	Programación PHP
Sergio	Ot Dirmet	Programación PHP
Sergio	Ot Dirmet	SQL desde cero
Jeremias	Santo Lote	Dibujo técnico
Carmen	Dilma Perna	Dibujo técnico

Ejercicio 2

¿Cuántos cursos imparte cada profesor? Construya una consulta que responda a esta cuestión de modo que el resultado muestre el nombre completo del profesor acompañado del número de cursos que imparte.

Número de cursos que imparte cada profesor:

CÓDIGO:
```
select P.NOMBRE, P.APELLIDOS , count(1) CURSOS
  from PROFESORES P, CURSOS C
 where P.ID_PROFE = C.ID_PROFE
 group by P.NOMBRE, P.APELLIDOS
```

NOMBRE	APELLIDOS	CURSOS
Ana	Saura Trenzo	1
Federico	Gasco Daza	1
Rosa	Honrosa Pérez	2

Ejercicio 3

¿Cuántos alumnos hay matriculados en cada uno de los cursos? Construya una consulta que responda a esta cuestión de modo que el resultado muestre el titulo del curso acompañado del número de alumnos matriculados. Modifique la anterior consulta de modo que muestre aquellos cursos que el número de alumnos matriculados sea exactamente de dos alumnos.

Número de alumnos matriculados en cada uno de los cursos:

CÓDIGO:
```
select C.TITULO CURSO, count(1) ALUMNOS
  from ALUMNOS_CURSOS AC, CURSOS C
 where AC.ID_CURSO = C.ID_CURSO
 group by C.TITULO
```

CURSO	ALUMNOS
Dibujo técnico	2
Modelos abstracto de datos	1
Programación PHP	3
SQL desde cero	2

Cursos en los que hay matriculados exactamente dos alumnos:

CÓDIGO:
```
select C.TITULO CURSO
  from ALUMNOS_CURSOS AC, CURSOS C
 where AC.ID_CURSO = C.ID_CURSO
 group by C.TITULO
having count(1) = 2
```

CURSO
Dibujo técnico
SQL desde cero

Ejercicio 4

Si ahora a usted le pidiesen que adaptara la BD, que consta de las tres tablas presentadas en esta lección, a la siguiente necesidad: A todo alumno se le asignara un profesor que lo tutele. ¿Qué cambios realizaría en la BD?

La solución pasa por añadir un campo en la tabla ALUMNOS que apunte a la tabla PROFESORES, en este campo se guardará el identificador del profesor que tutela al alumno. Por tanto añadiremos un campo en la tabla ALUMNOS llamado por ejemplo ID_TUTOR, que será una clave foránea de la tabla PROFESORES.

*** * ***

Lección 15 - Reunión interna y externa

Ejercicio 1

Construya una consulta que resuelva el número de cursos que imparte cada profesor usando la cláusula INNER JOIN.

CÓDIGO:

```
  select P.NOMBRE, P.APELLIDOS , count(1) CURSOS
    from PROFESORES P inner join CURSOS C
      on P.ID_PROFE = C.ID_PROFE
group by P.NOMBRE, P.APELLIDOS
```

NOMBRE	APELLIDOS	CURSOS
Ana	Saura Trenzo	1
Federico	Gasco Daza	1
Rosa	Honrosa Pérez	2

Ejercicio 2

Realice una consulta entre las tablas CURSOS, ALUMNOS y ALUMNOS_CURSOS de modo que aparezcan los alumnos matriculados en cada curso pero mostrando todos los cursos aunque no tengan alumnos matriculados.

CÓDIGO:

```
select C.TITULO CURSO, A.NOMBRE,A.APELLIDOS
  from (ALUMNOS_CURSOS AC inner join ALUMNOS A
    on AC.ID_ALUMNO = A.ID_ALUMNO) right join CURSOS C
    on AC.ID_CURSO = C.ID_CURSO
```

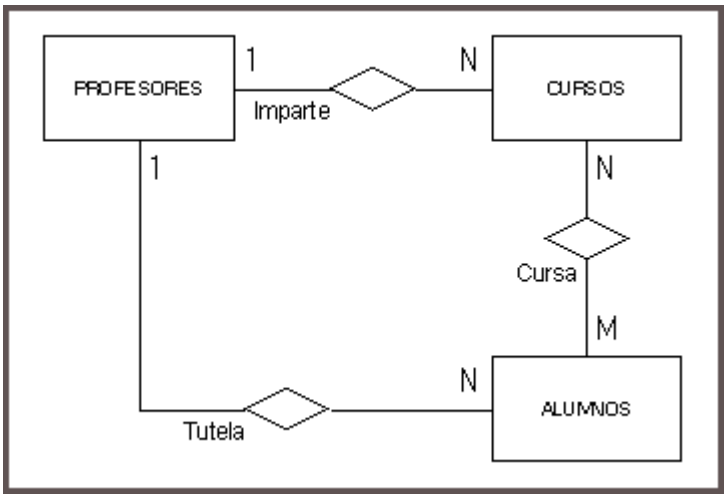
CURSO	NOMBRE	APELLIDOS
Programación PHP	Pablo	Hernandaz Mata
Programación PHP	Teresa	Lomas Trillo
Programación PHP	Sergio	Ot Dirmet
Modelos abstracto de datos	Marta	Fuego García
SQL desde cero	Pablo	Hernandaz Mata
SQL desde cero	Sergio	Ot Dirmet
Dibujo técnico	Jeremias	Santo Lote
Dibujo técnico	Carmen	Dilma Perna
SQL avanzado		

*** * ***

Lección 16 - El modelo entidad-relación

Ejercicio 1

Modifique el modelo entidad-relación presentado en esta lección para que considere la siguiente premisa: Todo alumno tendrá un profesor que lo tutele.



La premisa implica establecer una nueva relación entre ALUMNOS y PROFESORES de modo que un profesor puede tutelar a más de un alumno y un alumno es tutelado por un profesor. Por lo que en la BD se deberá montar una clave foránea de la tabla PROFESORES en la tabla ALUMNOS.

*** * ***

Lección 17 - Funciones

Ejercicio 1

Realice una consulta que devuelva la media de salarios de la tabla EMPLEADOS agrupado por sexo. Redondee la media de salarios a un solo decimal y decodifique la columna sexo para que aparezca el literal HOMBRES y MUJERES en lugar de H y M. No olvide rebautizar las columnas con un alias apropiado.

CÓDIGO:

```
select if(SEXO='M','MUJERES','HOMBRES') SEXO ,
       round(avg(SALARIO),1) MEDIA
  from EMPLEADOS
 group by if(SEXO='M','MUJERES','HOMBRES')
```

SEXO	MEDIA
HOMBRES	1450.0
MUJERES	1312.8

Ejercicio 2

Realice una consulta sobre la tabla EMPLEADOS que devuelva el nombre, los apellidos, la fecha de nacimiento y la edad actual en años de cada empleado. Para aquellos empleados con 18 años o más. Nota: la edad de un empleado en años es el número de días transcurridos desde el nacimiento dividido entre los 365 días que tiene un año.

CÓDIGO:

```
select NOMBRE,
       APELLIDOS,
       F_NACIMIENTO,
       truncate( datediff(current_date,f_nacimiento)
                 / 365 , 0) EDAD
  from EMPLEADOS
 where truncate( datediff(current_date,f_nacimiento)
                 / 365 , 0) >= 18
```

NOMBRE	APELLIDOS	F_NACIMIENTO	EDAD
Carlos	Jiménez Clarín	1985-05-03	26
Elena	Rubio Cuestas	1978-09-25	33
José	Calvo Sisman	1990-11-12	21
Margarita	Ródriguez Garcés	1992-05-16	19

Ejercicio 3

Realice una consulta sobre la tabla vehículos que devuelva el número de vehículos que deben pasar la revisión agrupado por el año en que deben pasarla.

CÓDIGO:
```
select date_format(PROX_ITV,'%Y') AÑO_ITV, count(1) VEHICULOS
  from VEHICULOS
 group by date_format(PROX_ITV,'%Y')
```

AÑO_ITV	VEHICULOS
2009	1
2010	2
2011	2

* * *

Lección 18 - INSERT, UPDATE, DELETE SQL

Ejercicio 1

Construya una instrucción de inserción en la tabla CURSOS para guardar un nuevo curso de pintura y asígnele una clave que no entre en conflicto con la existentes, posteriormente construya la instrucción para eliminar de la tabla el registro que acaba de crear.

Insert:

CÓDIGO:
```
insert into CURSOS(ID_CURSO,TITULO)
values (6,'Pintura')
```

Delete:

CÓDIGO:
```
delete from CURSOS
 where ID_CURSO = 6
```

Ejercicio 2

En esta lección se puso como ejemplo la actualización del salario de los empleados donde este se incrementaba un 2% para empleados con un sueldo inferior a 3000 euros. Sin embargo no parece muy justo que un empleado con un sueldo de 3000 Euros no reciba incremento alguno, y otros que rozan los 3000 euros pero no llegan reciban el incremento superando el importe de corte una vez aplicado dicho incremento. Construya una instrucción de actualización, que se debería ejecutar previamente, de modo que evite que para estos empleados el resultado del incremento sea superior a 3000 euros. Para ello

esta instrucción debe actualizar el salario de los empleados afectados a 3000 euros, para que cuando se realice el incremento no se les aplique la subida puesto que su sueldo será entones de 3000 euros justos.

CÓDIGO:

```
update EMPLEADOS
   set SALARIO = 3000
 where SALARIO < 3000
   and SALARIO * 1.02 > 3000
```

*** * ***

Lección 19 - Síntesis de la segunda parte

Ejercicio 1

Supongamos que tenemos las siguientes entidades en un modelo relacional que gestiona la liga profesional de fútbol: EQUIPOS y JUGADORES. La cardinalidad de esta relación es 1 a N, puesto que un equipo tiene una plantilla de N jugadores mientras que un jugador milita en un solo equipo. ¿Es JUGADORES una entidad débil?

JUGADORES es claramente una entidad fuerte, entre otras cosas porque un jugador no está sujeto a un equipo eternamente, sino que durante una campaña milita en un equipo pero la siguiente, o incluso antes, puede cambiar de club.

Ejercicio 2

Supongamos que tenemos las siguientes entidades en un modelo relacional que gestiona las reparaciones del alumbrado público de una urbanización: FAROLAS y REPARACIONES. La cardinalidad de esta relación es 1 a N, puesto que a una farola se le realizan N reparaciones mientras que una reparación se practica a una farola. ¿Es REPARACIONES una entidad débil?

REPARACIONES es una entidad débil, quizás se debería llamar REPARACIONES_FAROLA. Por farola se espera un número de reparaciones relativamente pequeño, hasta que la farola sea desechada y cambiada por otra. Es claramente dependiente de la entidad fuerte FAROLAS y los registros están claramente asociados a una y solo una farola. La entidad REPARACIONES no podría existir, en este contexto claro está, sin la existencia de la entidad FAROLAS.

Ejercicio 3

Supongamos que tenemos las siguientes entidades en un modelo relacional que gestiona la actividad de un almacén de distribución de género: ARTICULOS y FAMILIAS. La cardinalidad de esta relación es 1 a N, puesto que una familia agrupa N artículos mientras que un artículo pertenece a una sola familia. ¿Es ARTICULOS una entidad débil?

ARTCULOS es una entidad fuerte, puesto que la familia es solo un modo de agrupación de artículos y en ocasiones puede ser dudoso a que familia asociar un artículo. Cabe considerar además que ARTICULOS se relacione con otras muchas entidades tales como PEDIDOS, FACTURAS, VENTAS, etc... no parece práctico acarrear una clave compuesta al crear las claves foráneas en todas estas entidades. Por lo tanto ARTICULOS debe identificar sus registros con una clave propia y FAMILIAS es solo un modo de agrupar artículos.