__Víctor__ ha aprendido mucho sobre Sistemas Gestores de Bases de Datos. Ayuda a __María__ a hacer los diagramas Entidad-Relación para las aplicaciones que desarrollan con bases de datos. No tiene ningún problema pasando esos diagramas al __modelo relacional__ basado en tablas, y sabe normalizar las bases de datos resultantes, en base a las dependencias existentes en los datos, para mejorar su diseño evitando al máximo las redundancias de información.

Todo esto, aunque le ha costado bastante esfuerzo y tiempo de estudio, le ha resultado muy interesante, ya que ha ampliado bastante la limitada visión de lo que es una __base de datos__ que tenía tras la breve introducción que tuvo a este tema mientras estudiaba el lenguaje de programación.

No obstante, justamente por aquella visión parcial, sabe que hasta ahora todo lo que ha hecho es __diseñar__ la base de datos, pero que aún no ha abordado la parte práctica del asunto: __Construir y explotar la base de datos__.

Sabe que para hacerlo es necesario usar un nuevo lenguaje, __SQL__, con el que ya ha trabajado, pero apenas conoce las sentencias más elementales de este lenguaje, aquéllas que le permitían acceder a los datos desde las aplicaciones Java que ha hecho, haciendo consultas muy básicas y elementales. Eso sí, sabe que es un __lenguaje estándar de consulta__, que resulta imprescindible para realizar operaciones sobre las bases de datos. Por lo que ha buscado en Internet, sabe que SQL es algo mucho más potente, y también mucho más extenso.

Está impaciente por comenzar a aprender este __lenguaje__, porque sabe que su manejo le va a permitir dar otro gran salto en su formación como programador, permitiéndole administrar y utilizar bases de datos más eficientemente, y sacarle todo el partido posible a la información que contienen.

Y como si le estuviera leyendo el pensamiento, __José__ lo llama a su despacho junto a __Carmen__ para decirle que ésta se va a encargar de formarlo en SQL para que en muy poco tiempo pueda incorporarse como programador a proyectos más complejos, en los que ahora mismo está trabajando __José__, y para los que necesitaría su ayuda.

__Carmen__ y __Víctor__ se miran un momento y sonríen. Es una buena noticia para __Víctor__, pero también para __Carmen__, porque sabe que eso les va a permitir compartir algunos buenos momentos en el trabajo.

¿Qué es el lenguaje SQL?

En las unidades didácticas anteriores has aprendido que mediante el **modelo Entidad-Relación** se pueden **modelizar** situaciones del mundo real, de manera que disponemos de una herramienta gráfica para trasladar los elementos de un sistema de información **y sus relaciones**, a un esquema manejable que puede ser fácilmente interpretado por cualquiera que conozca las reglas por las que se rige. Posteriormente has comprobado cómo se puede **trasladar ese modelo Entidad-Relación** a otro esquema de información más orientado a su tratamiento, como es el **modelo relacional**.

Pero te habrás preguntado: ¿Dónde acaba todo esto? **¿De qué manera podemos almacenar y tratar la información esquematizada en los modelos E-R y relacional por medio de un Sistema Gestor de Bases de Datos**? La respuesta a estas preguntas se llama **SQL**.

- **SQL** es un lenguaje que nos **permite interactuar con los SGBDRelacionales** para especificar las operaciones que deseamos realizar sobre los datos y su estructura.
- SQL son las siglas de **S**tructured **Q**uery **L**anguage (**Lenguaje de Consulta Estructurado).**
- Es un **lenguaje declarativo**, lo cual quiere decir que en él se especifica al Sistema Gestor de Base de Datos **qué queremos obtener** y no la manera de cómo conseguirlo.
- Es un **lenguaje no procedimental** porque no necesitamos especificar el procedimiento para conseguir el objetivo, sino el objetivo en sí. **No es un lenguaje de programación** como puede ser Java o C.

SQL, un poco de historia

El **origen de SQL** está íntimamente ligado al de las **bases de datos relacionales**. En 1970 **Codd** propuso el modelo relacional que ya has estudiado en unidades anteriores. Basándose en esta idea en los **laboratorios de IBM** se definió el lenguaje **SEQUEL** (Structured English QUEry Language) que posteriormente sería implementado por el SGBD **System R**. Más tarde otros fabricantes de SGBD lo adoptarían, de forma que se acabó convirtiendo en un estándar de la industria informática.

En 1986 el ANSI (Instituto Americano de estándares) publicó la **primera versión** estándar del lenguaje, hoy conocido como **SQL-86**. Al año siguiente este estándar es adoptado por la **ISO** (Organismo Internacional de Estandarización), y a partir de entonces se han sucedido nuevas versiones que ampliaban en capacidad a las anteriores.

SQL en la actualidad

Hoy día **SQL es el lenguaje de consulta y manipulación de datos más extendido** y utilizado por todos los **desarrolladores y fabricantes de SGBD**. A lo largo de los años se han ido ampliando sus características conforme se sucedían avances en la tecnología informática.

A continuación te presentamos una tabla que te explica la **evolución de este lenguaje desde el lejano 1986** hasta ahora.

Año	Versión	Alias	Características
			Primera publicación hecha por ANSI.
1986	SQL-86	SQL-87	
			Confirmada por ISO en 1987.
1989	SQL-89		Revisión menor.
1992	SQL-92	SQL2	Revisión mayor.

1999	SQL-99	SQL3	Entre otras características se añade la orientación a objetos.
2003	SQL-2003		Profundiza en la orientación a objetos e introduce características de XML.

Para saber más

Cómo ves por la tabla anterior, la orientación a objetos ha irrumpido con fuerza en la tecnología de SGBD. El siguiente enlace te muestra las características de esta tecnología en auge.

Bases de datos relaciones orientadas a objetos **[Versión en caché]**

Como ves **SQL es un lenguaje** con cierta antigüedad que **ha tenido que evolucionar** para **adaptarse a nuevas características y requerimientos**. Es importante resaltar que **SQL no es propiedad de ningún fabricante, sino que es una norma a seguir**, desgraciadamente los fabricantes de software no suelen implementar SQL puro en sus productos, sino que a menudo incorporan pequeñas variaciones para conseguir funcionalidades concretas en sus desarrollos. Esto hace que lo qué debería ser un estándar no lo sea completamente en la realidad.

Como resumen de lo anterior, un concepto importante que debes asimilar es **que SQL es el lenguaje que usan los SGBDR en la actualidad**, y como tal, su estudio y práctica resulta imprescindible en tu formación como desarrollador de aplicaciones.

__Víctor__ sabía de su aprendizaje en Java, que SQL se puede usar embebido en otros lenguajes de programación. De esta forma es como normalmente ha trabajado, metiendo las sentencias SQL dentro de una sentencia especial de Java que es la que se encargaba de ejecutarlas sobre la base de datos a la que se había conectado la aplicación.

Ahora __Carmen__, sin embargo, para comenzar a explicarle en profundidad SQL, comienza ejecutando algunas sentencias SQL en modo interactivo, usando un intérprete de órdenes que proporciona directamente el SGBD, de forma que no necesitan nada más para escribir esas sentencias, ejecutarlas, y ver el resultado.

Bien, **ya sabemos qué es SQL. Ahora vamos a aprender cómo podemos usarlo para nuestros propósitos**. No olvides que estamos estudiando un lenguaje de interacción con bases de datos y que la herramienta para acceder y manipular esas bases de datos son los sistemas gestores de base de datos.

Los SGBDR permiten dos modos de acceso a las bases de datos:

- **Modo interactivo**, destinado principalmente a los **usuarios finales, avanzados u ocasionales, en el que las diversas sentencias SQL se escriben y ejecutan en línea de comandos, utilizando un intérprete de órdenes**. Este modo es el que utilizaremos en las prácticas de esta unidad didáctica, porque es el que permite centrarnos exclusivamente en SQL y aprender el lenguaje de una forma directa y sin necesidad de otro tipo de conocimientos.
- **Modo embebido**, destinado al uso por parte de los **programadores. En este caso las sentencias SQL se introducen en lenguajes de programación, llamados lenguajes anfitrión** (por ejemplo Java, lenguajes de la plataforma .NET de Microsoft, PHP, C++, etc.), de manera que el **resultado es una mezcla de ambos**. En este caso **el lenguaje anfitrión aporta lo que le falta a SQL, es decir la programación**. Aunque éste es el modo habitual en el que se utiliza SQL cuando se desarrollan aplicaciones, su estudio de esta forma requeriría el estudio simultáneo del lenguaje anfitrión, por lo qué pospondremos su estudio a una unidad didáctica posterior para no complicar el estudio de lo que realmente nos interesa en este momento.

__Carmen__ le dice a __Víctor__ que para aprender SQL, no hay nada mejor que ir probando las distintas sentencias según las vayan estudiando. Pero claro, para poder hacer eso, lo primero que necesitan es elegir el SGBD que van a usar, e instalarlo en un equipo. __Víctor__ le propone usar __MySQL__, porque lo conoce, y recuerda cómo se instalaba y configuraba, y recuerda también que ese proceso era muy sencillo. A __Carmen__ le parece una buena idea. Se trata de un SGBD open source, que incorpora la mayoría de las características de SQL, y que al ser ya conocido por __Víctor__, les va a permitir centrarse rápidamente en el estudio de SQL, que es lo que por ahora les interesa. Además, en Internet existe mucha información sobre MySQL, incluso en castellano, lo que no siempre es frecuente para las herramientas informáticas. Sin duda, deberán consultar la documentación disponible en más de una ocasión.

No obstante, __Carmen__ le advierte a __Víctor__ que algunas cosas serán distintas dependiendo del SGBDR que se use, y aunque básicamente todo lo que va a aprender aquí sobre SQL será aplicable a cualquier otro SGBDR, deberá contar con que sobre todo el proceso de instalación y configuración va a ser radicalmente distinto, y posiblemente bastante más complejo, en otros SGBD, por lo que más adelante, tendrá que aprender a manejar algún otro SGBDR.

- "Si lo aprendo contigo, me dedicaré en cuerpo y alma a su estudio", le contesta __Víctor__.

- "Zalamerías", le comenta __Carmen__, divertida.

En este momento te estarás haciendo muchas preguntas, y la mejor manera de responderlas es empezando a **ver en la práctica qué es eso de SQL** y como usarlo.

Ya sabes que **a través de SQL interactuaremos con un SGBDR que a su vez gestiona una o más bases de datos relacionales**. Por lo tanto lo primero que debemos hacer para empezar a aprender y practicar SQL es disponer de un SGBDR en nuestro ordenador de prácticas.

Se nos presenta aquí la primera decisión que adoptar **¿Qué SGBDR utilizar?** La respuesta es al mismo tiempo sencilla y complicada, veamos:

- Es sencilla porque nos vale **cualquier SGBDR que implemente al menos SQL-99.**
- Es complicada porque existen bastantes productos en el mercado que cumplen ese requisito.

Para las prácticas de esta unidad didáctica hemos elegido un SGBD que debes conocer ya si has cursado el módulo profesional de "Programación en Lenguajes Estructurados (PLE)". Recordarás que en él se dedica una unidad didáctica al estudio del acceso a bases de datos. **Utilizaremos como SGBD de pruebas para esta unidad MySQL.**

Si no has estudiado ese módulo profesional, no te preocupes demasiado. En esta unidad también se explica cómo se descarga e instala MySQL y se ponen ejemplos de uso. Además, el uso que haremos aquí es distinto al que se hacía en el módulo de PLE, ya que sólo se trata de aprender la sintaxis de SQL practicando en modo interactivo, mientras que en aquel módulo de PLE se hacía un uso en modo embebido en el lenguaje Java.

MySQL es un producto open source, que en **su versión actual soporta muchas características de SQL-99**. Su obtención es fácil, **se puede descargar gratuitamente de la web** y existe mucha documentación accesible sobre él. Además es un producto de amplia implantación en el mercado actual y su instalación y manejo son lo suficientemente sencillos como para permitir concentrarte en la materia de estudio de esta unidad sin complicaciones extras derivadas del propio producto. Ni que decir tiene que **todo lo que se estudie aquí será válido en cualquier otro SGBDR**.

Lo primero que debes hacer es **descargar e instalar** el software de MySQL. Necesitaremos las siguientes aplicaciones:

- **Servidor MySQL**. El SGBDR en sí.
- **MySQL Administrator**. Una aplicación para administrar MySQL de forma gráfica.
- **MySQL QueryBrowser**. Un intérprete de sentencias SQL en modo gráfico, lo usaremos intensamente en esta unidad. Por eso es muy recomendable que te descargues el manual que te proporcionamos en el siguiente enlace Para saber más, después de la Zona de Descarga.

Zona de descarga

Desde el siguiente enlace puedes descargar las aplicaciones MySQL. Esto es, el servidor de base de datos, la aplicación de administración y el intérprete de SQL.

Página oficial de descargas de MySQL

A continuación tienes una serie de recursos que te serán de mucha utilidad para montar tu sistema MySQL de práctica en tu ordenador. Dispones de animaciones sobre la descarga, instalación de las aplicaciones anteriores, y un manual de uso de la aplicación Query Browser.

El siguiente enlace te muestra el proceso de descarga de las tres aplicaciones que vas a necesitar.

A continuación tienes una animación para ver el proceso de instalación del servidor MySQL.

La siguiente animación te muestra el proceso de instalación de MySQL Administrator.

La última animación presenta el proceso de instalación de MySQL QueryBrowser.

Para saber más

Te recomendamos que visites el siguiente enlace para disponer de un completo manual de usuario de QueryBrowser.

Manual de MySQL Query Browser **[Versión en caché]**

Cuando termines de instalar y configurar las aplicaciones anteriores **dispondrás de un SGBDR potente, sólido y flexible instalado en tu ordenador, listo para ser usado**... pero ¿Qué hay de las bases de datos? Sí, tenemos un SGBDR, pero tras la instalación no existe ninguna.

Vamos a crear una base de datos. Empezaremos con una muy **sencilla**, una base de datos con **una sola tabla.**

A través de la siguiente animación te presentamos los pasos necesarios. Es muy importante que intentes realizar esa misma práctica en tu propio sistema para familiarizarte con él.

Para saber más

Por último es muy conveniente ya que vas a utilizar el producto que visites la página web oficial del fabricante. También te facilitamos un enlace a una página no oficial en castellano sobre MySQL.

Página web oficial de MySQL

Una página dedicada a este servidor de bases de datos relacionales. Está en castellano.

MySQL hispano

Carmen *decide empezar explicándole a* ***Víctor*** *los componentes del lenguaje SQL, que no son más que una división que se hace de las sentencias del lenguaje, agrupándolas según la utilidad que tienen, o el tipo de operaciones que realizan. De esta forma, tendrá que explicarle los tres grandes grupos de sentencias SQL:*

- *De Definición de datos (**DDL**)*
- *De Manipulación de datos(**DML**)*
- *De control (**DCL**)*

***Víctor** ya conoce la existencia de estos sub-lenguajes, pero sólo a nivel teórico. **Carmen** le explica que de lo que se trata ahora es de comenzar a ver las principales sentencias SQL de cada uno de ellos.*

El lenguaje SQL está compuesto por **sentencias**. Esas sentencias se pueden clasificar en tres grupos:

- **Sentencias DDL (Lenguaje de Definición de Datos):** Sirven para **crear, modificar y borrar elementos estructurales en los SGBDR**, como:
  - bases de datos,
  - tablas,
  - índices,
  - restricciones, etc.

  Las definiciones de esos objetos quedan almacenadas en el diccionario de datos del sistema.
- **Sentencias DML (Lenguaje de Manipulación de Datos):** Nos permiten indicar al sistema las **operaciones que queremos realizar con los datos almacenados en las estructuras creadas por medio de las sentencias DDL**. Por ejemplo son las sentencias que permitirán:
  - generar consultas,
  - ordenar,
  - filtrar,
  - añadir,
  - modificar,
  - borrar,
  - etc.
- **Sentencias DCL (Lenguaje de Control):** Un conjunto de sentencias orientado a gestionar todo lo relativo a:
  - usuarios,
  - permisos,
  - seguridad,
  - etc.

Con el tiempo han surgido **nuevas necesidades** en los SGBDR que han obligado a incorporar **nuevas sentencias** que no se pueden clasificar en los tres grupos clásicos anteriores**, cómo son las Sentencias para gestión de transacciones y bloqueos, las sentencias para replicación, etc.**

Las sentencias SQL a su vez se construyen a partir de:

- **Cláusulas**: Que modifican el comportamiento de las sentencias. Constan de palabras reservadas y alguno de los siguientes elementos.
- **Operadores lógicos y de comparación:** Sirven para ligar operandos y producir un resultado booleano (verdadero o falso).
- **Funciones de agregación:** Para realizar operaciones sobre un grupo de filas de una tabla.
- **Funciones:** Para realizar cálculos y operaciones de transformación sobre los datos.
- **Expresiones:** Construidas a partir de la combinación de operadores, funciones, literales y nombres de columna.

A lo largo de esta unidad estudiaremos y practicaremos gran parte de las sentencias SQL, y por supuesto las más importantes y utilizadas. Aunque su estudio completo no sería posible aquí, por su elevadísimo número y casuística de cada una, y para completar tu formación te remitimos a los buenos manuales que sobre SQL circulan en la red y en las librerías. En especial **te recomendamos la documentación del SGBDR** que vas a utilizar, MySQL. Puedes obtenerla de la misma web desde donde has descargado la aplicación.

Para saber más

Te facilitamos dos enlaces útiles. El primero es un enlace a un tutorial de SQL en línea, el segundo es el enlace a la documentación en castellano de MySQL, desde la propia página web del fabricante.

Tutorial SQL

Manual castellano de MySQL

*Para explicarle las distintas sentencias SQL, **Carmen** ha pedido consejo a **María** sobre qué ejemplo podría usar para ir introduciéndolas todas. **María**, tras pensarlo un poco, le sugiere usar como ejemplo una base de datos para una aplicación en la que había estado trabajando recientemente. Nada mejor que un sistema real para empezar a practicar. Se trata de una **aplicación de gestión de proyectos en una empresa**. Aunque la aplicación es más amplia, le sugiere centrase sólo en una parte del problema, más reducida, pero que le permitirá probar todas las sentencias que desea. Esa parte incluye 3 tablas (de empleados, departamentos y proyectos) de forma que habrá empleados pertenecientes a algún departamento, y asignados al trabajo de algún proyecto.*

*La ventaja es que **María** ya tiene hecho tanto el diagrama E-R como el esquema relacional, con las tablas en 3FN, por lo que se las pasa a **Carmen** que elabora el plan de trabajo y se lo comunica a **Víctor**:*

Directamente pueden comenzar con las sentencias de definición de datos (DDL) que permiten definir y modificar la estructura de la base de datos, para crear, modificar o borrar la base de datos, las tablas, los índices, etc. Incluso tiene las sentencias de creación de la base de datos guardadas en un script, por lo que si les surgen dudas, siempre podrán comprobar que lo que han hecho es correcto.

Posteriormente practicarán con las sentencias de manipulación de datos (DML), que le permitirán insertar, borrar y modificar las filas de una tabla, hacer consultas a una o varias tablas y definir vistas de la base de datos.

Finalmente practicarán con alguna sentencia de control de datos (DCL) para asignar privilegios a los usuarios, o denegárselos, creando así perfiles de usuario y limitando lo que puede hacer cada uno, y sobre qué datos. También verán cómo crear transacciones para mejorar la seguridad de las operaciones sobre la base de datos, y todo ello dándole un repaso a las funciones SQL.

*A **Víctor** le parece muy difícil poder memorizar la sintaxis de todas las sentencias SQL que va a estudiar, pero **María** le tranquiliza: No todas se usan con frecuencia, por lo que basta con que conozca las más usadas, que son bastantes menos, y sepa de la existencia de las demás. En caso de necesidad, se consulta la abundante documentación que existe sobre SQL, que para eso está.*

En esta unidad didáctica estudiaremos cómo SQL permite crear bases de datos y también manipularlas. Comprobarás que la unidad que estás comenzando a estudiar está repleta de prácticas sobre lo estudiado. Y para hacer de hilo conductor vamos a plantear **un ejemplo** que nos permitirá probar todos los conceptos que vayamos introduciendo. De esta forma el ejemplo se irá enriqueciendo conforme avances en el estudio de los contenidos de esta unidad y su comprensión y asimilación desde el principio te permitirá concentrarte en los aspectos propios de SQL y no en los del caso concreto que se quiere modelizar.

Hemos escogido un ejemplo que es bastante conocido y que se ha utilizado ampliamente en los libros dedicados al tema de bases de datos relacionales. Si buscas por Internet comprobarás que muchas páginas y documentos dedicados al estudio de los modelos de datos lo utilizan.

Como ya has aprendido en unidades anteriores la implementación en un SGBD de una base de datos es la culminación de un proceso que pretende modelizar una situación del mundo real para poder sistematizarla y tratarla por medios informáticos. Ya sabes que los pasos que se siguen para diseñar una base de datos relacional son:

- **Estudio una situación del mundo real** que se pretende modelizar.
- Diseño de un modelo conceptual de la situación. **Diagrama Entidad-Relación.**
- Traslado del diseño conceptual a un diseño lógico**. Modelo relacional**.
- **Implementación del modelo relacional en el SGBDR** que se vaya a utilizar (SQL).

Planteamiento de la situación que queremos modelizar

Piensa en la siguiente situación del mundo real:

- Una **empresa** pretende desarrollar una base de datos de **empleados y proyectos**.
- La empresa está estructurada en **departamentos**, en cada uno de los cuales se están desarrollando uno o varios **proyectos**, de forma que un proyecto sólo depende de un departamento.
- Por otro lado, cada departamento consta de **uno o varios empleados** que trabajan de forma exclusiva para ese departamento, pero pueden trabajar simultáneamente en varios proyectos.
- Cada empleado tiene un **jefe** encargado de **supervisar** su trabajo, pudiendo cada jefe supervisar el trabajo de varios empleados.
- **Tanto los empleados, como los departamentos y los proyectos se identifican por un código que es único** para cada uno de ellos.
- De cada **empleado** se quiere guardar su nombre y su fecha de ingreso en la empresa, así mismo todos los departamentos y los proyectos tienen un nombre descriptivo.
- Por último, se quiere guardar en la base de datos **cuántas horas ha trabajado** cada empleado en los proyectos en los que está involucrado.

Hasta aquí el enunciado de nuestro caso práctico, veamos a continuación el diseño conceptual de este supuesto.

¿Has comprendido bien el enunciado de la situación planteada? Con las enseñanzas de las unidades didácticas anteriores **no tendrás problema en asimilar el siguiente diagrama E-R** que modeliza de forma conceptual la situación descrita.

Modelo relacional de nuestro caso práctico

Aplicando lo aprendido en las **unidades anteriores** no tendrás problemas en comprobar que el diseño lógico (esquema relacional) correspondiente al diagrama E-R anterior es el siguiente.

El esquema anterior está en 3FN (tercera forma normal) y consta de cuatro tablas. A partir de ahora y en lo que resta de esta unidad didáctica utilizaremos este esquema relacional para **ejemplificar los contenidos teóricos** relativos al lenguaje SQL. No olvides que el vehículo para realizar las prácticas será el gestor de base de datos MySQL, que debes tener instalado y en funcionamiento. También debes tener a mano la **documentación de MySQL** que se instaló junto con la aplicación, o en línea a través de la página web oficial del producto.

Carmen ya le ha explicado a Víctor todas las sentencias DDL que por ahora van a necesitar, y es momento de ponerse a crear la estructura de la base de datos para su ejemplo. La lista es relativamente corta, aunque a Víctor no se lo parece:

*Para bases de datos: **`CREATE DATABASE`**, **`DROP DATABASE`** y **`SHOW DATABASES`***

*Para tablas: **`CREATE TABLE`**, **`ALTER TABLE`** y **`DROP TABLE`***

*Para índices: **`CREATE INDEX`** y **`DROP INDEX`***

Como Carmen le hace ver, es bastante sencillo, porque los nombres de las sentencias son bastante autoexplicativos de su finalidad.

*Además, **CREATE** no puede ser más que crear, **ALTER** no puede ser más que alterar o modificar y **DROP** no puede ser más que borrar o eliminar. Basta con poner detrás el tipo de elemento sobre el que quieres que se actúe: **DATABASE**, **TABLE** o **INDEX***

*Luego está **SHOW DATABASES**, que no pude servir más que para ver la lista de bases de datos que hay alojadas en el servidor. Parece lógico, ¿no?*

Pero Víctor le responde que lo que le parece complicado no es la lista de sentencias, sino la lista de cláusulas y la sintaxis completa de cada una de ellas, que puede ser bastante larga, y no siempre tan evidente. Presiente que va a tener que practicar mucho para trabajar con soltura con todas ellas.

- "Justamente para eso estamos aquí, para que empieces a practicar", - le contesta Carmen

Y le ha puesto tarea: Debe crear la base de datos, las tablas y los índices a partir de los diagramas de las tablas que les pasó María, con sus correspondientes llaves primarias y llaves externas, y practicar luego con las sentencias de borrado y modificación hasta que esté familiarizado con su uso, asegurándose de dejar la base de datos lista para seguir practicando con el próximo grupo de sentencias.

- "Al tajo pues, que mientras antes se empieza, antes se termina", -comenta Víctor antes de ponerse a trabajar. "¿Si termino antes de la hora de salir del trabajo me invitas a un café?"

- "Hecho. Y si te lo curras, pues igual te invito también aunque no termines".- dice Carmen.

Ya sabes que **el lenguaje SQL se compone de sentencias**, y que éstas **se pueden clasificar en tres grupos**.

Empezaremos estudiando **aquéllas que nos permiten crear, modificar y borrar las estructuras de nuestra base de datos.**

Este grupo de sentencias **constituyen el Lenguaje de Definición de Datos (DDL).** Las sentencias del DDL suelen ser bastante dependientes del SGBD que estemos utilizando, por lo cuál tendremos que consultar su documentación para conocer todos los detalles.

Creación y borrado de una base de datos

Lo primero que debes aprender es a crear la base de datos, puesto que éste es el objeto que contiene a todos los demás elementos.

Cuando se crea una base de datos, en el SGBD ocurren muchos **procesos internos** que culminan con la asignación de **espacio físico** de almacenamiento para contener las estructuras de datos y acceso a los mismos de la base de datos, y también el registro en el diccionario de datos del SGBD de las características de la base de datos creada.

Las herramientas que vamos a utilizar para practicar lo aprendido en esta unidad son las aplicaciones MySQL Administrator y MySQL Query Browser, en concreto usaremos MySQL Query Browser para practicar todas las sentencias SQL que vayamos aprendiendo, puesto que MySQL es un intérprete de órdenes SQL.

En el caso de la creación de una base de datos se puede hacer según lo indicado en la animación que te presentamos en el apartado 3 de esta misma unidad, utilizando MySQL Administrator. Como recordarás entonces creamos una base de datos llamada **biblioteca**.

Podemos ahora seguir los mismos pasos para crear la base de datos de nuestro ejemplo, a la qué llamaremos **gestionproyectos**.

El otro modo de crear una base de datos es utilizando el MySQL Query Browser y tecleando la sentencia SQL adecuada. A continuación te presentamos la sintaxis de la sentencia para crear bases de datos en SQL, **CREATE DATABASE**.

```
CREATE DATABASE [IF NOT EXISTS] <nombre_bd>
```

Crea una base de datos llamada **<nombre_bd>**. La cláusula **IF NOT EXISTS** es útil para evitar errores si es que la base de datos ya existía con anterioridad.

En nuestro ejemplo la sentencia SQL adecuada sería:

```
CREATE DATABASE IF NOT EXISTS gestionproyectos
```

Si ahora tecleamos la sentencia **SHOW DATABASES**, obtendremos la lista de las bases de datos que existen en nuestro SGBD, comprobando que aparece la recién creada **gestionproyectos.**

Existen reglas para construir nombres válidos de bases de datos, pero éstas son propias de cada sistema por lo que deberemos consultar la documentación del SGBDR que estemos utilizando. En MySQL deberás consultar cuáles son y tenerlas presentes siempre que nombres una base de datos

Para saber más

En MySQL las reglas de formación de nombres válidos puedes consultarlas en el siguiente enlace:

Reglas para construir nombres válidos en MySQL **[Versión en caché]**

Hay que hacer notar que la sintaxis de la sentencia **CREATE DATABASE** varía mucho de unos SGBD a otros y normalmente **se pueden especificar muchos parámetros en el momento de la creación**, como:

- **El juego de caracteres utilizado.**
- **El Sistema de almacenamiento físico**.
- El modo de ordenación de caracteres (collation).
- **Etc**.

La documentación de cada sistema en particular nos dará las distintas posibilidades. Por ejemplo en MySQL tenemos:

```
CREATE DATABASE [IF NOT EXISTS] <nombre_bd>
    [[DEFAULT] CHARACTER SET <juego_caracteres>]
    [[DEFAULT] COLLATE <modo_ordenación>]
```

Para saber más

Aquí tienes la sintaxis e información relativa a la creación de bases de datos en otros dos gestores de bases de datos relacionales muy utilizados. Compara esta información con la que dispones de MySQL, observarás que hay muchas diferencias.

Creación de bases de datos en SQL Server de Microsoft **[Versión en caché]**

Creación de bases de datos en Oracle **[Versión en caché]**

Para borrar una base de datos se utiliza la sentencia **DROP DATABASE**:

```
DROP DATABASE [IF EXISTS] <nombre_bd>
```

En nuestro ejemplo deberíamos escribir: **DROP DATABASE gestionproyectos**

Hay que tener cuidado ya que borraremos la base de datos y todo lo que contiene. Los SGBD introducen medidas de seguridad de forma que sólo los usuarios autorizados (normalmente el administrador y el propietario de la base de datos) pueden ejecutar este tipo de sentencias. Veremos la gestión de la seguridad y los usuarios más adelante.

Creación, modificación y borrado de tablas

En este momento dispones de un SGBD funcionando y una base de datos recién creada, pero **esa base de datos no es más que una estructura vacía. Ya sabes que los objetos que se utilizan para albergar datos en un SGBD relacional son las tablas** y a continuación vamos a aprender a crearlas, modificarlas y borrarlas.

Para **crear una tabla** en SQL se utiliza la sentencia **CREATE TABLE**.

A continuación te presentamos una **sintaxis reducida de esta sentencia**:

```
CREATE TABLE [IF NOT EXISTS] nombre_tabla
    (definicion_elemento_de_tabla,...)

definicion_elemento_de_tabla:
    definición_columna
  | PRIMARY KEY (nombre_columna,...)
  | FOREIGN KEY (nombre_columna,...) [definicion_referencia]

definicion_columna:
    nombre_columna tipo_dato [NOT NULL | NULL] [DEFAULT valor_defecto]
      [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]

tipo_dato:
    BIT[(longitud)]
  | INT[(longitud)] [UNSIGNED] [ZEROFILL]
```

```
  | INTEGER[(longitud)] [UNSIGNED] [ZEROFILL]
  | REAL[(longitud,decimales)] [UNSIGNED] [ZEROFILL]
  | DOUBLE[(longitud,decimales)] [UNSIGNED] [ZEROFILL]
  | FLOAT[(longitud,decimales)] [UNSIGNED] [ZEROFILL]
  | DECIMAL(longitud,decimales) [UNSIGNED] [ZEROFILL]
  | NUMERIC(longitud,decimales) [UNSIGNED] [ZEROFILL]
  | DATE  | TIME  | TIMESTAMP  | DATETIME  | YEAR
  | CHAR(longitud) | VARCHAR(longitud)
  | BINARY(longitud)
  | BLOB
  | TEXT [BINARY]
  | ENUM(valor1,valor2,valor3,...)
  | SET(valor1,valor2,valor3,...)

definicion_referencia:
    REFERENCES nombre_tabla [(columna1,...)]
               [ON DELETE opcion_referencia]
               [ON UPDATE opcion_referencia]

opcion_referencia:
    RESTRICT | CASCADE | SET NULL | NO ACTION
```

> ***Para saber más***
>
> ***Puedes consultar la sintaxis completa de esta sentencia en las páginas de documentación de MySQL.***
>
> *Creación de tablas en MySQL* [Versión en caché]

Habrá que escoger el dominio más adecuado para los valores que debe contener cada columna. **MySQL**, como todos los SGBDR, **ofrece una amplia variedad de tipos de dato diferentes**. La mayoría de ellos pertenecen al estándar SQL, pero como siempre habrá que consultar la documentación del sistema que estamos utilizando.

> ***Para saber más***
>
> ***Sería muy largo tratar pormenorizadamente todos los tipos de dato disponibles, por eso te recomendamos que consultes los siguientes enlaces de la documentación de MySQl.***
>
> *Tipos de dato numéricos en MySQL* [Versión en caché]
>
> *Tipos de dato fecha y hora en MySQL* [Versión en caché]
>
> *Tipos de dato de carácter en MySQL* [Versión en caché]

Cualquier tipo de dato utilizado como dominio de una columna puede contener el valor NULL. Este valor hay que interpretarlo como **ausencia de valor**, no hay que confundirlo, por ejemplo, con el valor 0 de una columna de tipo `INTEGER` o con la cadena vacía en una columna de tipo `VARCHAR`. Una columna que forme parte **de la clave primaria de una tabla no puede contener el valor NULL**, y **tampoco aquellas columnas definidas como NOT NULL**, será el propio SGBDR el encargado de mantener esta restricción e impedirá cualquier intento contrario a la regla.

En nuestro ejemplo la creación de tablas quedaría así:

```
CREATE TABLE departamento (
```

```
  cddep CHAR(2) NOT NULL PRIMARY KEY,
  nombre VARCHAR(30),
  ciudad VARCHAR(20)
) ENGINE=INNODB;

CREATE TABLE empleado (
  cdemp CHAR(3) NOT NULL PRIMARY KEY,
  nombre VARCHAR(30),
  fecha_ingreso DATE,
  cdjefe CHAR(3) NULL,
  cddep CHAR(2),
  FOREIGN KEY (cddep)
  REFERENCES departamento(cddep)
  ON UPDATE CASCADE ON DELETE RESTRICT,
  FOREIGN KEY (cdjefe)
  REFERENCES empleado(cdemp)
  ON UPDATE CASCADE ON DELETE SET NULL
) ENGINE=INNODB;

CREATE TABLE proyecto (
  cdpro CHAR(3) NOT NULL PRIMARY KEY,
  nombre VARCHAR(30),
  cddep CHAR(2)
  FOREIGN KEY (cddep)
  REFERENCES departamento(cddep)
  ON UPDATE CASCADE ON DELETE RESTRICT
) ENGINE=INNODB;

CREATE TABLE trabaja (
  cdemp CHAR(3) NOT NULL,
  cdpro CHAR(3) NOT NULL,
  nhoras INTEGER DEFAULT 0,
  PRIMARY KEY (cdemp,cdpro),
  FOREIGN KEY (cdemp)
  REFERENCES empleado(cdemp)
  ON UPDATE CASCADE ON DELETE CASCADE,
  FOREIGN KEY (cdpro)
```

```
  REFERENCES proyecto(cdpro)

  ON UPDATE CASCADE ON DELETE CASCADE

) ENGINE=INNODB;
```

Algunas consideraciones al respecto:

- **Nombres válidos:** Al igual que en el caso de los nombres válidos de base de datos**, los nombres de tablas, índices y columnas deben seguir unas reglas que dependen del SGBDR**. Te remitimos al apartado dedicado a la creación y borrado de bases de datos, donde ya se trató este aspecto para el caso particular de MySQL.
- **Restricciones de integridad y referencia**: Observa que **hemos definido junto con las columnas de cada tabla las restricciones para hacer cumplir la reglas de integridad de la entidad (`PRIMARY KEY`) y de referencia (`FOREIGN KEY`)**.

> **¡Importante!:**
>
> **En MySQL, para que tengan efecto las definiciones de claves externas, debe utilizarse el modo de almacenamiento de tablas InnoDB (`ENGINE=INNODB`).**
>
> **Por ejemplo si queremos que se cumpla la integridad de referencia en la columna cddep de la tabla empleado, de manera que todos los empleados tengan un código de departamento que exista en la tabla departamento debemos utilizar la estructura:**

```
FOREIGN KEY (<columnas que forman la clave externa>)

 REFERENCES tabla(<columnas que forman la clave primaria>)
```

Funciona de la siguiente manera: las **columnas indicadas en `FOREIGN KEY` hacen referencia a las columnas indicadas en `REFERENCES`**. Se puede indicar la acción a realizar cuando se modifican (`ON UPDATE`) o se borran (`ON DELETE`) los valores especificados en `REFERENCES`, esta acción puede ser: impedir, arrastrar el cambio, poner a nulo o no hacer nada (respectivamente: `RESTRICT, CASCADE, SET NULL, NO ACTION`).

En SQL quedaría así:

```
FOREIGN KEY (cddep)

   REFERENCES departamento(cddep)

   ON UPDATE CASCADE ON DELETE RESTRICT
```

Lo que se interpreta como: **La columna cddep de la tabla empleado hace referencia (`FOREIGN KEY (cddep)`) a la columna cddep de la tabla departamento (`REFERENCES departamento(cddep)`).** No puede existir una fila en empleado que contenga un valor de cddep que no exista previamente en la tabla departamento. Si se modifica el cddep de una fila en la tabla departamento (`ON UPDATE`) el cambio se transmite automáticamente a todas las filas de la tabla empleado que tuviesen ese departamento (`CASCADE`). Si se intenta borrar una fila de la tabla departamento (`ON DELETE`) , la operación se cancelará si existen filas en la tabla de empleados con ese valor de departamento (`RESTRICT`).

> ***Para saber más***
>
> ***Lee con atención el siguiente texto extraído de la documentación de MySQL sobre claves externas.***
>
> ***Restricciones de clave externa en MySQL*** **[Versión en caché]**

Es posible **modificar la estructura de una tabla** por medio de la sentencia `ALTER TABLE`. Su sintaxis es muy parecida a la de `CREATE TABLE`, lee el siguiente enlace para estudiar la sintaxis y ver algunos ejemplos.

> ***Para saber más***
>
> ***Sintaxis y ejemplos de la sentencia ALTER TABLE en MySQL*** **[Versión en caché]**

El **borrado de una tabla** es muy simple, aunque hay que tener cuidado en esta operación, porque un borrado indebido puede tener consecuencias graves, si se borran datos que no se debían borrar. Asegúrate dos veces siempre antes de usar esta sentencia.

La sentencia a utilizar es:

```
DROP TABLE <nombre_tabla>
```

Por ejemplo, para borrar la tabla empleado:

```
DROP TABLE empleado
```

> ***Para saber más***

Lee el siguiente enlace con información concreta sobre el borrado de tablas en MySQL

Borrado de tablas en MySQL **[Versión en caché]**

Creación y borrado de índices

Cuando tenemos que buscar **manualmente** un dato en una lista que es muy larga, solemos tardar bastante más que si la lista es pequeña, incluso aunque hagamos la búsqueda aplicando algún **criterio** para no tener que comprobar todos los datos uno a uno desde el principio.

- ¿Ocurre algo parecido en las bases de datos?
- ¿El tiempo que tarda el SGBD en encontrar un dato es independiente del número de datos que contengan las tablas?

La respuesta es que **no**. Cuando las tablas de una base de datos van aumentando de tamaño, es decir van teniendo más y más filas, las operaciones que se realizan sobre ellas van siendo cada vez más lentas.

Los índices son unas estructuras gestionadas por el SGBDR para agilizar el acceso a datos y los cálculos. El uso de índices es gestionado automáticamente por el motor del SGBDR, en concreto por el optimizador de consultas, encargado de decidir en cada momento qué índice usar y cómo hacerlo.

Cuando creamos una tabla y definimos columnas como `PRIMARY KEY` o utilizamos la cláusula `FOREIGN KEY` el SGBDR crea de forma automática los índices adecuados. También podemos crear índices de manera explícita utilizando la siguiente sintaxis:

```
CREATE INDEX <índice> ON <tabla>(<columna1>,<columna2>,...)
```

Por **ejemplo**, para crear un índice de la tabla empleado llamado emp_nombre, y basado en la columna nombre del empleado utilizaremos la siguiente sentencia.

```
CREATE INDEX empleado_nombre ON empleado (nombre)
```

A partir de ese momento el motor de datos del SGBDR usará el índice para agilizar las operaciones sobre la tabla empleado que impliquen a la columna nombre. Hay que hacer notar que en tablas pequeñas el uso de índices puede tener un impacto negativo en el rendimiento.

En el caso de índices basados en **columnas de tipo texto CHAR y VARCHAR**, es posible y muy interesante limitar el número de caracteres que formarán parte del índice. Por ejemplo, para crear el índice anterior usando sólo los primeros 10 caracteres podemos escribir.

```
CREATE INDEX empleado_nombre ON empleado (nombre(10))
```

Para saber más

Los índices tienen gran importancia para manejar con agilidad tablas con muchas filas. Es muy conveniente que leas lo que aparece en la documentación de MySQL sobre los índices.

Información sobre índices de la página oficial de MySQL **[Versión en caché]**

La siguiente animación te muestra cómo podemos comprobar qué índices existen sobre una determinada tabla.

Para borrar un índice simplemente usaremos `DROP INDEX <índice> ON <tabla>`.

En el ejemplo anterior.

```
DROP INDEX empleado_nombre ON empleado
```

Creación de la base de datos para prácticas

Vamos a aplicar todo lo anterior a la **creación de la base de datos sobre Gestión de Proyectos** que utilizaremos en nuestras prácticas de esta unidad. Como estamos utilizando MySQL utilizaremos las herramientas que nos proporciona. Ya sabes que son MySQL Administrator y MySQL Query Browser.

Se puede crear una base de datos y sus tablas por medio de **ventanas de formulario** usando MySQL Administrator, esto ya lo has practicado en esta misma unidad.

Pero vamos ahora a usar otro método utizando MySQL Query Browser y las sentencias SQL estudiadas hasta ahora. En la siguiente **animación** te explicamos todo lo necesario para crear la base de datos utilizando las sentencias SQL adecuadas.

Utilizaremos una pestaña de script. Un script SQL es una agrupación de sentencias SQL que pueden ser editadas, almacenadas y recuperadas para hacer más productivo nuestro trabajo.

Visualiza ejemplos de scripts

Es conveniente grabar nuestro script para poder utlizarlo siempre que lo necesitemos sin tener que volverlo a escribir. **Te proporcionamos a continuación el script utilizado para la creación de la base de datos gestiónproyectos.** Observa que los

comentarios son útiles y están delimitados entre "**/***" y "***/**".

Descarga el script

*Efectivamente, **Carmen** invitó a **Víctor** a un café, con pastelitos incluidos, porque terminó el trabajo asignado a tiempo, y correctamente realizado.*

*Pero **Víctor** sabe que hoy le espera continuar el trabajo. Ya está asumido el uso de sentencias de definición de datos, pero una vez que hemos definido la estructura de la base de datos, toca llenarla con **datos de verdad**. De eso se encargan las sentencias SQL de manipulación de datos (**DML**), y nada más llegar al trabajo, **Carmen** le entrega la lista de sentencias que usarán para este fin, y que deberán empezar a repasar. Después **Víctor** se tendrá que encargar de aplicarlas para introducir los datos de prueba, que **Carmen** le ha proporcionado en unas fotocopias, en la base de datos, practicando las sentencias de inserción (**INSERT**), modificación (**UPDATE**) y borrado (**DELETE**)*

*Posteriormente sobre esos datos, se encargará de hacer bastantes consultas, unas más simples y otras más complicadas, para practicar con todas las posibilidades de la sentencia **SELECT**, que son muchas, pero que son muy importantes, ya que la operación que más frecuentemente se hace sobre los datos de la base de datos es precisamente consultarlos.*

*- "Si lo terminas todo para hoy, te invito a cenar", - le dice **Carmen** para motivarlo.*

*- "Me temo que hoy tendré que cenar solo, porque esto es importante, y no quiero hacerlo atropelladamente" - contesta **Víctor**.*

Si has seguido las explicaciones del apartado anterior y has practicado el ejemplo propuesto como hilo conductor de esta unidad, en este momento **dispones de un SGBDR robusto y muy utilizado en la actualidad, ejecutándose en tu ordenador (MySQL), y una base de datos (gestionproyectos)** con sus tablas vacías.

Llega ahora el momento de **empezar a llenar de filas esas tablas y manipular esos datos**. Para ello SQL dispone de una amplia **variedad de sentencias** que cubren todas las necesidades de tratamiento de datos implementando todas las operaciones del álgebra relacional.

Comprobarás que el **repertorio de sentencias SQL** y sus diferentes posibilidades es muy extenso, por lo que no podemos estudiarlo en su totalidad. Como siempre te recomendamos que consultes las páginas de la documentación de MySQL para completar lo tratado en este apartado.

Empezaremos con las sentencias **que permiten añadir, modificar y borrar** filas a las tablas y continuaremos con las sentencias que permiten consultar y manipular los datos. Todo estará acompañado de ejemplos que es muy importante que vayas practicando.

Inserción de filas en una tabla

Si partimos de una estructura de tablas vacías lo primero será aprender a **insertar filas** en ellas. Para esto se utiliza la sentencia **INSERT**.

Existen dos formas básicas de la sentencia **INSERT**, añadir una fila indicando los valores que deben tomar las columnas, o bien extraer las filas de una tabla ya existente y añadirlas a otra tabla. Veamos las sintaxis correspondientes a los dos modos de operación indicados:

Añadir una fila a una tabla con expresión de los valores de las columnas:

```
INSERT [INTO] <tabla> [(<columna>,...)]
  VALUES ({<expresión> | DEFAULT},...)
```

Al escribir una sentencia **INSERT** el **número de columnas y de valores debe ser igual y escrito en el mismo orden.** Se pueden dejar columnas sin especificar, en este caso tomarán el valor por defecto indicado cuando se creó la tabla (**DEFAULT**).

Por ejemplo, si queremos añadir el departamento con código "02", llamado "Ventas" y situado en la ciudad de "Almería" escribiremos:

```
INSERT INTO departamento
  (cddep,nombre,ciudad)
  VALUES ('02','Ventas','Almería')
```

Si no queremos suministrar la ciudad del departamento escribiremos:

```
INSERT INTO departamento (cddep,nombre) VALUES ('10','Contabilidad')
```

En este caso no le hemos dado valor a la columna ciudad.

Si quisiéramos añadir el empleado con código "A03", de nombre "Pedro Rojo", con fecha de ingreso 7 de marzo de 1995, cuyo jefe es el empleado con código "A11" y que trabaja en el departamento "01", deberíamos escribir:

```
INSERT INTO empleado (cdemp,nombre,fecha_ingreso,cdjefe,cddep)
  VALUES ('A03','Pedro Rojo','1995-3-7','A11','01')
```

El ejemplo anterior merece varios comentarios:

- **Si el empleado con código "A11" no existe en la tabla de empleados la sentencia INSERT fallará** y no tendrá ningún efecto, en este caso se habrá incumplido la cláusula **FOREIGN KEY** que indica que cdjefe es una clave externa y por lo tanto **se está incumpliendo la regla de integridad de referencia**. Podemos añadir el empleado "A11" a la tabla de empleados y repetir la inserción del empleado "A03". Observa que también existe una cláusula **FOREIGN KEY** respecto de la tabla de departamentos.
- La columna fecha_ingreso se declaró de tipo **DATE**. **En MySQL las fechas deben ser suministradas en el formato 'año-mes-día', que es el formato en inglés. Ten cuidado con esto. La introducción de fechas con nuestro formato habitual en español puede ser fuente de frecuentes errores...**

Para saber más

Las columnas de fecha y hora pueden ser complicadas de tratar por la gran cantidad de formatos y variaciones que admiten. En el siguiente enlace tienes información interesante sobre cómo trata MySQL las columnas de tipo fecha y hora.

Notas sobre el uso de tipos de dato de fecha y hora en MySQL [Versión en caché]

Añadir filas a una tabla procedentes de otra tabla:

```
INSERT [INTO] <tabla> [(<columna>,...)]

  SELECT ...
```

Para entender bien la estructura de esta sintaxis deberíamos conocer la sentencia SQL que permite seleccionar filas de una tabla (SELECT), puesto que la selección de las filas de origen se hace con una sentencia SELECT normal. Como **esta sentencia será estudiada más adelante en esta unidad** posponemos su explicación en detalle hasta ese momento. Ahora te presentamos un sencillo ejemplo:

Si quisiéramos añadir todos los empleados del departamento "02" al proyecto con código "AEE", suponiendo que existe tal proyecto en la tabla de proyectos, la sentencia a utilizar sería:

```
INSERT INTO trabaja (cdemp,cdpro)

  SELECT cdemp,'AEE' FROM empleado WHERE

  cddep='02'
```

No te preocupes si en este momento no entiendes bien cómo funciona la sentencia **SELECT**. Más adelante las estudiarás en detalle, ya que es la sentencia más importante y versátil de SQL.

Modificación de filas de una tabla

La información contenida en las tablas de una base de datos a menudo necesita ser actualizada para corregir errores de inserción, o para cambiar los valores de las columnas de forma que se reflejen los cambios en la situación que la base de datos está representando.

En nuestro **ejemplo** podríamos querer modificar los datos ya almacenados para corregir el nombre de un empleado mal escrito en el momento de la inserción, o para reflejar que un departamento ha cambiado de ciudad.

La sentencia SQL para modificar datos de una tabla es:

```
UPDATE <tabla>

  SET <columna_1>=<expresión_1> [, <columna_2>=<expresión_2> ...]

  WHERE <criterio>]
```

Si no se especifica ningún criterio de selección la modificación se efectúa sobre todas las filas de la tabla. En cambio si queremos modificar solamente alguna o algunas filas concretas habrá que determinar a cuales nos referimos por medio de la cláusula WHERE, especificando una condición lógica que permita discriminar a qué filas afectará la operación (criterio).

Por ejemplo, si queremos cambiar la ciudad donde se ubica el departamento "02", y hacer que sea "Córdoba", escribiremos:

```
UPDATE departamento

  SET ciudad='Córdoba'

  WHERE cddep='02'
```

Es importante que te des cuenta que si hubiéramos escrito:

```
UPDATE departamento

  SET ciudad='Córdoba'
```

¡Habríamos cambiado TODOS los departamentos a "Córdoba"!

Es posible especificar expresiones como nuevo valor de columnas. Si deseamos incrementar en 5 las horas de trabajo dedicadas al proyecto "AEE" por todos los empleados que trabajan en él, deberíamos escribir:

```
UPDATE trabaja

SET nhoras=nhoras+5

WHERE cdpro='02'
```

Si se intenta hacer un cambio que entra en conflicto con alguna de las reglas de la tabla, se producirá un error y el cambio no será llevado a efecto. Por ejemplo si intentamos modificar el valor de la clave primaria de una fila por un valor que exista previamente (recuerda que no pueden existir dos filas con el mismo valor de clave primaria), la operación no se efectuará. **Esto lo gestiona automáticamente el SGBDR**, asegurando la integridad de los datos de la base de datos.

Asimismo, si modificamos el valor de una columna que forma parte de una restricción `FOREIGN KEY` el SGBDR actuará según lo indicado en `ON UPDATE`. Por ejemplo si actualizamos el valor de un cddep en la tabla departamento el cambio se transmitirá a todos los empleados y proyectos que referenciaban a ese departamento. Esto es así puesto que en la creación de la base de datos fijamos **`ON UPDATE`** al valor **`CASCADE`**.

```
UPDATE departamento

 SET cddep='22'

 WHERE cddep='02'
```

Cambia el cddep del departamento "02" por "22", en la tabla de departamentos, y también en las tablas proyecto y empleado. El cambio se transmite en cascada tal y como indicamos en la reglas **`FOREIGN KEY`** correspondientes.

Borrado de filas de una tabla

Para borrar filas de una tabla se utiliza la sentencia `DELETE`. Permite borrar una sola fila o un conjunto de éstas. Para indicar qué filas serán borradas se utiliza una cláusula **`WHERE`** que indica mediante una condición lógica qué filas serán borradas. La sintaxis es la siguiente:

```
DELETE FROM <tabla>

[WHERE <criterio>]
```

Por ejemplo si queremos eliminar los empleados que pertenecen al departamento "02" escribiríamos:

```
DELETE FROM empleado

 WHERE cddep='02'
```

> **¡Cuidado! Si no especificamos ningún criterio se eliminarán todas las filas de la tabla.**

Por supuesto **el borrado de filas no debe dejar la base de datos en un estado de inconsistencia**. Esto quiere decir que **el SGBDR comprobará las restricciones `FOREIGN KEY` definidas** y actuará según lo indicado en **`ON DELETE`**. Veamos un ejemplo:

Si intentamos **eliminar un departamento** que es referenciado en la tabla de empleados, es decir, un departamento donde trabaja al menos un empleado, el SGBDR abortará la operación de borrado. Sólo se podrán eliminar departamentos en los que no trabaje ningún empleado. Esto es así porque se especificó **`ON DELETE RESTRICT`** al definir la clave externa cddep de la tabla empleado.

```
DELETE FROM departamento

 WHERE cddep='02'
```

Abortará si existe alguna fila en la tabla empleado cuyo valor de columna cddep sea "02".

Ejercitando las sentencias `INSERT`, `UPDATE` y `DELETE`

En los apartados anteriores **has practicado las sentencias SQL que permiten añadir, actualizar y eliminar** filas de las tablas de una base de datos. Antes de seguir con las siguientes sentencias SQL que vamos a estudiar y practicar, es importante que dispongamos de una base de datos que contenga un conjunto de datos lo suficientemente amplio y que refleje las distintas situaciones que pueden darse en el ejemplo del mundo real que estamos modelando y que hemos adoptado como ejemplo común para toda esta unidad.

Aunque tú podrías elegir el contenido que quisieras **para la base de datos de GESTIONPROYECTOS, te proponemos a continuación un posible contenido de las tablas que puedes introducir con las sentencias SQL `INSERT` adecuadas.** También debes practicar las sentencias **`UPDATE`** y **`DELETE`**. Es importante que sean precisamente esos datos para que puedas comprobar las soluciones de los ejercicios propuestos.

Esos datos de prueba que te pedimos que insertes en las tablas están disponibles en el siguiente enlace, que te aconsejamos que imprimas para tenerlo delante al realizar y comprobar tus ejercicios:

Ejercicios

En el conjunto de datos de prueba anterior se ha procurado abarcar los distintos casos que pueden darse en la situación del mundo real que estamos tratando. En especial:

- Departamentos sin trabajadores.
- Empleados con datos desconocidos o que no proceden, marcados como **NULL** .
- Empleados que no están en ningún proyecto, en uno o en varios.
- Departamentos que no gestionan proyectos, un proyecto o varios proyectos.
- Ciudades con un departamento o con varios.
- Existen columnas con valores NULL, indicando valor desconocido (fecha_ingreso) o dato no necesario (cdjefe del empleado "A11").

Esta variedad de casos nos permitirá probar las sentencias de SQL más utilizadas que serán estudiadas a continuación, las sentencias de selección **`SELECT`**.

Te sugerimos que escribas las sentencias `INSERT` adecuadas en un script SQL, esto te será útil para poder regenerar fácilmente la base de datos al estado original de prueba propuesto en este apartado.

Para facilitar tu trabajo te facilitamos el script SQL completo y listo para su uso.

Consultas sobre las filas de una tabla. Sentencia `SELECT`

Hasta ahora las sentencias SQL estudiadas producían cambios en la estructura de la base de datos o sobre las filas de las tablas. **Veremos ahora cómo podemos realizar consultas sobre los datos almacenados**, de forma que podamos obtener información de los datos existentes.

Las consultas en SQL son la operación más común y se hacen con la sentencia **`SELECT`**. Una sentencia **`SELECT`** tiene como sintaxis más básica y sencilla la compuesta por las cláusulas **`SELECT`** y **`FROM`**.

```
SELECT <lista_de_expresiones>
  FROM <tabla>
```

Siendo <lista_de_expresiones> una lista de expresiones a evaluar por cada fila afectada de la tabla, (normalmente serán nombres de columnas). Por **ejemplo**, para obtener los nombres de todos los departamentos y las ciudades donde se encuentran, se teclea en SQL:

```
SELECT nombre,ciudad
  FROM departamento
```

Si queremos obtener todas las columnas de una tabla se puede utilizar cómo <lista_de_expresiones> el carácter "*". Por ejemplo, para obtener todos los datos de todos los empleados podemos utilizar la siguiente sentencia **`SELECT`**:

```
SELECT *
  FROM empleado
```

Se pueden utilizar expresiones que realicen algún cálculo u operación. Por ejemplo, si suponemos que cada hora de trabajo se paga a 40 euros. Una lista de lo que hay que pagar a cada empleado que trabaja en un proyecto se haría de la siguiente manera:

```
SELECT cdemp, nhoras*40
  FROM trabaja
```

Podemos utilizar el programa MySQL Query Browser para pasar sentencias al SGBDR y obtener los resultados.

Observa la siguiente animación en la que se ejecutan los ejemplos anteriores.

La sentencia **`SELECT`** admite otras cláusulas que la hacen más potente y versátil. Estas son las siguientes:

- **`WHERE`**
- **`ORDER BY`**
- **`GROUP BY`**
- **`HAVING`**
- **`LIMIT`**

Las veremos en detalle en el próximo apartado.

Cláusula WHERE

Hasta ahora hemos utilizado sentencias de consulta`SELECT` que no discriminaban las filas de la tabla sobre la que se realizaba la consulta. En la práctica se utiliza mucho esa discriminación y se consigue con la cláusula **`WHERE`**. Esta cláusula va **seguida de una condición lógica** o criterio que se evaluará para cada fila de la tabla, y sólo se listarán las filas que cumplan dicha condición. **`WHERE`** aparecerá después de la cláusula **`FROM`**.

```
SELECT <lista_de_expresiones>

  FROM <tabla>

  WHERE <criterio>
```

Por **ejemplo**, si queremos obtener los nombres de los empleados del departamento "02" podemos ejecutar la sentencia:

```
SELECT nombre

  FROM empleado

  WHERE cddep='02'
```

Para construir el criterio disponemos de una gran variedad de operadores lógicos, a continuación los veremos y practicaremos.

- **Operadores de comparación**: **`<, <=, =, >=, >, <>`**. Con el significado habitual.
  Por ejemplo, empleados que han trabajado 30 o más horas:
  ```
  SELECT cdemp
     FROM trabaja
     WHERE nhoras >=30
  ```
- **Operador de rango**: **`BETWEEN ... AND`**. Expresa un rango de valores en el formato desde-hasta.
  Por ejemplo, empleados que entraron en la empresa en el año 1998:
  ```
  SELECT *
     FROM empleado
     WHERE fecha_ingreso BETWEEN '1998-01-01' AND '1998-12-31'
  ```
- **Operador de pertenencia a un conjunto: `<expresión> IN (<conjunto>)`**. Seleccionará las filas donde la expresión se evalúe como un valor perteneciente al conjunto expresado.
  Por ejemplo, departamentos que están situados en "Almería" o "Málaga":
  ```
  SELECT *
     FROM departamento
     WHERE ciudad IN ('Almería','Málaga')
  ```
- **Operador de correspondencia con un patrón**: **`<expresión> LIKE <patrón>`**. Seleccionará las filas donde la expresión coincida con las reglas establecidas por el patrón. Para construir el patrón se utilizan caracteres comodín, los más usados son: "%" que se sustituye por cualquier secuencia de 0 o más caracteres, y "_" que se sustituye por un único carácter.
  Por ejemplo, empleados cuyo nombre contiene el apellido "Verde":
  ```
  SELECT *
     FROM empleado
     WHERE nombre LIKE '%Verde%'
  ```
- **Condición de valor nulo o no nulo: `<expresión> IS NULL`** y **`<expresión> IS NOT NULL`** respectivamente. Se verán afectadas las filas cuyo valor de expresión sea NULL o no, respectivamente. Por ejemplo, empleados que no tienen jefe asignado:
  ```
  SELECT *
     FROM empleado
     WHERE cdjefe IS NULL
  ```
- **Operadores lógicos para enlazar más de un criterio**: **`AND`** y **`OR`**. Sirven para obtener un resultado lógico combinando el resultado de dos criterios. Tienen el habitual significado del **Y y el O lógicos**.
  Por ejemplo, empleados del departamento "02" o del departamento "04":
  ```
  SELECT *
     FROM empleado
     WHERE cddep='02' OR cddep='04'
  ```
- **Operador de negación lógico: `NOT`**. Niega lógicamente el criterio que sigue a continuación.
  Por ejemplo, empleados que no son del departamento "02":
  ```
  SELECT *
     FROM empleado
     WHERE NOT cddep='02'
  ```

Las expresiones lógicas pueden agruparse con paréntesis para indicar el orden de evaluación o aclarar las expresiones complicadas. Por ejemplo, las expresiones **`cdpro='DAG' OR (cdpro='GRE' AND nhoras>=15)`** y **`(cdpro='DAG' OR cdpro='GRE') AND nhoras>=15`**, no producen el mismo resultado. Prueba a lanzar las sentencias SQL correspondientes y compruébalo.

Cláusula `DISTINCT`

En muchas ocasiones las consultas arrojan resultados repetidos. Por ejemplo, si utilizamos la sentencia **`SELECT`** siguiente para ver

los departamentos donde trabaja algún empleado:

```
SELECT cddep
  FROM empleado
```

Obtendremos departamentos repetidos tantas veces como empleados existan en cada departamento.

La cláusula DISTINCT suprime los resultados repetidos de la consulta, de forma que se muestren sólo los resultados distintos, es decir, cada resultado aparecerá en la consulta una sola vez.

Por tanto, para la consulta anterior se pueden conseguir resultados más correctos utilizando **DISTINCT**:

```
SELECT DISTINCT cddep
  FROM empleado
```

DISTINCT se aplica a la fila completa. Si escribimos:

```
SELECT DISTINCT cddep, cdjefe
  FROM empleado
```

Se obtienen las combinaciones únicas de empleado y jefe.

Cláusula ORDER BY

Una de las características intrínsecas a los SGBDR y SQL es que **el orden de aparición de las filas en una consulta no es predecible**. Esto quiere decir que **en momentos diferentes la misma consulta puede producir el mismo conjunto de filas pero con ordenaciones diferentes**. Esto es así porque el motor de base de datos toma decisiones en función del momento, en aras de la mayor optimización y rendimiento de las operaciones.

Para conseguir un orden de filas dado en función del criterio que deseemos imponer se utiliza la cláusula ORDER BY. En ella se pueden disponer una serie de criterios separados por comas para ordenar los resultados de una consulta. Normalmente esos criterios serán nombres de columna, aunque es posible utilizar expresiones.

Por **ejemplo**, para listar los nombres de los empleados en orden alfabético podemos emplear la siguiente sentencia **SELECT**:

```
SELECT nombre
  FROM empleado
  ORDER BY nombre
```

Para obtener los nombres de empleado **ordenados por el departamento** donde trabajan, y a igualdad de éste, usar como segundo criterio de ordenación el orden alfabético de nombre, se emplearía:

```
SELECT cddep, nombre
  FROM empleado
  ORDER BY cddep, nombre
```

También es posible ordenar referenciando la posición de la columna deseada de la cláusula **SELECT**:

```
SELECT cddep, nombre
  FROM empleado
  ORDER BY 1, 2
```

Se puede decidir si el orden será ascendente (ASC) o descendente (DESC). Si queremos la lista de departamentos ordenada por orden alfabético descendente de la ciudad de localización podemos usar:

```
SELECT *
  FROM departamento
  ORDER BY ciudad DESC
```

El sentido de la ordenación por defecto es ascendente. **Se pueden combinar criterios ascendentes y descendentes**. Por ejemplo, si queremos la lista de departamento por orden ascendente de ciudad y descendente de nombre, se puede emplear.

```
SELECT *
```

```
  FROM departamento

  ORDER BY ciudad ASC, nombre DESC
```

Por supuesto **se puede combinar la cláusula `ORDER BY` con la cláusula `WHERE`**. En este caso `ORDER BY` debe aparecer después de `WHERE`. Por ejemplo, la lista de empleados por orden alfabético del departamento "02" se obtendría con:

```
SELECT *

  FROM empleado

  WHERE cddep='02'

  ORDER BY nombre
```

Cláusula `GROUP BY` y funciones de agregación

En muchos casos se necesita **obtener resultados resumen de los datos contenidos en las filas de una tabla**. Por **ejemplo** calcular la suma de las horas trabajadas por todos los empleados de la empresa, u obtener el número de departamentos de los que consta la empresa.

En estos casos se utilizan las funciones de agregación. Son funciones que realizan cálculos sobre expresiones basadas en los datos de la tabla y resumen en un solo dato (numérico) el resultado. Las funciones de agregación son:

- **Suma `SUM(<expresión>)`** : **Calcula la suma** de los valores de expresión de cada fila. Por **ejemplo**, para calcular la suma de horas trabajadas por todos los empleados se utilizaría:

  ```
  SELECT SUM(nhoras)

    FROM trabaja
  ```

- **Media aritmética `AVG(<expresión>)`**: AVG es la abreviatura de "Average", que significa media. Como te imaginarás, **calcula la media aritmética** de los valores de expresión. Por ejemplo, la media de horas trabajadas por cada empleado se calcula como:

  ```
  SELECT AVG(nhoras)

    FROM trabaja
  ```

- **Valor mínimo `MIN(<expresión>)`**: **Produce como resultadoel mínimo** de los valores del conjunto de expresiones evaluadas. Por **ejemplo**, el mínimo valor de "horas trabajadas" (correspondiente al trabajador que menos ha horas ha trabajado):

  ```
  SELECT MIN(nhoras)

    FROM trabaja
  ```

- **Valor máximo `MAX(<expresión>)`**: **Obtiene el máximo** del conjunto de expresiones. Por **ejemplo**, fecha de ingreso en la empresa más reciente, es decir, la fecha en la que ingresó el último trabajador contratado por la empresa. (Evidentemente se entiende que las fechas más recientes son mayores que las más antiguas):

  ```
  SELECT MAX(fecha_ingreso)

    FROM empleado
  ```

- **Cuenta del número de filas de una consulta `COUNT(*)`**: Calcula el número de filas que se obtienen en una consulta. Por **ejemplo** para saber el número de empleados de la empresa:

  ```
  SELECT COUNT(*)

    FROM empleado
  ```

- **Cuenta del número de filas que no producen el valor NULL `COUNT(<expresión>)`**: Produce como resultado un número que indica cuántas filas de la consulta no producen NULL al evaluar la expresión. Por **ejemplo** para calcular el número de empleados que están asignados a un departamento:

  ```
  SELECT COUNT(cddep)

    FROM empleado
  ```

- **Cuenta del número de filas distintas de una consulta `COUNT(DISTINCT <expresión>)`**: Análogo al anterior pero sin contar las filas repetidas. Por **ejemplo**, si queremos saber en cuántas ciudades diferentes existen departamentos de la empresa:

  ```
  SELECT COUNT(DISTINCT ciudad)
  ```

```
FROM departamento
```

Observaciones:

- **No se pueden combinar funciones de agregación con otro tipo de expresiones**, esto provocaría un error de SQL. Aunque sí se puede combinar varias funciones de agrupamiento en la misma consulta. Si se quiere saber la fecha de ingreso más antigua y más reciente:

```
SELECT MIN(feha_ingreso), MAX(fecha_ingreso)
  FROM empleado
```

- **Es muy útil combinar estas funciones de agregación con la cláusula `GROUP BY`**, de esta forma se pueden calcular subtotales de grupos de filas con alguna característica en común. Por ejemplo, si quisiéramos saber cuál es el número de horas trabajadas en cada proyecto se podría emplear:

```
SELECT cdpro, SUM(nhoras)
  FROM trabaja
  GROUP BY cdpro
```

- **En caso de utilizar `GROUP BY` sí es posible mezclar expresiones simples con funciones de agregación**, siempre que las expresiones simples formen parte de la cláusula `ORDER BY`.
- **También es posible establecer varios niveles de agrupamiento**. Por ejemplo, lista del número de empleados por cada departamento y jefe:

```
SELECT cddep, cdjefe, COUNT(nombre)
  FROM empleado
  GROUP BY cddep,cdjefe
```

Cláusula `HAVING`

Al igual que la cláusula `WHERE` se utiliza para seleccionar filas individuales del resultado de una consulta, **la cláusula `HAVING` puede ser utilizada para seleccionar grupos de filas**. El formato de la cláusula `HAVING` es análogo al de la cláusula `WHERE`, consistiendo en la palabra clave `HAVING` seguida del criterio lógico de selección. Por **ejemplo**, si se quisiera obtener un listado de proyectos con número total de horas trabajadas, salvo aquellos en los que no se haya trabajado ninguna hora, se podría escribir en SQL:

```
SELECT cdpro, SUM(nhoras)
 FROM trabaja
 GROUP BY cdpro
 HAVING SUM(nhoras)>0
```

Es fácil confundir `WHERE` y `HAVING`. ¿Cuándo utilizar uno y otro? Recuerda:

- **`WHERE` para seleccionar filas individuales de la cláusula `FROM`,**
- **`HAVING` para seleccionar filas de agrupamiento de la cláusula `GROUP BY`.**

Veamos un **ejemplo** combinado. Si se desea seleccionar a los empleados cuyo jefe es el empleado "A11", agruparlos por departamento, y obtener la lista de aquellos departamentos en los que haya más de un empleado, se puede escribir:

```
SELECT cddep, COUNT(nombre)
FROM empleado
WHERE cdjefe='A11'
GROUP BY cddep
HAVING COUNT(nombre)>1
```

Por supuesto se puede combinar lo anterior con la ordenación por medio de la cláusula `ORDER BY`.

```
SELECT cddep, COUNT(nombre)
FROM empleado
WHERE cdjefe='A11'
```

```
GROUP BY cddep

HAVING COUNT(nombre)>1

ORDER BY cddep
```

Cláusula LIMIT

En ocasiones resulta muy necesario poder limitar el número de filas que devuelve una consulta. Esto se consigue con la cláusula **LIMIT**. La sintaxis de esta cláusula es la siguiente:

```
LIMIT <desplazamiento>,<número de filas>
```

Limita la sentencia **SELECT** a las primeras **<número de filas>** contadas a partir de la fila **<desplazamiento>**.

Por **ejemplo**, para listar los tres empleados más antiguos de la empresa:

```
SELECT nombre,fecha_ingreso

 FROM empleado

 ORDER BY fecha_ingreso ASC

 LIMIT 3
```

Otro **ejemplo** puede ser conocer los dos empleados que más horas han trabajado en proyectos:

```
SELECT cdemp,SUM(nhoras)

 FROM trabaja

 GROUP BY cdemp

 ORDER BY SUM(nhoras) DESC

 LIMIT 2
```

Subconsultas

En muchas ocasiones el **criterio lógico** para seleccionar las filas de una consulta en la cláusula **WHERE** viene dado por los resultados de otra consulta previa**. A esta consulta previa se la denomina subconsultao consulta anidada. Es una característica muy importante del lenguaje SQL**, hasta el punto de que muchas consultas no podrían ser realizadas sin el uso de esta capacidad.

- **Una sentencia SQL con una subconsulta es frecuentemente el modo más natural de expresar una consulta**, ya que se asemeja mucho a la descripción de la consulta en lenguaje natural.
- **Las subconsultas hacen más fácil la escritura de sentencias SELECT**, ya que permiten "descomponer una consulta en partes" (la consulta y sus subconsultas) y luego recomponerlas.

Una subconsulta es una consulta que aparece dentro de la cláusula WHERE o HAVING de otra sentencia SQL. Por **ejemplo**, si analizamos el enunciado "lista de empleados que han trabajado más horas de la media de horas trabajadas de todos los empleados en proyectos", nos daremos cuenta que en realidad hay dos consultas en el enunciado, en una de ellas hay que calcular la media de horas trabajadas, y en la otra hay que listar los empleados que superan esa media. En SQL quedaría así:

```
SELECT cdemp,nhoras

 FROM trabaja

 WHERE nhoras > (SELECT AVG(nhoras)FROM trabaja)
```

En la cláusula HAVING las subconsultas sirven para seleccionar los grupos de filas del resultado. Por ejemplo, lista de empleados que han trabajado más horas en proyectos que las horas dedicadas al proyecto "AEE":

```
SELECT cdemp, SUM(nhoras)

 FROM trabaja

 GROUP BY cdemp

 HAVING SUM(nhoras) > (SELECT SUM(nhoras) FROM trabaja WHERE cdpro='AEE')
```

Es necesario hacer una serie de consideraciones relativas a las subconsultas:

- **Una subconsulta debe producir una única columna de datos como resultado**. Esto significa que una subconsulta siempre tiene un único elemento de selección de la cláusula SELECT.

- **La cláusula ORDER BY no tiene sentido en una subconsulta.** Los resultados de la subconsulta se utilizan internamente por parte de la consulta principal y nunca son visibles al exterior, por lo que carece de sentido ordenarlos.
- **Es posible referirse a nombres de columnas de la consulta principal desde dentro de la subconsulta.**

Existe una gama muy rica de operadores para relacionar la consulta principal y la subconsulta. Vamos a verlos a continuación.

- **Operadores de comparación:** <, <=, =, >=, >, <>: Se compara el valor de la expresión WHERE utilizada en la consulta principal con un único valor proporcionado por la subconsulta. Por ejemplo, lista de los departamentos que están en la misma ciudad que el departamento "01":

```
SELECT cddep

 FROM departamento

 WHERE ciudad = (SELECT ciudad FROM departamento WHERE cddep='01')
```

- **Operador de pertenencia a un conjunto** `<expresión> IN (<subconsulta)`: Se **seleccionan las filas** en las que la evaluación de la expresión sea alguno de los valores del conjunto de valores producido por la subconsulta. Por **ejemplo**, lista de empleados que tienen como jefe un empleado del departamento "04":

```
SELECT nombre, cddep, cdjefe

 FROM empleado

 WHERE cdjefe IN ( SELECT cdemp FROM empleado WHERE cddep='04')
```

- **Operador de existencia** `EXIST(<subconsulta>)`: Este operador comprueba si la subconsulta siguiente devuelve alguna fila o no, evaluándose a **VERDADERO o FALSO** respectivamente. Hay que hacer notar que en la subconsulta debe aparecer alguna columna de la consulta principal. Para ello se puede utilizar la notación `<tabla>.<columna>`, de manera que no se produzcan ambigüedades. Por ejemplo, si quisiéramos obtener los nombres de los empleados que trabajan en departamentos situados en Almería, podríamos escribir:

```
SELECT nombre

 FROM empleado

 WHERE EXISTS (SELECT * FROM departamento WHERE ciudad='Almería' AND

 empleado.cddep=departamento.cddep)
```

  Observa cómo hemos incluido una condición en la cláusula `WHERE` de la subconsulta que la relaciona con la consulta principal `empleado.cddep=departamento.cddep`.
- **También es posible utilizar alias para los nombres de tabla en el caso de que la tabla de la consulta principal y de la subconsulta sean la misma, un alias es como un sobrenombre dado a una tabla**. Se pueden definir los alias en la cláusula `FROM`, a continuación del nombre la tabla referenciada. Por ejemplo, si quisiéramos obtener la lista de los empleados que son jefes de algún empleado, deberíamos comprobar para cada empleado de la tabla empleado "sí existe" (`EXISTS`) algún otro empleado de la tabla empleado que tiene como cdjefe el cdemp del empleado que estamos comprobando. Quedaría así:

```
SELECT nombre

 FROM empleado a

 WHERE EXISTS (SELECT * FROM empleado b WHERE a.cdemp=b.cdjefe)
```

- **Operadores de comparación cuantificada** `<operador de comparación>[SOME|ALL]`: Combinan el uso de los operadores de comparación (`<,>,<=,>=,=,<>`) con la cuantificación "alguno" (`SOME`) o "todos" (`ALL`). Veamos un ejemplo de cada caso.
  Lista de los empleados que tienen una antigüedad superior a alguno del departamento "03":

```
SELECT nombre,fecha_ingreso,cdemp

 FROM empleado

 WHERE fecha_ingreso>SOME(SELECT fecha_ingreso FROM empleado WHERE cddep='03')
```

  Lista de empleados y horas trabajadas de los empleados que han trabajado más horas que todos los empleados del proyecto "GRE":

```
SELECT cdemp,nhoras

 FROM trabaja

 WHERE nhoras > ALL (SELECT nhoras FROM trabaja WHERE cdpro='GRE')
```

Consultas combinadas a varias tablas

Hasta ahora hemos estado viendo **sentencias y cláusulas** de SQL que nos permitían obtener información de una sola tabla. Pero seguramente sabes que una base de datos puede contener muchas tablas, y que muchas veces la información de tablas distintas está relacionada.

¿Qué podemos hacer para consultar una información que se encuentre repartida por varias tablas?

Es muy habitual que los datos que queremos obtener por medio de una consulta se encuentren en tablas diferentes. En estos casos hay que combinar las tablas necesarias para construir una única consulta. Existen **dos tipos** de combinaciones de tablas, la unión (`UNION`) y la composición (`JOIN`). Vamos a ver a continuación en qué consiste cada una.

Unión de tablas.

La unión de tablas en una consulta produce como resultado la unión, en el sentido algebraico de la teoría de conjuntos, de las filas de ambas tablas, es decir, produce una nueva tabla que tiene todas las filas de la primera tabla y a continuación también todas las filas de la segunda tabla.

Por supuesto las filas de ambas tablas deben tener el mismo número de columnas y del mismo tipo. También es posible utilizar la unión de consultas `SELECT`.

Por ejemplo, si queremos tener una lista de los empleados que pertenecen al departamento "04", unidos a aquellos cuyo jefe el el empleado "A11":

```
(SELECT * FROM empleado WHERE cddep='04')

  UNION

   (SELECT * FROM empleado WHERE cdjefe='A11')
```

Composición de tablas.

En este tipo de combinación de tablas las filas de una tabla se concatenan a las filas de la otra tabla. **A esto se le llama JOIN de tablas**.

Vamos a explicarlo con un **ejemplo**. Si quisiéramos listar los nombres de los empleados, junto con el nombre del departamento donde trabajan, deberíamos obtener datos de la tabla empleado (nombre del empleado) y de la tabla departamento (nombre de departamento). En SQL quedaría así:

```
SELECT empleado.nombre,departamento.nombre

  FROM empleado, departamento

  WHERE empleado.cddep=departamento.cddep
```

Veamos algunas consideraciones sobre la **composición (JOIN)** de tablas apoyándonos en el ejemplo anterior:

- **Las consultas combinadas suelen relacionar tablas por medio de una columna común que es clave primaria en una tabla y externa en la otra.** En el ejemplo, la columna cddep es clave primaria en la tabla departamento y es clave externa en la tabla empleado.
- **Siempre que exista ambigüedad será necesario utilizar la notación `<tabla>.<columna>` , y en general es una buena costumbre utilizarla en cualquier caso.** En el ejemplo anterior existen las columnas nombre y cddep en ambas tablas de combinación, por lo que es necesario indicar a cuál nos referimos exactamente.
- **Se pueden componer filas de una tabla con filas de esa misma tabla. En este caso deberemos emplear alias de tabla** para no cometer ambigüedades.

Lo anterior es un ejemplo del tipo de composición más sencillo, el producto cartesiano. Consiste en combinar todas las filas de una tabla con todas las filas de la otra, y luego seleccionar sólo las filas en las que coincide el campo clave común, que se usa para hacer el JOIN, y dejar una sola vez esa columna común. Esto es el producto cartesiano en términos de teoría de conjuntos si las filas de cada tabla fuesen los elementos de dos conjuntos, que se complementa con una condición `WHERE` que selecciona qué filas del producto cartesiano formarán el resultado final del `JOIN`.

Piensa que en vez de **tablas de bases de datos** estuvieras manejando esas mismas tablas pero impresas en papel, cada una en una hoja. La forma de encontrar la información que se pide, ¿cuál sería?

- Pues empezaríamos con la primera fila de la primera tabla empleado,
- nos fijaríamos en el nombre del primer empleado, y en el código del departamento en el que trabaja, para a continuación empezar a comparar ese código de departamento con el de todas las filas de la tabla departamentos en la segunda hoja, hasta que encontráramos una coincidencia.
- Entonces nos fijaríamos en el nombre de ese departamento y lo anotaríamos junto al nombre del empleado.
- Y volveríamos a repetir el proceso con todas las filas de la tabla empleado.
- Pues bien, más o menos es lo mismo que hace automáticamente la composición (JOIN)

Existen otros tipos de composición ligeramente diferentes al anterior. Los veremos a continuación.

- **Composición interna `INNER JOIN`**: Básicamente funciona **igual que el producto cartesiano sólo que utiliza índices para**

agilizar el tiempo en obtener el resultado. Empareja las filas de una tabla buscando directamente en las filas de la otra tabla las filas que cumplen la condición, esto lo consigue usando el índice. La sintaxis es la siguiente:

```
SELECT <lista_de_expresiones>

 FROM <tabla1> INNER JOIN <tabla2> ON

 <expresión_composición>
```

En donde **<expresión_composición>** es la combinación por medio de un operador de comparación (**<, >, <=, >= ,=, <>**) de dos columnas procedentes de cada tabla.
Por **ejemplo**, si queremos resolver de forma más eficiente el ejemplo propuesto cuando tratamos la composición por producto cartesiano, listado de nombres de los empleados junto con el nombre del departamento donde trabajan. Podemos escribir:

```
SELECT a.nombre, b.nombre

 FROM empleado a INNER JOIN departamento b ON a.cddep=b.cddep
```

Observa además el uso de alias de tablas, aunque no es necesario en este caso nos ahorra escribir repetidamente el nombre de las tablas.

- **Composiciones externas `LEFT JOIN` y `RIGHT JOIN`: En el caso de las composiciones producto cartesiano e `INNER JOIN` se concatenan filas de dos tablas que constan de una columna y valores comunes. Pero si alguna de las filas de las tablas a componer tienen valores nulos (NULL)** en alguna de sus filas, o el valor que contienen no coincide con ninguno de los de la otra tabla, se pierden esas filas en la composición resultado, **lo cual puede hacer aparecer resultados erróneos.**

Para corregir este comportamiento se utilizan las composiciones `LEFT JOIN` y `RIGHT JOIN`. En ambos casos las filas con valores nulos en las columnas de composición, o con valores no coincidentes en ambas tablas de composición son tenidos en cuenta. `LEFT JOIN` añade al resultado del `INNER JOIN` las filas de la tabla izquierda que no tienen correspondencia en la tabla de la derecha, `RIGHT JOIN` hace lo propio con las filas de la tabla derecha. **Veamos un par de ejemplos**:

```
SELECT a.nombre, b.nombre

 FROM empleado a LEFT JOIN departamento b ON a.cddep=b.cddep
```

El resultado es el `INNER JOIN` de ambas tablas, más los empleados que no están asignados a ningún departamento.

```
SELECT a.nombre, b.nombre

 FROM empleado a RIGHT JOIN departamento b ON a.cddep=b.cddep
```

El resultado es el `INNER JOIN` de ambas tablas más los departamentos donde no hay empleados en este momento.

Para que entiendas mejor las diferencias entre las distintas composiciones estudiadas hemos confeccionado una animación que te aclarará de forma gráfica esas diferencias.

Vistas

__Víctor__ ha visto que hay muchas __consultas__ que habrá que ejecutar de vez en cuando, porque se usarán para comprobar el "estado de situación" de alguna información concreta contenida en la base de datos. En el __ejemplo__ se necesita consultar constantemente la suma de horas trabajadas en cada uno de los proyectos, con la finalidad de hacer un seguimiento del coste que se lleva acumulado. Esta consulta requiere buscar información en dos tablas, de empleados y proyectos, y requiere cierto tiempo escribirla, con la posibilidad añadida de que nos equivoquemos alguna vez y obtengamos un resultado distinto del empleado.

__Víctor__ le plantea a __Carmen__ si no sería conveniente escribir la sentencia en us script SQL, de forma que podamos ejecutarla cada vez que sea necesario sin tener que volver a teclearla.

__Carmen__ le dice que hay una solución todavía mejor: __Crear una vista__, con un nombre, que usaremos como si fuera una tabla de la base de datos que contiene justo la información que deseamos, aunque no exista realmente como tabla en la base de datos.

Nos va a permitir obtener la información que deseamos de forma más simple.

Además nos va a permitir hacer consultas distintas usando la vista creada como una tabla más.

De esta forma, mediante las __vistas__, la base de datos presenta para los usuarios un aspecto más adecuado a las necesidades de cada uno, permitiéndole trabajar a cada uno como si la base de datos tuviera distintas tablas en cada caso, aunque el almacenamiento físico sea único.

Ya has comprobado la versatilidad y potencia del leguaje SQL en su faceta de **manipulación de datos**. También has visto que algunas sentencias son laboriosas de escribir. ¿Es necesario volver a escribirlas cada vez que quiero obtener la misma información? ¿Tendré, por ejemplo, que escribir cada mes la consulta para obtener la lista de los trabajadores que más han trabajado, o la lista de trabajadores asignados a cada proyecto cada vez que haya cambios en los departamentos o en los proyectos? Es un poco tedioso, ¿no?, porque aunque la información obtenida cambie en cada momento según la información que contenga la base de datos, la sentencia SQL necesaria va a ser siempre la misma.

En SQL existe una forma de poder almacenar sentencias SQL para simplificar su manejo: **Son las vistas.**

Una vista no es más que una sentencia SQL almacenada que puede ser lanzada (ejecutada) sin tener que volver a escribirla, y que funcionalmente actúa cómo si fuera una tabla más.

Se puede crear una vista con la sintaxis básica:

```
CREATE VIEW <nombre_vista> (<columnas>) AS <sentencia_select>
```

Para saber más

Existen unas consideraciones sobre las vistas que hay que tener en cuenta para poder utilizarlas correctamente. Visita el siguiente enlace de la documentación oficial de MySQL donde se trata este tema.

Vistas en MySQL

Veamos un **ejemplo**. En la empresa de nuestras prácticas se requiere a menudo una lista de nombres de proyecto y la suma de horas trabajadas en cada uno de ellos. Se podría crear una vista que resumiese esa consulta con un nombre descriptivo como "horas_trabajadas_por_proyecto".

```
CREATE VIEW horas_trabajadas_por_proyecto (proyecto,horas_trabajadas) AS
  SELECT p.nombre,SUM(t.nhoras)
   FROM proyecto p LEFT JOIN trabaja t
   ON p.cdpro=t.cdpro
   GROUP BY p.nombre
```

A partir de ese momento podríamos obtener el listado requerido con sólo escribir:

```
SELECT * FROM horas_trabajadas_por_proyecto
```

Al ser tratada como una tabla se podrían hacer consultas basadas en la vista. Por ejemplo, lista de proyectos con más de 100 horas trabajadas.

```
SELECT * FROM horas_trabajadas_por_proyecto WHERE horas >= 100
```

Para borrar una vista se emplea `DROP VIEW <nombre_vista>`.

Por supuesto las tablas subyacentes no se verán afectadas por el borrado de la vista.

Víctor *le consulta a* ***Carmen*** *ya que hay algunos ejercicios entre los que le ha propuesto que le están costando más trabajo del que deberían y no encuentra la forma de terminarlos correctamente.*

Carmen *mira los intentos que ha realizado, y ve claramente lo que sucede. Víctor No sabe usar las distintas* ***funciones*** *que proporciona SQL, de control de flujo, de manejo de caracteres, numéricas, de fecha y hora, de conversión de tipos... Sin ellas, hay tareas que no se podrían realizar, o que serían extremadamente complicadas. Le da un cariñoso tirón de orejas, y lo pone a repasar esas funciones.*

- "Ya sabes que en esto de aprender a manejar las bases de datos no hay atajos, así que no intentes cogerlos, y dedica a cada cosa el tiempo que requiere",- le comenta ella.

En los apartados anteriores hemos hecho mención a expresiones como combinación de columnas y operadores. **SQL también dispone de una extensa gama de funciones que pueden ser utilizadas en las expresiones**. Típicamente las expresiones constan de un nombre y aceptan unos parámetros entre paréntesis para devolver un valor.

```
<nombre_función>(<parámetro1>,<parámetro2>,...)
```

Aunque hay otros, los grupos fundamentales de funciones son:

- **Funciones de control de flujo**
- **Funciones de caracteres.**
- **Funciones numéricas.**
- **Funciones de fecha y hora.**
- **Funciones de conversión.**

Para saber más

No sería posible estudiarlas todas ya que son varios cientos, por lo que te remitimos al siguiente enlace donde puedes ver la lista completa, y al que deberás acudir a documentarte cada vez que tengas dudas al usar alguna de ellas.

Lista de funciones de MySQL clasificadas por grupos.

A continuación trataremos las más importantes y utilizadas.

- **Funciones de control de flujo.**
  - **IF(<expresión1>,<expresión2>,<expresión3>)** : Devuelve **<expresión2>** o **<expresión3>** en función de si **<expresión1>** es VERDAD o FALSO respectivamente.
    Ejemplo: **SELECT IF(3>5, 'verdad','falso')** Devuelve 'falso'.
  - **IFNULL(<expresión1>,<expresión2>)** : Devuelve **<expresión1>** si **<expresión1>** es distinto de NULL, en otro caso devuelve **<expresión2>**.
    Ejemplo: **SELECT IFNULL(cddep,'Desconocido')** Devuelve 'Desconocido' para los cddep que sean NULL, en los demás quedan como están. Es muy útil en consultas que devuelven valores nulos para cambiarlos por un valor descriptivo.
- **Funciones de caracteres**
  - **CHAR_LENGTH(<cadena>)**: Devuelve un entero que es la longitud de la cadena de caracteres.
    Ejemplo: **SELECT CHAR_LENGTH('Hola')** Devuelve 4.
  - **CONCAT(<cadena1>,<cadena2>,...)** : Concatena las cadenas de caracteres.
    Ejemplo: **SELECT CONCAT('H','o','l','a')** Devuelve 'Hola'.
  - **INSTR(<cadena>,<subcadena>)**: Devuelve la posición de la primera ocurrencia de la subcadena dentro de la cadena.
    Ejemplo: **SELECT INSTR('Hola', 'ol')** Devuelve 2.
  - **LOWER(<cadena>)**: Pasa la cadena a minúsculas.
    Ejemplo: **SELECT LOWER('HOLA')** Devuelve "hola"
  - **UPPER(<cadena>)**: Pasa la cadena a mayúsculas.
    Ejemplo: **SELECT UPPER('hola')** Devuelve "HOLA".
  - **RTRIM(<cadena>)** y **LTRIM(<cadena>)** : Devuelve la cadena sin espacios en blanco a la derecha (**RTRIM**) o izquierda (**LTRIM**).
    Ejemplo **SELECT LTRIM(RTRIM(' Hola '))** Devuelve 'Hola'.
  - **SUBSTR(<cadena>,<posición>,<longitud>)**: Devuelve la subcadena de longitud **<longitud>** a partir de la posición **<posición>** de la cadena **<cadena>**.
    Ejemplo: **SELECT SUBSTR('Hola',2,2)** Devuelve "ol".
- **Funciones numéricas**
  - **ABS(<número>)** : Devuelve el valor absoluto de un número.
    Ejemplo: **SELECT ABS(-21)** Devuelve 21.
  - **CEIL(<número>)**: Devuelve el entero más pequeño no menor que número.
    Ejemplo: **SELECT CEIL(21.9)** Devuelve 22.
  - **MOD(<numero1>,<número2>)**: Devuelve el resto de la división entera entre **<numero1>** y **<numero2>**.
    Ejemplo: **SELECT MOD(18,7)** Devuelve 4.
  - **ROUND(<número>)**: Devuelve el redondeo al entero más cercano.
    Ejemplos: **SELECT ROUND(20.7)** Devuelve 21. **SELECT ROUND(20.3)** Devuelve 20.
  - **TRUNCATE(<número>,<Posiciones_decimales>)**: Devuelve el número con las posiciones decimales dadas sin redondeo.
    Ejemplo: **SELECT TRUNCATE(68.99999,2)** Devuelve 68.99.
- **Funciones de fecha y hora**
  - **ADDDATE(<fecha>,<numero_dias>)**: Devuelve la fecha proporcionada, incrementada en el número de días indicado.
    Ejemplo: **SELECT ADDDATE('2005-04-19',15)** Devuelve '2005-05-04'.
  - **CURDATE()**: Devuelve la fecha actual del sistema.
  - **CURTIME()**: Devuelve la hora actual del sistema.
  - **DATEDIFF(<fecha1>,<fecha2>)**: Devuelve el número de días entre las dos fechas.
    Ejemplo: **SELECT DATEDIFF('2005-04-19','2005-04-2')** Devuelve 17.
  - **DATE_FORMAT(<fecha>,<formato>)**: Devuelve la fecha formateada según el formato especificado.
    Ejemplo: **SELECT DATE_FORMAT('2005-04-19', '%d %m %Y')** Devuelve '19 04 2005' (Consultar la documentación de MySQL para ver los formatos posibles).
  - **DAY(<fecha>)**: Devuelve el día del mes de una fecha.
    Ejemplo: **SELECT DAY('2005-04-19')**. Devuelve 19.
  - **MONTH(<fecha>)**: Devuelve el mes de un fecha.
    Ejemplo: **SELECT MONTH('2005-04-19')**. Devuelve 4.
  - **NOW()**: Devuelve la fecha y hora del sistema.
  - **YEAR(<fecha>)**: Devuelve el año de un fecha.
    Ejemplo: **SELECT YEAR('2005-04-19')**. Devuelve 2005.
- **Funciones de conversión:**
  - **CONVERT(<expresion>,<tipo_dato>)**: Devuelve la expresión convertida al tipo de dato suministrado.
    Ejemplo: **SELECT CONVERT(21,DECIMAL)** Devuelve 21.00.
  - **CAST(<expresión> AS <tipo_dato>)**: Análoga a la anterior.
- **Otras funciones:**
  - **PASSWORD(<cadena>)**: Devuelve la cadena encriptada en el formato de encriptación usado por MySQL.
    Ejemplo: **SELECT PASSWORD('Hola')** Devuelve '*70E61FCD7F1FC3F6D951F0A36510223608C1141A'. Esta función es útil para encriptar contraseñas de nuestras aplicaciones. La encriptación es en un solo sentido.
  - **MD5(<cadena>)**: Análoga a la anterior utilizando el algoritmo de encriptación MD5.

__Víctor__ ha aprendido mucho y rápido. Ya hay pocas cosas que __Carmen__ le pida hacer con la base de datos y que no pueda hacerlas casi de inmediato. En el fondo es fácil, y no es más que cuestión de práctica. __Carmen__ y __Víctor__ saben que seguir __practicando__ un poco es fundamental para él, y por eso le proporciona una batería de ejercicios para que los haga. Dispone de las soluciones, que se las ha

*facilitado **María,** ya que todo son consultas u operaciones que ella ya había realizado para su aplicación de gestión de proyectos.*

*No obstante, **Carmen** no piensa darle a **Víctor** las soluciones hasta que haya intentado resolver todos los ejercicios por sí mismo.*

*Si termina todos los ejercicios correctamente antes de que acabe la semana, y se los entrega, podrán pasar un fin de semana tranquilo, e ir de excursión por ahí. De lo contrario, **Víctor** tendrá que quedarse repasando, porque es imprescindible que avance a buen ritmo en su formación. **José** realmente necesita que pueda empezar a ayudarle en el menor tiempo posible. Y lo primero es lo primero...*

A continuación tienes enunciados que se resuelven como sentencias SQL. Puedes practicarlos con MySQL y la base de datos utilizada para todos los ejemplos de esta unidad. Al final encontrarás una animación con todas las soluciones, pero es muy conveniente que no mires la solución hasta que no hayas resuelto el ejercicio por ti mismo. La solución está sólo para comprobar si te equivocaste a posteriori, y por si te surgen dudas insalvables.

No olvides que para sincronizar tu trabajo con los ejercicios propuesto tienes que:

1. **Crear la base de datos gestionproyectos a partir del esquema de tablas definido en el apartado 6.2 de esta unidad.**
2. **Añadir filas a las tablas según lo expuesto en el apartado 7.4 de esta unidad. Esto es importante para que puedas comprobar los resultados.**

En el siguiente enlace encontrarás los enunciados de los ejercicios, que te recomendamos que imprimas para tenerlos delante mientras vas intentando solucionarlos:

Enunciados

Y seguidamente, el enlace a la presentación con las soluciones a esos ejercicios.

Soluciones

*Finalmente **Víctor** está muy contento, porque ha terminado los ejercicios que le propuso **Carmen** a tiempo, y para celebrarlo, este fin de semana se van a ir a comer fuera, y a darse una vuelta por el parque natural de Cabo de Gata, que en esta época está precioso, y sin el agobio de gente del verano.*

*No obstante, **Carmen** le dice que para el lunes sigue teniendo aún una tarea pendiente: Repasar algunas sentencias SQL de Control (DCL) para asignar y denegar privilegios de acceso a usuarios, y para saber definir transacciones.*

***Víctor:** "¿Son muchas sentencias?"*

***Carmen:** No demasiadas. "**GRANT** y **REVOKE** para asignar y denegar permisos y privilegios a los usuarios y **STARTTRANSACTION**, **COMMIT** y **ROLLBACK** para definir y usar transacciones.*

***Víctor:**"Seguro que está chupado, con lo bien que tú lo explicas. Me lo cuentas el Domingo mientras te invito a comer, ¿vale?"*

***Carmen:**"Cuenta con ello".*

Ya sabes que en los SGBD existe la figura del **Administrador de Base de Datos. Es una persona que se encarga de gestionar todo lo relativo a organización y seguridad del SGBD para dar servicio al resto de usuarios.**

El administrador del SGBD dispone para llevar a cabo su tarea de un conjunto de sentencias SQL muy específicas que gestionan todo lo relativo a usuarios y permisos. Las dos sentencias más importantes son **GRANT** para conceder permisos y **REVOKE** para quitarlos.

La sintaxis de **GRANT** en la siguiente:

```
GRANT <tipo_privilegio> ON <base_de_datos>.<tabla> TO <usuario> [IDENTIFIED BY '<password>']
```

Por ejemplo para crear y autorizar al usuario 'espeamarillo' con el privilegio de poder hacer consultas sobre la tabla empleado de la base de datos gestionproyectos, identificándose con el password 'x3y4z5' escribiríamos:

```
GRANT select ON gestionproyectos.empleado TO espeamarillo IDENTIFIED BY 'x3y4z5'
```

Observaciones:

- **Si el usuario no existe se crea en ese momento** y se le asigna el password dado.
- **Para poder conceder (GRANT) permisos hay que ser el propietario de la tabla o tener el permiso adecuado sobre ella.**
- **Se pueden utilizar comodines para indicar las bases de datos y las tablas**. Por ejemplo para referirnos a todas las tablas de la base de datos gestionproyectos podemos abreviar como "gestionproyectos.*". Para referirnos a todas las tablas de todas las bases de datos se puede escribir "*.*".
- **Si queremos conceder todos los permisos posibles podemos utilizar ALL como tipo de privilegio.**

Para **quitar permisos** concedidos a un usuario utilizaremos la sentencia **REVOKE**.

```
REVOKE <tipo_privilegio> ON <base_de_datos>.<tabla> FROM <usuario>
```

Por ejemplo, si queremos quitar el permiso de inserción de filas a todas las tablas de la base de datos gestionproyectos, al usuario 'espeamarillo', escribiremos:

```
REVOKE insert ON gestionproyectos.* TO espeamarillo
```

La siguiente animación te ayudará a entenderlo mejor.

Ejemplo de uso de GRANT y REVOKE

Para saber más

Es conveniente que profundices en los detalles sobre las sentencias `GRANT` y `REVOKE`. Su uso es bastante específico del sistema concreto que usemos. En el caso particular de MySQL puedes consultar la documentación en línea.

Documentación en línea de la sentencia GRANT en MySQL [Versión en caché]

Existen ocasiones en las que interesa agrupar una serie de sentencias SQL para que se lleven a cabo como un todo, de forma que se ejecuten todas, o no se ejecute ninguna.

Por **ejemplo**, en una base de datos que gestione el sistema de cuentas corrientes de un banco, una transferencia de dinero de una cuenta a otra implica realizar dos sentencias `UPDATE`, en una de ellas restaremos dinero del saldo de la cuenta ordenante y en la otra sumaremos el importe de la transferencia al saldo de la cuenta receptora. Es obvio que para que la transferencia de dinero sea efectiva deben llevarse a cabo las dos, y si ocurre algún error poder restaurar los saldos a la situación de partida.

A esa agrupación de sentencias SQL se le llama transacción, y cuando termina es posible grabarla de forma efectiva (`COMMIT`) o deshacerla (`ROLLBACK`).

Veamos cómo se lleva esto a la práctica con MySQL, nuestro sistema de pruebas.

Para iniciar una transacción se utiliza `START TRANSACTION`. A continuación podemos iniciar la secuencia de sentencias SQL para terminar grabando con `COMMIT` o deshaciendo con `ROLLBACK`.

A continuación tienes una animación que te ayudará a ver de forma práctica el manejo de transacciones.

Ejemplo de uso de sentencias SQL para manejo de transacciones