Gestión de datos

Autores: Guerau Fernandez Isern, Joan Colomer Vila, Maria Begoña Hernández Olasagarre, Enrique Blanco García
El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Josep Jorba Esteve
PID_00298303
Primera edición: septiembre 2023

Introducción

Objetivos

1. Bases de datos relacionales

1.1. Introducción

1.2. El modelo entidad-relación

1.3. El lenguaje SQL y el SGBD MySQL

1.4. Empezar a trabajar con MySQL

1.5. Creación de tablas y restricciones

1.6. Insertar y manipular datos en las tablas

1.7. Consultas básicas en las tablas

1.8. Consultas básicas: filtros y condiciones

1.9. Consultas avanzadas: agrupaciones

1.10. Consultas avanzadas: consultas multitabla

1.11. Consultas avanzadas: subconsultas

1.12. Otras utilidades de MySQL

1.13. Copias de seguridad y restauración de BBDD

1.14. Ejemplo práctico: gestión del catálogo de genes humanos

1.15. *Triggers*, procedimientos y funciones

2. Bases de datos NoSQL

2.1. Introducción

2.2. Ficheros JSON

2.3. El SGDB MongoDB

2.4. Empezar a trabajar con el SGBD MongoDB

2.5. Insertar documentos

2.6. Buscar documentos

2.7. Modificar documentos

2.8. Eliminar documentos

2.9. Importar ficheros JSON a MongoDB

2.10. Buscar en un *array* de documentos

2.11. Agregaciones en MongoDB

Resumen

Actividades

Ejercicios de autoevaluación

Solucionario

Bibliografía

Introducción

Los ficheros son las unidades lógicas de información persistente dentro de un ordenador. Como carecen de estructura propia, es el programador quien establece cómo organizar la información en su interior (por ejemplo, ficheros FASTA, XML o JSON).

Mediante los comandos apropiados del terminal o línea de comandos, podemos analizar el contenido de los ficheros de un modo relativamente eficiente y cómodo. En el módulo «Introducción a los entornos de trabajo GNU/Linux» hemos trabajado con comandos del terminal para gestionar la información almacenada en ficheros de texto que contienen anotaciones biológicas. Sin embargo, cuando el volumen de información excede ciertos límites, como es el caso de la anotación completa de un genoma, y se incrementa el número de personas involucradas en un proyecto de investigación, es necesario organizar y estructurar la información en una base de datos y gestionarla utilizando un programa especializado, también llamado Sistema de Gestión de Bases de Datos (SGBD).

Un SGBD es la herramienta idónea para administrar eficientemente elevadas cantidades de registros o datos. La responsabilidad sobre la gestión y los formatos internos de los datos corresponde al sistema, liberando al propio usuario de estas tareas.

Con un sistema de la gestión de la información tendremos a nuestra disposición un conjunto de herramientas e instrucciones que nos permitirán extraer nuevo conocimiento de toda esta información.

En este módulo veremos dos modelos de gestión de los datos, el modelo relacional basado en el lenguaje SQL y utilizaremos el SGBD MySQL, y el modelo no relacional, también llamado NoSQL, y utilizaremos el SGBD MongoDB basado en colecciones de documentos.

Objetivos

1. Conocer el modelo entidad-relación para diseñar bases de datos.
2. Dominar la conversión de entidades y relaciones en tablas con atributos.
3. Crear y administrar una base de datos con MySQL.
4. Entender las características básicas del lenguaje SQL.
5. Crear y poblar con registros reales las tablas SQL.
6. Realizar consultas SQL a una base de datos relacional.
7. Conocer la utilidad de los subprogramas almacenados, procedimientos, funciones y disparadores (Triggers).
8. Iniciarse con los ficheros JSON y el SGBD NoSQL MongoDB.
9. Administrar un sistema de gestión de base de datos.

1. Bases de datos relacionales

1.1. Introducción

El modelo relacional es un modelo de datos basado en la lógica de predicados y en la teoría de conjuntos. Su idea fundamental es el uso de relaciones. Estas relaciones podrían considerarse en forma lógica, como conjuntos de datos llamados *tuplas*. Pensamos cada relación como si fuese una tabla que está compuesta por registros: cada fila de la tabla sería un registro o *tupla*, y columnas, también llamadas *campos*.

Entre los paradigmas actuales de bases de datos, el modelo relacional está muy extendido y se adapta a la mayoría de los entornos bioinformáticos por su eficiencia y simplicidad.

Otra ventaja de este paradigma es que existen numerosas implementaciones *open source* que proporcionan los servicios completos de un sistema gestor de base de datos relacional con diferentes interfaces gráficas de usuario.

Formalmente, el paradigma relacional está dividido en tres componentes básicos:

- Las tablas y las relaciones entre estas estructuran los datos.
- El álgebra relacional opera sobre la información.
- Un conjunto de axiomas mantiene la integridad del sistema.

Una tabla modela un elemento del mundo real, caracterizando sus atributos. Una relación entre dos tablas emula las asociaciones lógicas existentes entre dos elementos de distintas clases en la realidad, permitiendo el acceso cruzado de información.

Para un universo de datos en particular, la organización de las tablas y las relaciones que lo conforman reciben el nombre de esquema relacional. Una vez definida esta estructura, debe crearse una base de datos para ser poblada con los datos reales (conocidos como instancias o registros), siendo administrada desde ese momento por un sistema de gestión de bases de datos.

Utilizando el álgebra relacional, el usuario puede realizar consultas para extraer nueva información y actualizarla.

Para realizar un diseño eficiente de la base de datos debemos seguir estas reglas:

- Reunir todas las clases de información que deseamos guardar.
- Estructurar de forma lógica la información en diferentes categorías.
- Definir los atributos que caracterizan cada categoría.
- Asignar identificadores suficientemente descriptivos a los atributos.
- Decidir el tipo de datos asociado a cada atributo.
- Descomponer cada pieza de información en la unidad más elemental.
- Seleccionar los atributos que identifican de forma única cada categoría.
- Identificar las relaciones entre categorías.

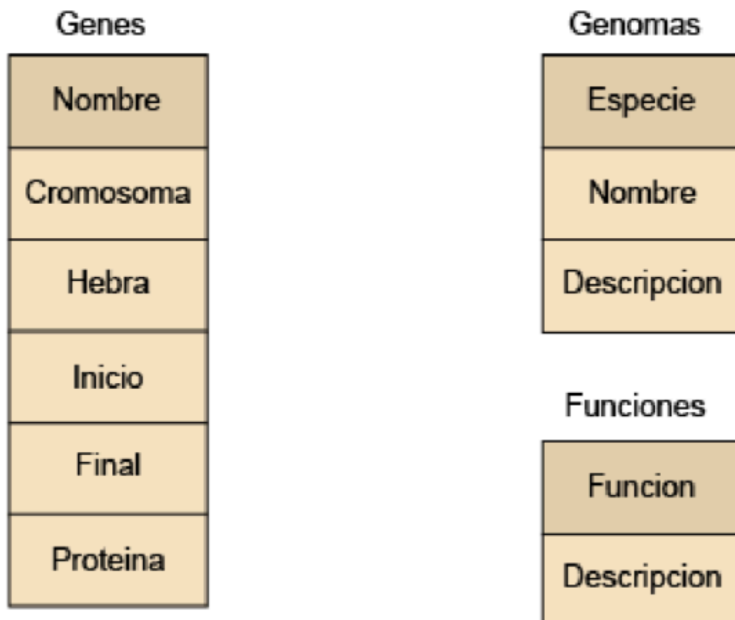
1. Bases de datos relacionales

1.2. El modelo entidad-relación

Podemos estructurar cualquier realidad en diferentes entidades que pueden interactuar entre ellas siguiendo determinadas reglas. Por ejemplo, el genoma, en tanto que parte de la realidad biológica de una célula, también puede estructurarse en distintos componentes. A partir de esta organización artificialmente construida, podemos modelar la totalidad de los elementos que lo conforman utilizando entidades y relaciones. Estas estructuras, contenedores de información o datos pueden ser digitalizados en un ordenador para su gestión y análisis; en el caso del genoma, análisis bioinformáticos.

El modelo entidad-relación nos ayuda a diseñar nuestra propia base de datos. Una entidad representa una clase de elementos en el entorno real que deseamos modelar. Una ocurrencia es una instancia o ejemplo particular de una entidad. La conectividad o participación entre entidades debe especificarse explícitamente mediante relaciones. El número de ocurrencias de una entidad que podrán relacionarse con instancias de otra entidad será:

- Uno a uno (1:1): un elemento de la primera entidad puede relacionarse con un único elemento de la segunda.
- Uno a varios (1:N): un elemento de la primera entidad puede relacionarse con varios elementos de la segunda (pero no al contrario).
- Varios con varios (M:N): un elemento de la primera entidad puede relacionarse con varios elementos de la segunda (y viceversa).



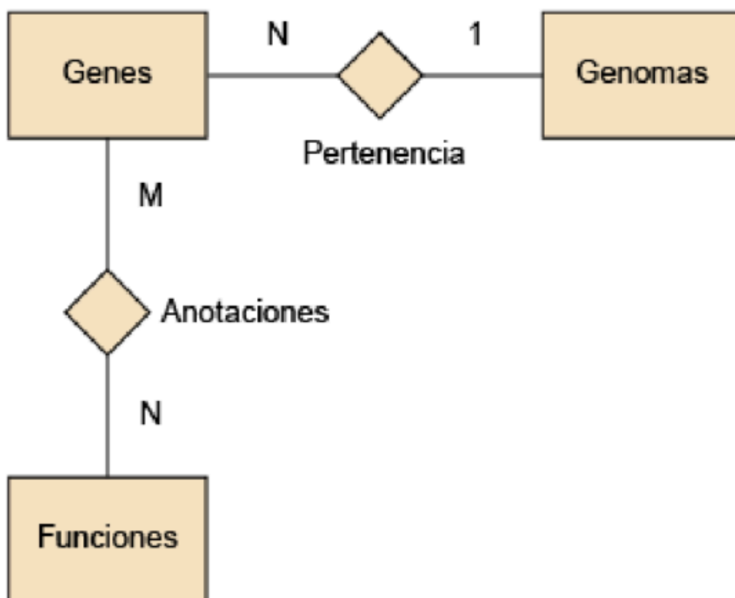


Figura 1. Modelo entidad-relación de un catálogo de genes.
Fuente: elaboración propia.

El modelo contiene tres entidades y dos relaciones. Gráficamente, las entidades se representan utilizando rectángulos, y las relaciones, mediante líneas rectas con un rombo para indicar la conectividad.

Este esquema basado en entidades y relaciones resulta sencillo de emplear posteriormente para construir la base de datos definitiva. Para mostrar las diferentes etapas de diseño, procederemos a modelar un escenario muy habitual en entornos de investigación bioinformáticos: la caracterización del catálogo de genes de un genoma. Podéis observar las diferentes entidades con sus atributos y las relaciones entre entidades que van a formar nuestro modelo en la figura 1.

Los genes son fragmentos de ADN ubicados en una localización precisa del genoma que codifican la secuencia de una proteína. Estas moléculas, por otro lado, desempeñan una función biológica específica dentro del organismo (*). Lógicamente, cada genoma posee su propio catálogo de genes. Analizando esta información previa, decidimos modelar este entorno utilizando las entidades **genes**, **genomas** y **funciones**, con sus propios atributos (mostrados en la figura 2).

Lógicamente, las entidades no son objetos aislados de su entorno. Debemos, por tanto, cumpliendo las especificaciones de nuestro problema, unir mediante relaciones aquellas entidades que están interconectadas en el mundo real. En este caso, los genes poseen la capacidad de pertenecer a un genoma para desarrollar una función concreta en el organismo. Para satisfacer ambas propiedades, definimos las relaciones binarias **pertenencia** y **anotaciones**, cada una con una conectividad diferente (figura

1). Las entidades y relaciones que forman este modelo deben ser convertidas en tablas. Las entidades y sus atributos pasarán a ser tablas de nuestra base de datos, pero no todas las relaciones van a ser tablas, va a depender del tipo de conectividad. Es básico establecer para cada tabla un atributo especial (o una combinación de ellos) que identifique cada instancia de forma unívoca. Este atributo especial recibe el nombre de clave primaria. Es preferible usar un código característico en lugar de un nombre para facilitar la identificación de cualquier instancia mediante la clave primaria (por ejemplo, un código numérico asignado en función del orden de entrada en la tabla).

```
GENES (nombre,cromosoma,hebra,inicio,final,proteina)
GENOMAS (especie,nombre,descripcion)
FUNCIONES (funcion,descripcion)
```

Figura 2. Entidades convertidas en tablas.
Hemos subrayado la clave primaria de cada tabla.
Fuente: elaboración propia.

Las dos relaciones existentes en nuestro esquema deben modelarse de distinta forma, debido a que cada una presenta una combinación diferente de posibles ocurrencias entre tablas. La asociación *pertenencia* entre las tablas *genes* y *genomas* debe representarse con una relación con cardinalidad 1:N, pues un gen solo pertenece a un genoma, pero un genoma contiene muchos genes. Esta relación no necesita una tabla nueva, puede implementarse referenciando simplemente desde una tabla (genes), que es la parte *n* de la relación *pertenencia*, la clave primaria de la otra (genomas) que es la parte 1 de la relación *pertenencia*. Dentro de la tabla *genes*, este atributo recibe la denominación de clave foránea.

```
GENES (nombre,cromosoma,hebra,inicio,final,proteina,especie)
GENOMAS (especie,nombre,descripcion)
```

Figura 3. Relaciones (1:N) convertidas en claves foráneas.
Indicamos con un subrayado superior la clave foránea de cada tabla.
Fuente: elaboración propia.

La relación *anotaciones* entre las tablas *genes* y *funciones* tiene una conectividad M:N, pues un gen puede poseer varias anotaciones, pero una anotación también puede ser compartida por varios genes. Para su correcta implementación, introduciremos una nueva tabla llamada *anotaciones*. Esta poseerá, como clave primaria la combinación de las claves primarias de cada tabla original.

Dado que ambas claves primarias por separado poseen todas las propiedades necesarias, el diseñador garantiza con esta medida que cada instancia de esta nueva tabla estará dotada de un identificador único, que estará formado por el nombre del gen junto con el nombre de la función en particular para que sea un identificador de la instancia único que no se pueda repetir.

```
GENES (nombre,cromosoma,hebra,inicio,final,proteina,especie)
FUNCIONES (funcion,descripcion)
ANOTACIONES (nombre,funcion)
```

Figura 4. Relaciones (M:N) convertidas en tablas.
Fuente: elaboración propia.

Las relaciones con cardinalidad M:N exportadas desde el modelo entidad-relación están caracterizadas también por sus propios atributos. Así, es posible añadir un campo para guardar el origen de cada anotación (por ejemplo, computacional, experimental o fuente bibliográfica). Podemos extraer esta información a partir de los datos recuperados del sistema de anotación automática utilizado para poblar de ejemplos nuestro catálogo de genes:

```
ANOTACIONES (nombre,funcion,origen)
```

Figura 5. Atributos en relaciones convertidas en tablas.
Fuente: elaboración propia.

La selección de las claves idóneas resulta esencial dentro del diseño de una base de datos relacional. De hecho, la integridad de un modelo relacional debe cumplir dos requisitos fundamentales relacionados con la gestión de estas:

1. La clave primaria no debe contener un valor indefinido o nulo y el valor ha de ser único.

2. La clave foránea debe hacer referencia a una clave primaria de otra tabla.

1. Bases de datos relacionales

1.3. El lenguaje SQL y el SGBD MySQL

SQL (en inglés, *Structured Query Language*, 'lenguaje de consultas estructuradas') es el lenguaje de acceso a las bases de datos relacionales más extendido. Con este sistema, el cliente especifica las instrucciones para crear, dotar de contenido, modificar o eliminar las tablas de la base de datos, y realizar las consultas.

Para trabajar con el lenguaje SQL es necesario disponer de un sistema gestor de bases de datos (SGBD), una aplicación informática normalmente basada en el modelo cliente-servidor para administrar bases de datos relacionales.

Existen diferentes SGBD, como Oracle, PostgreSQ... En este módulo usaremos el SGBD MySQL. En Linux, el servidor MySQL funciona en segundo plano, sin interferir en la planificación de procesos del sistema. De este modo, cuando el usuario desea utilizar la base de datos, debe conectarse con la aplicación gestora mediante un programa cliente, a través de un entorno gráfico o desde el propio terminal con el intérprete de comandos de SQL, denominado **mysql**.

SQL fue comercializado por IBM en 1981. MySQL es un gestor distribuido por Oracle bajo licencia GNU o comercial.

1. Bases de datos relacionales

1.4. Empezar a trabajar con MySQL

En la máquina virtual que os proporcionamos hay instalado un SGBD MySQL. Al iniciar la máquina virtual también se inicia el servidor MySQL.

El sistema cliente-servidor permite acceder a un único servidor desde varios clientes. Por defecto, el cliente del sistema es un terminal o línea de comandos, pero se puede acceder también al servidor desde un cliente con interfaz gráfica de usuario (GUI).

En la máquina virtual está instalada la GUI **MySQL Workbench**, pero es posible acceder al servidor MySQL desde muchos clientes con diferentes GUI.

Para empezar a trabajar con MySQL utilizaremos el terminal de Linux ejecutando el comando mysql.

Una vez establecida la conexión al servidor, el programa cliente permanece siempre a la espera de la introducción de un nuevo comando SQL por parte del usuario.

Es importante ser cuidadoso con la sintaxis de los comandos, finalizando cada instrucción con el símbolo «;».

Para ilustrar el funcionamiento del juego de instrucciones de SQL mostrado en la tabla 1 sobre el modelo relacional anterior (ver figura 4), implementaremos una base de datos que denominaremos **catalogo**.

Tabla 1. Manual de referencia de comandos de MySQL

Comando	Descripción
`CREATE USER`	Dar de alta a un nuevo usuario
`DROP USER`	Dar de baja a un usuario existente
`ALTER USER`	Modificar la cuenta de un usuario
`GRANT`	Autorizar a un usuario sobre una base de datos
`REVOKE`	Revocar las autorizaciones de un usuario
`SHOW GRANTS`	Mostrar las autorizaciones de un usuario
`CREATE DATABASE`	Crear una nueva base de datos
`DROP DATABASE`	Eliminar una base de datos existente
`USE DATABASE`	Acceder a una base de datos existente
`SHOW DATABASES`	Mostrar la lista de las bases de datos
`CREATE TABLE`	Crear una nueva tabla
`DROP TABLE`	Eliminar una tabla existente
`SHOW TABLES`	Mostrar la lista de las tablas
`DESCRIBE`	Mostrar los atributos de una tabla
`LOAD DATA`	Poblar una tabla con un fichero de registros
`INSERT`	Poblar una tabla con un registro
`UPDATE`	Actualizar un registro de la tabla
`DELETE`	Borrar un registro de la tabla
`SELECT`	Realizar una consulta sobre una o más tablas

`Help`	Mostrar ayuda sobre un comando del sistema
`Pager`	Mostrar listados página a página con otro programa
`Source`	Ejecutar un *script* de comandos de SQL
`Status`	Mostrar información sobre el estado del sistema
`System`	Ejecutar un comando del terminal
`Warnings`	Mostrar avisos del sistema
`Quit`	Salir del gestor MySQL
`Exit`	Salir del gestor MySQL
`Mysql`	Comando del terminal para invocar al gestor MySQL
`Mysqldump`	Comando del terminal para el *backup* de una base de datos

Fuente: elaboración propia.

El SGBD MySQL permite gestionar múltiples bases de datos, implementando un sistema de seguridad basado en autorizaciones. Una vez instalado el servidor en nuestro entorno Linux, se crea inicialmente un usuario con el rol de administrador (*) (en inglés, *root*) con todos los permisos. Por regla general, solo el usuario con dicho perfil posee los permisos suficientes para gestionar el conjunto de usuarios y bases de datos en su totalidad. De este modo, el administrador garantizará el acceso a una determinada base de datos exclusivamente a un grupo de usuarios establecido previamente. Pese a que puede optarse por trabajar directamente como administrador, siempre es recomendable que configuremos una nueva cuenta a nuestro nombre como usuario convencional para autorizarle posteriormente a trabajar con una base de datos en concreto.

En definitiva, el protocolo que debemos seguir para llevar a cabo nuestro trabajo con el gestor de bases de datos consiste en dos fases:

1. Invocamos al intérprete de SQL como administrador, realizamos las siguientes tareas y abandonamos el programa:
    1. Creamos un nuevo usuario a nuestro nombre.
    2. Creamos una nueva base de datos.
    3. Autorizamos al nuevo usuario a trabajar con esa base de datos.
2. Accedemos al intérprete de SQL con nuestro propio usuario y procedemos a interactuar con la nueva base de datos:
    1. Especificamos que vamos a trabajar con dicha base de datos.
    2. Creamos nuevas tablas (vacías) dentro de la base de datos.
    3. Insertamos nuevos registros en dichas tablas. También podemos modificarlos y eliminarlos.
    4. Realizamos consultas sobre los registros de las tablas para obtener la información deseada.

Para empezar a trabajar primero debemos ejecutar el programa **mysql** desde nuestro terminal de Linux empleando el usuario *root*:

```
% mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or g.
Your MySQL connection id is 5
Server version: 5.7.17-0ubuntu0.16.04.1 (Ubuntu)
Copyright (c) 2000, 2016, Oracle and/or its affiliates. All
rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or
its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or 'h' for help. Type 'c' to clear the current
input statement.

mysql>
```

Figura 6. Iniciar el gestor MySQL como administrador.
Fuente: elaboración propia.

El comando CREATE USER permite al usuario ***root*** dar de alta a un nuevo usuario en nuestro sistema. Es recomendable asignar una contraseña a cada nuevo usuario de nuestro sistema:

```
CREATE USER usuario

IDENTIFIED BY password;
```

Figura 7. Sintaxis del comando CREATE USER.
Fuente: elaboración propia.

Ahora, actuando como administradores, crearemos un nuevo usuario llamado **eblanco**. Para indicar que este usuario trabajará localmente desde nuestra máquina, que actúa de servidor, emplearemos el término **localhost**.

En algunos ejemplos, para que sean más compresibles, vamos a estructurar las instrucciones de SQL en varias líneas. Nótese que tras introducir cada línea del comando presionando la tecla «Enter», el intérprete de MySQL insertará los símbolos -> para denotar que aún no hemos terminado de introducir el comando completo. MySQL no procederá a ejecutar el comando hasta reconocer el carácter «;».

Os animamos a reproducir en vuestro ordenador los comandos de SQL presentados en esta unidad.

```
mysql> CREATE USER 'eblanco'@'localhost'
    -> IDENTIFIED BY '123456';

Query OK, 0 rows affected (0,35 sec)
```

Figura 8. Crear un nuevo usuario con el administrador.
Fuente: elaboración propia.

Existen varios comandos para gestionar el conjunto de bases de datos del sistema (consultad tabla 1). En este momento debemos proceder a crear la base de datos donde trabajará el nuevo usuario con el comando `CREATE DATABASE`:

```
CREATE DATABASE basededatos;
```

Figura 9. Sintaxis del comando `CREATE DATABASE`.
Fuente: elaboración propia.

A continuación, creamos la base de datos **catalogo** y posteriormente empleamos el comando `SHOW DATABASES` para obtener el listado de bases de datos existentes. Podemos comprobar que la nueva base de datos ha sido creada correctamente:

```
mysql> CREATE DATABASE catalogo;

Query OK, 1 row affected (0,00 sec)

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| catalogo           |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0,10 sec)
```

Figura 10. Creación de la base de datos *catalogo*.
Al finalizar la ejecución de un comando, el intérprete muestra por pantalla el número de elementos de los resultados (en inglés, *rows*).
Fuente: elaboración propia.

El administrador concede permisos sobre las operaciones que un determinado grupo de usuarios puede realizar sobre la base de datos. El comando GRANT permite autorizar el acceso de un usuario a una base de datos en particular:

```
GRANT operaciones ON basededatos

TO usuario IDENTIFIED BY password;
```

Figura 11. Sintaxis del comando `GRANT`.
Fuente: elaboración propia.

Para cerrar el trabajo del administrador, debemos garantizar el acceso a la nueva base de datos **catalogo** a nuestro usuario **eblanco** empleando el comando `GRANT`. Con la cláusula **ALL**, el administrador concede el conjunto completo de permisos a este usuario, aunque exclusivamente sobre esta base de datos. Si queremos eliminar privilegios a un usuario debemos utilizar el comando `REVOKE`.

Finalmente, para abandonar la ejecución del programa **mysql** como administradores, podemos emplear los comandos `quit` o `exit`.

```
mysql> GRANT ALL ON catalogo.* TO 'eblanco'@'localhost';

Query OK, 0 rows affected (0,21 sec)

mysql> quit;
```

Figura 12. Autorizar el acceso a un nuevo usuario.
Fuente: elaboración propia.

A partir de este instante, a la hora de ejecutar el programa **mysql** vamos a trabajar sobre nuestra base de datos con el nuevo usuario **eblanco**. En primer lugar, debemos acceder al gestor de MySQL empleando el nombre de usuario y la contraseña que hemos creado.

```
% mysql -u eblanco -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or g.
Your MySQL connection id is 8
Server version: 5.7.17-0ubuntu0.16.04.1 (Ubuntu)
Copyright (c) 2000, 2016, Oracle and/or its affiliates. All
rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or
its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or 'h' for help. Type 'c' to clear the current
input statement.

mysql>
```

Figura 13. Iniciar el gestor MySQL como un usuario convencional.
Fuente: elaboración propia.

Como es la primera vez que accedemos con este usuario, vamos a obtener información sobre el listado de las bases de datos accesibles utilizando el comando `SHOW DATABASES`.

Como mostramos a continuación, en la figura 14, nuestro nuevo usuario puede trabajar con dos bases de datos: **catalogo** y una segunda base de datos (**information_schema**) que contiene información interna sobre la configuración del sistema.

Con el comando `SHOW GRANTS` podemos ver también las autorizaciones que el administrador ha concedido a este usuario.

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| catalogo           |
+--------------------+
2 rows in set (0,00 sec)

mysql> SHOW GRANTS;
+--------------------------------------------------------------+
| Grants for eblanco@localhost                                  |
+--------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'eblanco'@'localhost'
|
| GRANT ALL PRIVILEGES ON `catalogo`.* TO 'eblanco'@'localhost'
|
+--------------------------------------------------------------+
2 rows in set (0,00 sec)
```

Figura 14. Conocer las bases de datos disponibles.
Fuente: elaboración propia.

Cualquier usuario, una vez dentro del sistema, debe especificar el nombre de la base de datos que va a usar antes de empezar a realizar operaciones sobre ella. Para indicar el nombre de la base de datos que vamos a seleccionar, usaremos el comando **USE**:

```
USE basededatos;
```

Figura 15. Sintaxis del comando USE.
Fuente: elaboración propia.

Como deseamos trabajar con la base de datos **catalogo**, procedemos a declarar este hecho en el intérprete de MySQL. Una vez esta instrucción ha sido ejecutada con éxito, ya estamos en disposición de crear las tablas que sirven como soporte de las entidades y las relaciones diseñadas con anterioridad para poblarlas posteriormente con nuevos registros.

```
mysql> USE catalogo;

Database changed
```

Figura 16. Seleccionar la base de datos para trabajar.
Fuente: elaboración propia.

1. Bases de datos relacionales

1.5. Creación de tablas y restricciones

Una base de datos está formada por un conjunto de tablas que nos permitirán estructurar la información que percibimos en un escenario concreto del mundo real. Cada tabla almacenará en forma de registros la serie de ejemplos de cada clase de entidades o relaciones entre entidades especificadas previamente. Para crear una tabla de registros vacía con el comando `CREATE TABLE` es necesario primero declarar sus atributos (consultad la figura 17).

```
CREATE TABLE nombre
(campo1 tipo1 [NOT NULL, AUTO_INCREMENT],
 campo2 tipo2 [NOT NULL, AUTO_INCREMENT],
 ...
 campoN tipoN [NOT NULL, AUTO_INCREMENT],
 PRIMARY KEY(campox,campoy,...),
 [FOREIGN KEY(campox,campoy,...)
 REFERENCES tabla(campoa,campob,...)]);
```

Figura 17. Sintaxis del comando `CREATE TABLE`.
Los elementos opcionales se muestran entre corchetes.
Fuente: elaboración propia.

A la hora de definir la clase de información que almacenaremos en cada atributo, MySQL proporciona una gran variedad de tipos numéricos y alfanuméricos básicos (tabla 2). El espacio de memoria requerido para almacenar cada variable depende de la precisión especificada en cada caso. Los tipos `DATE` y `TIME` resultan especialmente útiles para llevar el registro de nuestras actividades en el tiempo. El usuario puede declarar, además, variables del tipo objeto (en inglés, *Binary Large Objects* o BLOB) para almacenar ficheros de texto, documentos en formato PDF o incluso imágenes dentro de alguna tabla de la base de datos.

Tabla 2. Tipos de datos en MySQL.

Tipo genérico	Tipos MySQL
Entero	TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT
Decimal	DECIMAL, FLOAT, DOUBLE/REAL
Texto	CHAR, VARCHAR, TINYTEXT, TEXT
Objetos	BLOB, MEDIUMBLOB, LONGBLOB
Tiempo	DATE, TIME

Fuente: elaboración propia.

Junto con la declaración de los atributos, para crear una tabla debemos especificar qué atributo o combinación de atributos será la clave primaria, identificando de forma unívoca cada instancia (figura 18). En caso de existir, las claves foráneas para referenciar a los atributos de otras tablas también deben indicarse explícitamente.

El diseñador puede activar dos controles internos sobre el valor de un atributo en el momento de registrar nuevas instancias en la base de datos. En primer lugar, es posible rechazar aquellos registros que no posean un valor definido para un atributo concreto. Esta circunstancia se especifica con la construcción `NOT NULL`, justo después de la declaración de tipos. `NOT NULL` sería una

restricción de campo obligatorio, no acepta valores nulos. En caso contrario, el sistema asignará por defecto el valor NULL a ese campo y aceptará valores nulos. Este requerimiento debe satisfacerse inexcusablemente en aquellos atributos que pertenecen a la clave primaria. En segundo lugar, para los identificadores numéricos asociados a cada instancia, el propio sistema puede encargarse de gestionar un contador automático de valores mediante la construcción `AUTO_INCREMENT`.

Volviendo nuevamente a nuestra base de datos **catalogo** (figura 16), nos encontramos ahora en disposición de crear las tablas del catálogo de genes especificadas formalmente en las figuras 18, 19, 20 y 21. Debemos escoger adecuadamente los tipos de datos para cada atributo o campo, según su contenido, definiendo claramente cuáles son las claves primarias y foráneas. El usuario puede empezar creando las tablas más elementales, es decir, aquellas que no poseen claves foráneas (genomas y funciones). Observad cómo declaramos la clave primaria y nos aseguramos de que ninguna instancia puede darse de alta en la base de datos con un valor nulo para las claves primarias, **especie** y **función**. Para verificar que el proceso de creación ha funcionado correctamente, el usuario puede consultar en la base de datos sobre las tablas existentes con el comando `SHOW TABLES`.

```
mysql> SHOW TABLES;

Empty set (0,00 sec)

mysql> CREATE TABLE genomas
    -> (especie     VARCHAR(100) NOT NULL,
    ->  nombre      VARCHAR(100),
    ->  descripcion TEXT,
    ->  PRIMARY KEY (especie));

Query OK, 0 rows affected (0,28 sec)

mysql> CREATE TABLE funciones
    -> (funcion     VARCHAR(20) NOT NULL,
    ->  descripcion VARCHAR(100),
    ->  PRIMARY KEY (funcion));

Query OK, 0 rows affected (0,03 sec)

mysql> SHOW TABLES;

+--------------------+
| Tables_in_catalogo |
+--------------------+
| funciones          |
| genomas            |
+--------------------+
2 rows in set (0,00 sec)
```

Figura 18. Crear las tablas *genomas* y *funciones* en nuestra base de datos *catalogo*.
Fuente: elaboración propia.

Es posible revisar la definición de una tabla con la instrucción `DESCRIBE`:

```
mysql> DESCRIBE genomas;

+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| especie     | varchar(100) | NO   | PRI | NULL    |       |
| nombre      | varchar(100) | YES  |     | NULL    |       |
| descripcion | text         | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
3 rows in set (0,01 sec)
```

Figura 19. Muestra de la descripción de una tabla de nuestro catálogo.
Fuente: elaboración propia.

A continuación, para crear la tabla **genes**, además de la clave primaria, indicamos un campo o atributo que es la clave foránea (que debe apuntar o hacer referencia a la clave primaria de la tabla *genomas*):

```
mysql> CREATE TABLE genes
    -> (nombre    VARCHAR(20) NOT NULL,
    ->  cromosoma VARCHAR(5),
    ->  hebra     VARCHAR(1),
    ->  inicio    INT,
    ->  final     INT,
    ->  proteina  VARCHAR(20),
    ->  especie   VARCHAR(100),
    ->  PRIMARY KEY (nombre),
    ->  FOREIGN KEY (especie)
    ->  REFERENCES genomas(especie));

Query OK, 0 rows affected (0,06 sec)

mysql> DESCRIBE genes;

+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| nombre    | varchar(20)  | NO   | PRI | NULL    |       |
| cromosoma | varchar(5)   | YES  |     | NULL    |       |
| hebra     | varchar(1)   | YES  |     | NULL    |       |
| inicio    | int(11)      | YES  |     | NULL    |       |
| final     | int(11)      | YES  |     | NULL    |       |
| proteina  | varchar(20)  | YES  |     | NULL    |       |
| especie   | varchar(100) | YES  | MUL | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
7 rows in set (0,00 sec)
```

Figura 20. Creación de la tabla genes en nuestra base de datos *catalogo*.
Fuente: elaboración propia.

Finalmente, creamos la tabla **anotaciones** para relacionar los genes con las anotaciones funcionales. Junto con los dos valores que identifican cada registro (**gen** y **funcion**), añadiremos un atributo para registrar el origen de la información:

```
mysql> CREATE TABLE anotaciones
    -> (nombre       VARCHAR(20) NOT NULL,
    ->  funcion      VARCHAR(20) NOT NULL,
    ->  origen       VARCHAR(20),
    ->  PRIMARY KEY (nombre,funcion),
    ->  FOREIGN KEY (nombre)
    ->  REFERENCES  genes(nombre),
    ->  FOREIGN KEY (funcion)
    ->  REFERENCES  funciones(funcion));

Query OK, 0 rows affected (0,11 sec)

mysql> DESCRIBE anotaciones;

+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| nombre  | varchar(20) | NO   | PRI | NULL    |       |
| funcion | varchar(20) | NO   | PRI | NULL    |       |
| origen  | varchar(20) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
3 rows in set (0,01 sec)
```

Figura 21. Creación de la tabla *anotaciones* en nuestra base de datos *catalogo*.
Los dos componentes de la clave primaria de la tabla *anotaciones* deben declararse como claves foráneas con origen en las tablas *genes* y *funciones*.
Fuente: elaboración propia.

Es posible definir otras restricciones a los campos de las tablas con el comando CHECK.

Por ejemplo, podemos indicar que un campo numérico como el campo **incio** de la tabla **genes** que es tipo numérico solo acepte valores positivos CHECK (inicio >0), o que un campo de tipo cadena de caracteres solo acepte determinados valores; por ejemplo, para que el campo *hebra* de la tabla *genes* solo acepte los caracteres «+» o «-», podemos hacer hebra ENUM('+','-').

Una vez creada la tabla podemos modificarla con la instrucción **ALTER TABLE**.

Por ejemplo, si queremos eliminar el campo origen de la tabla **anotaciones** escribimos

```
ALTER TABLE anotaciones DROP COLUMN origen;
```

y si queremos volver a añadir el mismo campo escribimos

```
ALTER TABLE anotaciones ADD origen VARCHAR(20);
```

Durante el tiempo de vida de una base de datos es frecuente que debamos actualizar su contenido. En determinados casos, esto puede implicar la eliminación completa de usuarios, de tablas o incluso, de la propia base de datos. Para implementar estos servicios, MySQL posee la familia de comandos DROP.

```
DROP DATABASE basededatos;

DROP USER usuario;

DROP TABLE tabla;
```

Figura 22. Sintaxis del comando DROP.
Fuente: elaboración propia.

1. Bases de datos relacionales

1.6. Insertar y manipular datos en las tablas

Una vez el esquema relacional de entidades previamente diseñado está estructurado sobre MySQL mediante tablas, es el momento de dotar de contenido cada tabla.

MySQL dispone de dos formas de insertar nuevos registros en las tablas de la base de datos:

1. Carga simultánea de múltiples registros desde un fichero de texto.

2. Inserción individual de cada nuevo registro de forma manual.

Para importar una cantidad elevada de registros (*) es posible utilizar el comando LOAD DATA. Para invocar este comando debemos especificar el nombre del fichero de texto que alberga la información de los registros junto con el nombre de la tabla donde deben ser dados de alta. Es necesario disponer de acceso a un fichero de texto tabulado, donde cada fila representa un nuevo registro y cada columna alberga el valor de un atributo (especificado en el mismo orden que en la tabla).

```
LOAD DATA LOCAL INFILE fichero.txt INTO TABLE tabla;
```

En determinados entornos de UNIX es necesario activar específicamente la opción –local-infile a la hora de invocar al programa mysql. Esta opción, no obstante, está activada por defecto habitualmente en la mayoría de las distribuciones de Linux.

Podemos proceder a introducir los primeros datos en nuestro catálogo de genes. Es recomendable empezar por las tablas más elementales, aquellas que no poseen claves foráneas. En nuestro caso, las tablas **genoma** y **funciones** se ajustan perfectamente a esta descripción. Por ejemplo, para poblar la tabla **genomas** editaremos el siguiente fichero, **genomas.txt**, desde nuestro terminal de UNIX:

```
D. melanogaster Mosca de la fruta   Tambien denominada del vinagre
H. sapiens      Hombre              Nuestra propia especie
M. musculus     Raton               Otro organismo modelo
```

Para proceder a la carga de estos datos en la tabla *genomas*, el usuario debe introducir el siguiente comando desde el intérprete de MySQL:

```
mysql> LOAD DATA LOCAL INFILE 'genomas.txt' INTO TABLE genomas;

Query OK, 3 rows affected (0,08 sec)
Records:  3  Deleted:  0  Skipped:  0  Warnings:  0
```

Presentamos el contenido del fichero **funciones.txt,** que emplearemos para poblar la tabla *funciones* con tres nuevos registros:

```
GO:0003700          Factor de transcripcion
GO:0006338          Remodelado de cromatina
GO:0007254          Via JNK
```

Ahora empleamos el fichero **funciones.txt** para poblar la correspondiente tabla:

```
mysql> LOAD DATA LOCAL INFILE 'funciones.txt' INTO TABLE funciones;
Query OK,  3 rows affected (0,01 sec)
Records:  3  Deleted:  0  Skipped:  0  Warnings:  0
```

Una vez hemos poblado las tablas **genomas** y **funciones** con varias instancias de especies y funciones biológicas, respectivamente, es el momento de editar el fichero **genes.txt** para dar de alta nuevos genes en la tabla genes:

```
MYC   chr8  + 128748314 128753678 NP_002458 H.sapiens
HNF1A chr12 + 121416548 121440312 NP_000536 H.sapiens
cbt   chr2L -    476437    479046 NP_722636 D.melanogaster
ash2  chr3R +. 20477248  20479098 NP_733023 D.melanogaster
```

Y ahora procedemos a realizar la carga con el comando **LOAD DATA**:

```
mysql> LOAD DATA LOCAL INFILE 'gens.txt' INTO TABLE gens;

Query OK,  4 rows affected (0,02 sec)
Records:  4  Deleted:  0  Skipped:  0  Warnings:  0
```

Antes de proceder a realizar las primeras consultas, editaremos el fichero de texto anotaciones.txt para asignar funciones a los genes que hemos registrado en los comandos previos.

```
MYC    GO:0003700  Experimental
MYC    GO:0006338  Literatura
HNF1A  GO:0003700  Experimental
cbt    GO:0003700  Experimental
cbt    GO:0007254  Experimental
ash2   GO:0006338  Experimental
ash2   GO:0003700  Computacional
```

Observad que los genes pueden poseer más de una anotación funcional. Por otro lado, la misma función biológica puede ser desempeñada por genes distintos. Pero los dos valores juntos forman una clave única.

Estamos en condiciones de poblar nuestra última tabla *anotaciones*:

```
mysql> LOAD DATA LOCAL INFILE 'anotaciones.txt'
    -> INTO TABLE anotaciones;

Query OK,  7 rows affected (0,08 sec)
Records:  7  Deleted:  0  Skipped:  0  Warnings:  0
```

La carga de datos desde un fichero de texto en las tablas es extremadamente útil. No obstante, en determinados casos necesitamos dar de alta un nuevo registro de forma aislada, pero la edición de un fichero de texto únicamente con este objetivo es menos eficiente.

En estos casos, el comando INSERT es más adecuado, dado que implementa esta funcionalidad en el gestor MySQL de bases de datos. El usuario debe especificar en el mismo orden tanto la lista de atributos del nuevo registro como sus correspondientes valores. El resto de atributos no incluidos en la relación anterior tomarán el valor NULL (excepto para aquellos donde está expresamente prohibida esta circunstancia durante la creación de la tabla, campos obligatorios):

```
INSERT INTO tabla (campo1, campo2,...,campoN)
VALUES (valor1,valor2,...,valorN);
```

Por regla general, sin embargo, un registro contiene todos los campos declarados para una tabla. Por tanto, respetando el orden de los campos en la tabla, podemos omitir la relación completa de los atributos:

```
INSERT INTO tabla
VALUES (valor1,valor2,...,valorN);
```

A modo de ejemplo, mostramos a continuación la secuencia de comandos de inserción equivalente a la carga simultánea de las funciones ejecutada anteriormente.

Las cadenas de texto deben introducirse utilizando siempre comillas simples, mientras que los valores numéricos no necesitan ningún formato adicional.

```
mysql> INSERT INTO funciones
    -> VALUES ('GO:0003700',
    ->         'Factor de transcripcion');

mysql> INSERT INTO funciones
    -> VALUES ('GO:0006338',
    ->         'Remodelado de cromatina');

mysql> INSERT INTO funciones
    -> VALUES ('GO:0007254',
    ->         'Via JNK');
```

En el caso de intentar dar de alta un registro cuya clave primaria ya existe, el sistema nos advertirá del error, abortando dicha operación. Una tabla no puede tener una clave primaria repetida.

Una vez tenemos los datos introducidos en las tablas podemos modificarlos con la instrucción UPDATE o eliminar registros con la instrucción DELETE.

Por ejemplo, si queremos modificar el valor **'Factor de transcripción'** situado en el campo **descripcion** del registro o fila con clave primaria 'GO:0003700' de la tabla funciones, y queremos que el nuevo valor sea **'Transcription factor'** escribiremos la instrucción siguiente:

```
UPDATE funciones
SET descripcion ='Transcription factor'
WHERE funcion = 'GO:0003700';
```

Para eliminar únicamente algunos registros de una determinada tabla podemos utilizar el comando **DELETE** junto con la cláusula **WHERE**. De este modo, seleccionaremos con precisión los registros que deben ser dados de baja.

Si el usuario desea eliminar todos los registros de una tabla, conservando la estructura de esta (para reutilizarla en el futuro), basta con omitir la condición:

```
DELETE FROM tabla WHERE condiciones;
DELETE FROM tabla;
```

Si queremos eliminar el registro de la tabla **funciones** con clave primaria '**GO:0007254**' escribiremos la instrucción siguiente:

```
DELETE FROM funciones WHERE funcion = 'GO:0007254';
```

Si intentamos eliminar un registro que está referenciado por una clave foránea de otra tabla, el sistema lo impedirá y saltará un error, a no ser que a la clave foránea le indiquemos la opción **ON DELETE CASCADE**, que permite eliminar en cascada el registro que deseamos eliminar y los registros de la otra tabla que están referenciados.

1. Bases de datos relacionales

1.7. Consultas básicas en las tablas

Las bases de datos son herramientas excepcionalmente útiles para consultar información y extraer nuevo conocimiento. En este sentido, los esquemas entidad-relación no son una excepción. Más bien al contrario, utilizando el álgebra relacional es posible consultar el contenido de las tablas de múltiples formas. Básicamente, las consultas SQL (en inglés, *queries*) consisten en filtros que delimitan el segmento del conjunto completo de los registros de una o más tablas en el cual estamos interesados, para mostrar después el valor de sus atributos.

La instrucción **SELECT** implementa el proceso de consulta sobre las tablas, mostrando los atributos indicados para aquellos registros que cumplen una determinada condición.

Podéis encontrar información sobre el uso del terminal para llevar a cabo operaciones similares sobre ficheros de texto en el módulo «El entorno de trabajo UNIX».

Vamos a explorar en los próximos subapartados cómo enriquecer nuestras consultas, empleando para ello nuestro catálogo de genes, que está almacenado en la base de datos **catalogo**. Antes de hacer consultas más elaboradas, mostramos a continuación la forma más sencilla de realizar una consulta.

```
SELECT campo1,campo2,...,campon FROM tabla;
```

Figura 23. Sintaxis básica del comando SELECT.
Fuente: elaboración propia.

La consulta más habitual consiste en mostrar el contenido completo de una tabla. El carácter *, precisamente, indica que deseamos visualizar el listado íntegro de los valores de todos los atributos para el subconjunto de registros seleccionados. Para poner en práctica este comando sobre nuestro catálogo, seleccionamos todos los valores de cada instancia guardada en la tabla *genes*:

```
mysql> SELECT * FROM genes;

+--------+-----------+-------+-----------+-----------+-----------+----------------+
| nombre | cromosoma | hebra | inicio    | final     | proteina  | especie        |
+--------+-----------+-------+-----------+-----------+-----------+----------------+
| ash2   | chr3R     | +     |  20477248 |  20479098 | NP_733023 | D.melanogaster |
| cbt    | chr2L     | -     |    476437 |    479046 | NP_722636 | D.melanogaster |
| HNF1A  | chr12     | +     | 121416548 | 121440312 | NP_000536 | H.sapiens      |
| MYC    | chr8      | +     | 128748314 | 128753678 | NP_002458 | H.sapiens      |
+--------+-----------+-------+-----------+-----------+-----------+----------------+
4 rows in set (0,00 sec)
```

Figura 24. Mostrando el contenido íntegro de la tabla genes.
Fuente: elaboración propia.

Para evitar el exceso de información, el usuario puede seleccionar los atributos o campos de los registros de una tabla que desea ver por pantalla. Simplemente sustituyendo en la pregunta el símbolo * por un listado de atributos, separados por comas, podemos delimitar la vista de los registros, en este caso genes que obtenemos como resultado:

```
mysql> SELECT nombre,especie FROM genes;

+--------+----------------+
| nombre | especie        |
+--------+----------------+
| ash2   | D.melanogaster |
| cbt    | D.melanogaster |
| HNF1A  | H.sapiens      |
| MYC    | H.sapiens      |
+--------+----------------+
4 rows in set (0,00 sec)
```

Figura 25. Mostrando los valores de algunos atributos de la tabla *genes*.
Fuente: elaboración propia.

El recuento del número de registros que cumple una condición concreta es una de las consultas más frecuentes en SQL. En este caso es suficiente con añadir la función **COUNT** a la consulta que estemos realizando para contabilizar el número de líneas de la salida:

```
mysql> SELECT COUNT(*) FROM genes;

+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
1 row in set (0,00 sec)
```

Figura 26. Contando todos los registros de la tabla *genes*.
Fuente: elaboración propia.

La función **DISTINCT** elimina los resultados duplicados. Por ejemplo, si deseamos contar el número de organismos en nuestra tabla *genes*, podemos combinar las funciones **COUNT** y **DISTINCT** sobre el atributo *especies* del siguiente modo:

```
mysql> SELECT especie FROM genes;

+----------------+
| especie        |
+----------------+
| D.melanogaster |
| D.melanogaster |
| H.sapiens      |
| H.sapiens      |
+----------------+
4 rows in set (0,00 sec)

mysql> SELECT DISTINCT especie FROM genes;

+----------------+
| especie        |
+----------------+
| D.melanogaster |
| H.sapiens      |
+----------------+
2 rows in set (0,07 sec)

mysql> SELECT COUNT(DISTINCT especie) FROM genes;

+-------------------------+
| COUNT(DISTINCT especie) |
+-------------------------+
|                       2 |
+-------------------------+
1 row in set (0,07 sec)
```

Figura 27. Contando registros únicos de una tabla.
Fuente: elaboración propia.

El comando **ORDER BY** ordena la lista de resultados producida por un comando **SELECT**, de forma ascendente o descendente (según si añadimos la cláusula **ASC** o **DESC**, respectivamente). En la próxima figura ordenamos los genes por su posición en cada cromosoma o por su nombre, de distintas maneras:

```
mysql> SELECT nombre,cromosoma,inicio FROM genes ORDER BY inicio;

+--------+-----------+-----------+
| nombre | cromosoma | inicio    |
+--------+-----------+-----------+
| cbt    | chr2L     |    476437 |
| ash2   | chr3R     |  20477248 |
| HNF1A  | chr12     | 121416548 |
| MYC    | chr8      | 128748314 |
+--------+-----------+-----------+
4 rows in set (0,00 sec)

mysql> SELECT nombre,cromosoma,inicio FROM genes ORDER BY nombre;

+--------+-----------+-----------+
| nombre | cromosoma | inicio    |
+--------+-----------+-----------+
| ash2   | chr3R     |  20477248 |
| cbt    | chr2L     |    476437 |
| HNF1A  | chr12     | 121416548 |
| MYC    | chr8      | 128748314 |
+--------+-----------+-----------+
4 rows in set (0,00 sec)

mysql> SELECT nombre,cromosoma,inicio FROM genes
    -> ORDER BY nombre DESC;

+--------+-----------+-----------+
| nombre | cromosoma | inicio    |
+--------+-----------+-----------+
| MYC    | chr8      | 128748314 |
| HNF1A  | chr12     | 121416548 |
| cbt    | chr2L     |    476437 |
| ash2   | chr3R     |  20477248 |
+--------+-----------+-----------+
4 rows in set (0,01 sec)
```

Figura 28. Ordenar los registros de una tabla.
Fuente: elaboración propia.

Cuando se trabaja con tablas que contienen miles de elementos, resulta conveniente mostrar inicialmente solo los primeros registros para comprobar el correcto funcionamiento de la consulta. La función `LIMIT` permite mostrar exclusivamente los primeros *n* registros de la consulta en ejecución:

```
mysql> SELECT * FROM genes LIMIT 1;

| nombre | cromosoma | hebra | inicio   | final    | proteina  | especie         |
+--------+-----------+-------+----------+----------+-----------+-----------------+
| ash2   | chr3R     | +     | 20477248 | 20479098 | NP_733023 | D.melanogaster  |
+--------+-----------+-------+----------+----------+-----------+-----------------+
1 row in set (0,00 sec)
```

Figura 29. Mostrar un fragmento de la consulta.
Fuente: elaboración propia.

1. Bases de datos relacionales

1.8. Consultas básicas: filtros y condiciones

El comando SELECT permite también extraer de las tablas únicamente aquellos registros que poseen ciertas propiedades. Para especificar el filtro a realizar sobre el contenido de una tabla, debe añadirse la cláusula **WHERE**:

```
SELECT campo1,campo2,...,campon FROM tabla WHERE condicion;
```

Figura 30. Sintaxis básica del comando SELECT con condiciones.
Fuente: elaboración propia.

La condición puede ser simple o compuesta, evaluándose sobre uno o más atributos de varias tablas. Los operadores de comparación más habituales se muestran en la tabla 3.

Tabla 3. Operadores de comparación en consultas de MySQL.

Operador	Significado
`=,<>`	Igual/diferente
`<,>`	Menor/mayor
`<=,>=`	Menor/mayor o igual
`LIKE`	Búsqueda de un patrón de texto
`NOT`	Negación de una condición
`AND/OR`	Condiciones combinadas
`REGEXP`	Expresión regular

Fuente: elaboración propia.

Los operadores numéricos también resultan muy útiles para buscar registros en un rango concreto de fechas del calendario.

Vamos a probar estos operadores para realizar consultas más concretas sobre nuestra base de datos **catalogo**. En primer lugar, podemos interrogar a la base de datos sobre los genes ubicados en la hebra positiva de la cadena de ADN en cualquier especie, preguntar por aquellos que no pertenecen a nuestra especie o buscar los genes anotados antes del primer millón de bases en cualquier cromosoma:

```
mysql> SELECT * FROM genes WHERE hebra LIKE '+';

+--------+-----------+-------+-----------+-----------+-----------+----------------+
| nombre | cromosoma | hebra | inicio    | final     | proteina  | especie        |
+--------+-----------+-------+-----------+-----------+-----------+----------------+
| ash2   | chr3R     | +     |  20477248 |  20479098 | NP_733023 | D.melanogaster |
| HNF1A  | chr12     | +     | 121416548 | 121440312 | NP_000536 | H.sapiens      |
| MYC    | chr8      | +     | 128748314 | 128753678 | NP_002458 | H.sapiens      |
+--------+-----------+-------+-----------+-----------+-----------+----------------+
3 rows in set (0,00 sec)

mysql> SELECT * FROM genes WHERE especie NOT LIKE 'H.sapiens';

+--------+-----------+-------+----------+----------+-----------+----------------+
| nombre | cromosoma | hebra | inicio   | final    | proteina  | especie        |
+--------+-----------+-------+----------+----------+-----------+----------------+
| ash2   | chr3R     | +     | 20477248 | 20479098 | NP_733023 | D.melanogaster |
| cbt    | chr2L     | -     |   476437 |   479046 | NP_722636 | D.melanogaster |
+--------+-----------+-------+----------+----------+-----------+----------------+
2 rows in set (0,00 sec)

mysql> SELECT * FROM genes WHERE inicio <= 1000000;

+--------+-----------+-------+--------+--------+-----------+----------------+
| nombre | cromosoma | hebra | inicio | final  | proteina  | especie        |
+--------+-----------+-------+--------+--------+-----------+----------------+
| cbt    | chr2L     | -     | 476437 | 479046 | NP_722636 | D.melanogaster |
+--------+-----------+-------+--------+--------+-----------+----------------+
1 row in set (0,00 sec)
```

Figura 31. Consultas con una condición.
Fuente: elaboración propia.

La cláusula puede complementarse con el modificador %, que actúa de comodín en las expresiones alfanuméricas. A continuación, seleccionamos solo aquellos registros que pertenecen al genoma de la mosca:

```
mysql> SELECT * FROM genes WHERE especie LIKE '%melano%';

+--------+-----------+-------+----------+----------+-----------+----------------+
| nombre | cromosoma | hebra | inicio   | final    | proteina  | especie        |
+--------+-----------+-------+----------+----------+-----------+----------------+
| ash2   | chr3R     | +     | 20477248 | 20479098 | NP_733023 | D.melanogaster |
| cbt    | chr2L     | -     |   476437 |   479046 | NP_722636 | D.melanogaster |
+--------+-----------+-------+----------+----------+-----------+----------------+
2 rows in set (0,00 sec)
```

Figura 32. Consultas sobre patrones de texto.
Fuente: elaboración propia.

También podemos combinar preguntas sobre valores de distintos tipos. Por ejemplo, si deseamos averiguar cuántos genes de la mosca de la fruta están anotados en la hebra positiva de la cadena de ADN:

```
mysql> SELECT * FROM genes WHERE especie LIKE '%melano%'
    -> AND hebra LIKE '+';

+--------+-----------+-------+----------+----------+-----------+----------------+
| nombre | cromosoma | hebra | inicio   | final    | proteina  | especie        |
+--------+-----------+-------+----------+----------+-----------+----------------+
| ash2   | chr3R     | +     | 20477248 | 20479098 | NP_733023 | D.melanogaster |
+--------+-----------+-------+----------+----------+-----------+----------------+
1 row in set (0,00 sec)
```

Figura 33. Consultas con dos condiciones.
Fuente: elaboración propia.

1.9. Consultas avanzadas: agrupaciones

Mediante el desglose de datos de una tabla, en función de algún campo concreto, podemos calcular estadísticas sobre cada categoría. El comando GROUP BY permite realizar agrupaciones de los datos de las tablas según los criterios que establezcamos, y es posible combinar estas clasificaciones con operadores de agregación como **COUNT, MAX, MIN, AVG** o SUM.

```
SELECT campo1,campo2,...,campon FROM tabla GROUP BY atributo;
```

Figura 34. Sintaxis del comando SELECT sobre grupos.
Fuente: elaboración propia.

Por ejemplo, solicitamos las especies presentes en nuestra base de datos:

```
mysql> SELECT especie FROM genes GROUP BY especie;

+----------------+
| especie        |
+----------------+
| D.melanogaster |
| H.sapiens      |
+----------------+
2 rows in set (0,00 sec)
```

Figura 35. Datos agrupados por especie.
Fuente: elaboración propia.

Posteriormente, podemos contar el número exacto de ejemplos de cada especie:

```
mysql> SELECT especie,COUNT(*) FROM genes GROUP BY especie;

+----------------+----------+
| especie        | COUNT(*) |
+----------------+----------+
| D.melanogaster |        2 |
| H.sapiens      |        2 |
+----------------+----------+
2 rows in set (0,00 sec)
```

Figura 36. Número de especies almacenadas en la tabla *genes*.
Fuente: elaboración propia.

Ahora obtenemos las estadísticas básicas sobre la longitud de los genes:

```
mysql> SELECT especie,cromosoma,inicio,final,final-inicio
    -> FROM genes;

+----------------+-----------+-----------+-----------+--------------+
| especie        | cromosoma | inicio    | final     | final-inicio |
+----------------+-----------+-----------+-----------+--------------+
| D.melanogaster | chr3R     |  20477248 |  20479098 |         1850 |
| D.melanogaster | chr2L     |    476437 |    479046 |         2609 |
| H.sapiens      | chr12     | 121416548 | 121440312 |        23764 |
| H.sapiens      | chr8      | 128748314 | 128753678 |         5364 |
+----------------+-----------+-----------+-----------+--------------+
4 rows in set (0,00 sec)

mysql> SELECT especie,AVG(final-inicio),MIN(final-inicio),
    -> MAX(final-inicio) FROM genes GROUP BY especie;

+----------------+-------------------+-------------------+-------------------+
| especie        | AVG(final-inicio) | MIN(final-inicio) | MAX(final-inicio) |
+----------------+-------------------+-------------------+-------------------+
| D.melanogaster |         2229.5000 |              1850 |              2609 |
| H.sapiens      |        14564.0000 |              5364 |             23764 |
+----------------+-------------------+-------------------+-------------------+
2 rows in set (0,00 sec)
```

Figura 37. Cálculo de promedios en la base de datos *catalogo*.
Fuente: elaboración propia.

1. Bases de datos relacionales

1.10. Consultas avanzadas: consultas multitabla

Para aprovechar al máximo el modelo relacional y generar nuevo conocimiento de los datos existentes resulta esencial combinar información de varias tablas. Mediante un atributo que dos tablas posean en común, esta operación resulta extremadamente sencilla con SQL. Mientras efectuamos una consulta con el comando `SELECT`, debemos emplear la cláusula **`JOIN`** especificando el atributo compartido por ambas tablas. Cuando se referencian atributos de dos o más tablas en la misma consulta, es necesario utilizar la sintaxis `nombre_tabla.nombre_atributo` para especificar claramente el origen de cada atributo. Es posible añadir condiciones sobre otros atributos de las tablas con la cláusula **`WHERE`**.

Dadas dos tablas que denominaremos **tabla1** y **tabla2**, que poseen un atributo comparable (**tabla1.x** y **tabla2.y**), la sintaxis del comando `JOIN` para obtener tanto los valores en común como los valores presentes exclusivamente en la primera o en la segunda tabla, respectivamente, es la siguiente:

```
SELECT tabla1.x,tabla2.y
FROM tabla1 JOIN tabla2
ON tabla1.x=tabla2.y
WHERE condiciones;
-----------------------------------------------
SELECT tabla1.x,tabla2.y
FROM tabla1 LEFT JOIN tabla2
ON tabla1.x=tabla2.y
WHERE condiciones;
-----------------------------------------------
SELECT tabla1.x,tabla2.y
FROM tabla1 RIGHT JOIN tabla2
ON tabla1.x=tabla2.y
WHERE condiciones;
```

Figura 38. Sintaxis de la cláusula `JOIN`.
Fuente: elaboración propia.

Para trabajar con más de dos tablas, podemos generalizar la misma sintaxis (por ejemplo, `tabla1 JOIN (tabla2,3,4)`).

Antes de proceder a ejecutar consultas sobre las tablas de la base de datos **catalogo**, proponemos poner a prueba el funcionamiento de la cláusula `JOIN` sobre un ejemplo más simple.

Vamos a generar dos tablas que denominaremos **tabla1** y **tabla2**, con un único atributo. Posteriormente, poblaremos ambas tablas con una serie de valores tales que resultará muy sencillo mostrar el conjunto completo de combinaciones a la hora de comparar las dos tablas.

Primero procedemos a crear las dos tablas:

```
mysql> CREATE TABLE tabla1
    -> (x VARCHAR(10));

Query OK, 0 rows affected (0,5 sec)

mysql> CREATE TABLE tabla2
    -> (y VARCHAR(10));

Query OK, 0 rows affected (0,5 sec)
```

Figura 39. Crear dos tablas para combinar con la cláusula JOIN.
Fuente: elaboración propia.

Os animamos a reproducir en vuestro ordenador los comandos de SQL presentados en este apartado.

Ahora vamos a crear dos sencillos ficheros de texto con tres valores cada uno: (1,2,3) y (3,4,5). De este modo, vamos a poder estudiar detalladamente la clase de consulta de MySQL que necesitamos para recuperar los valores comunes entre ambas tablas (3) o, alternativamente, los valores que únicamente aparecen en la primera (1 y 2) o en la segunda tabla (4 y 5).

```
1
2
3

-----------------------------------------------

3
4
5
```

Figura 40. Los ficheros datos1.txt y datos2.txt.
Fuente: elaboración propia.

Finalmente, poblamos las dos tablas utilizando ambos ficheros:

```
mysql> LOAD DATA LOCAL INFILE 'datos1.txt' INTO TABLE tabla1;

Query OK, 3 rows affected (0,67 sec)
Records: 3  Deleted: 0  Skipped: 0  Warnings: 0

mysql> LOAD DATA LOCAL INFILE 'datos2.txt' INTO TABLE tabla2;

Query OK, 3 rows affected (0,43 sec)
Records: 3  Deleted: 0  Skipped: 0  Warnings: 0
```

Figura 41. Poblamos las dos tablas para combinar con la cláusula JOIN.
Fuente: elaboración propia.

Ya estamos en disposición de profundizar en el funcionamiento de las consultas que incluyen la cláusula **JOIN**. Si no añadimos ninguna opción, este comando genera todas las parejas posibles cuyo primer elemento pertenece a la primera tabla y el segundo elemento pertenece a la segunda:

```
mysql> SELECT tabla1.x,tabla2.y
    -> FROM tabla1 JOIN tabla2;

+------+------+
| x    | y    |
+------+------+
| 1    | 3    |
| 2    | 3    |
| 3    | 3    |
| 1    | 4    |
| 2    | 4    |
| 3    | 4    |
| 1    | 5    |
| 2    | 5    |
| 3    | 5    |
+------+------+
9 rows in set (0,00 sec)
```

Figura 42. Combinación con JOIN de dos tablas para obtener todas las combinaciones.
Fuente: elaboración propia.

Nuestro primer objetivo en una comparación es localizar los elementos comunes. Para ello es suficiente con incorporar la cláusula **ON** a la consulta y especificar el atributo compartido por las dos tablas. Esto filtrará aquellas parejas del resultado anterior que no cumplan esta propiedad, mostrándose aquello que buscábamos:

```
mysql> SELECT tabla1.x,tabla2.y
    -> FROM tabla1 JOIN tabla2
    -> ON tabla1.x=tabla2.y;

+------+------+
| x    | y    |
+------+------+
| 3    | 3    |
+------+------+
1 row in set (0,00 sec)
```

Figura 43. Combinación con JOIN de dos tablas con la cláusula ON.
Fuente: elaboración propia.

En determinados casos, en lugar de la lista de los valores comunes, estaremos interesados también en aquellos valores de una u otra tabla que no pertenecen a la otra. Para ello debemos modificar el comportamiento del comando JOIN con las cláusulas **LEFT** o **RIGHT**. Si realizamos una unión por la parte izquierda, recuperaremos un listado de los registros de la primera tabla junto con su registro equivalente en la segunda. En el caso de que este valor análogo no existiera, MySQL lo indicará con el valor NULL. Si, por el contrario, realizamos la unión por la derecha, obtendremos un listado de los valores de la segunda tabla bajo las mismas condiciones.

```
mysql> SELECT tabla1.x,tabla2.y
    -> FROM tabla1 LEFT JOIN tabla2
    -> ON tabla1.x=tabla2.y;

+------+------+
| x    | y    |
+------+------+
| 3    | 3    |
| 1    | NULL |
| 2    | NULL |
+------+------+
3 rows in set (0,00 sec)

mysql> SELECT tabla1.x,tabla2.y
    -> FROM tabla1 RIGHT JOIN tabla2
    -> ON tabla1.x=tabla2.y;

+------+------+
| x    | y    |
+------+------+
| 3    | 3    |
| NULL | 4    |
| NULL | 5    |
+------+------+
3 rows in set (0,00 sec)
```

Figura 44. Combinación con `JOIN` de dos tablas con las cláusulas `LEFT` y `RIGHT`.
Fuente: elaboración propia.

Para acabar de refinar el resultado, debemos mantener exclusivamente los valores que solo están en una tabla. Para lograrlo, podemos añadir al final de la consulta una condición `WHERE` que exija que el valor no esté presente en la otra tabla. La cláusula `IS` realiza la evaluación de la expresión que se encuentra a continuación, para acabar respondiendo con un valor booleano (cierto o falso).

```
mysql> SELECT tabla1.x,tabla2.y
    -> FROM tabla1 LEFT JOIN tabla2
    -> ON tabla1.x=tabla2.y
    -> WHERE tabla2.y IS NULL;

+------+------+
| x    | y    |
+------+------+
| 1    | NULL |
| 2    | NULL |
+------+------+
2 rows in set (0,00 sec)

mysql> SELECT tabla1.x,tabla2.y
    -> FROM tabla1 RIGHT JOIN tabla2
    -> ON tabla1.x=tabla2.y
    -> WHERE tabla1.x IS NULL;

+------+------+
| x    | y    |
+------+------+
| NULL | 4    |
| NULL | 5    |
+------+------+
2 rows in set (0,00 sec)
```

Figura 45. Combinación con `JOIN` de dos tablas empleando una condición.
Fuente: elaboración propia.

Ahora ya estamos en condiciones de efectuar consultas sobre dos o más tablas de nuestro catálogo de genes. Por ejemplo, podemos preguntar por aquellos genes humanos para los cuales se ha documentado una anotación funcional de carácter experimental:

```
mysql> SELECT genes.nombre,genes.especie,
    -> anotaciones.funcion,anotaciones.origen
    -> FROM genes JOIN anotaciones
    -> ON genes.nombre=anotaciones.nombre
    -> WHERE anotaciones.origen='experimental'
    -> AND genes.especie='H.sapiens';

+--------+-----------+------------+--------------+
| nombre | especie   | funcion    | origen       |
+--------+-----------+------------+--------------+
| HNF1A  | H.sapiens | GO:0003700 | Experimental |
| MYC    | H.sapiens | GO:0003700 | Experimental |
+--------+-----------+------------+--------------+
2 rows in set (0,00 sec)
```

Figura 46. Consultas sobre el catálogo de genes utilizando el comando `JOIN`.
Fuente: elaboración propia.

1. Bases de datos relacionales

1.11. Consultas avanzadas: subconsultas

En ocasiones deseamos realizar un tipo de pregunta sobre nuestros datos, pero esta no es factible porque la organización en tablas escogida no lo permite. Para solventar este problema, MySQL permite anidar una consulta dentro de otra, con el objetivo de utilizar la pregunta interior para darle la forma apropiada a los datos, que podrán ser tratados posteriormente mediante la consulta exterior.

Sintaxis básica de una subconsulta:

```
SELECT lista_columnas
FROM nombre_tabla
WHERE condición = (SELECT lista_columnas2
FROM nombre_tabla2
WHERE condiciones);
```

Sintácticamente, desde el punto de vista de la consulta principal, la subconsulta interior desempeñará el papel de una tabla convencional. Por esta razón, es posible asignar un nombre tanto a la consulta interior como a los atributos de los resultados que se desprenderán de esta. Para ello emplearemos la cláusula **AS**, que permite asociar un nombre a un grupo de operaciones o atributos en SQL. El nombre empleado para esos atributos resulta útil para referirse a ellos desde la consulta exterior.

```
SELECT subconsulta.valor1,..., subconsulta.valorn FROM
   (SELECT atributo1 AS valor1,..., atributon AS valorn
    FROM tabla GROUP BY atributoi) AS subconsulta;
```

Figura 47. Sintaxis de las subconsultas.
Fuente: elaboración propia.

Para ejemplificar la clase de escenario donde las subconsultas resultan potencialmente interesantes, imaginemos una tabla genérica llamada *tabla* con dos atributos, que denominaremos *clase* y *subclase*. Cada registro de esta tabla pertenece a una clase general, y dentro de esa clase, a una subclase más específica. Supongamos que nos gustaría calcular el promedio de subclases diferentes, clase por clase, que han sido utilizadas para etiquetar cada registro. Para obtener la respuesta, definiremos una subconsulta que recibirá el nombre de *contador*. Esta subpregunta agrupará los datos por clases para contar el número total de subclases asignado a los registros de cada clase principal. Finalmente, la consulta exterior simplemente deberá calcular el promedio de los totales calculados por la subconsulta.

```
clase1 subclasex
clase1 subclasey
clase1 subclasez
clase2 subclasea
clase2 subclaseb
clase3 subclasen
...
-----------------------------------------

SELECT AVG(contador.totales) FROM

(SELECT count(subclase) AS totales

 FROM tabla GROUP BY clase) AS contador;
```

Figura 48. Emplear una subconsulta dentro de una consulta principal.
Fuente: elaboración propia.

1.12. Otras utilidades de MySQL

Junto con el inventario de comandos que interactúan con nuestra base de datos empleando el lenguaje SQL, el SGBD MySQL proporciona un conjunto de aplicaciones elementales para asistirnos a la hora de trabajar con el sistema.

Por ejemplo, el comando `help` muestra por pantalla un breve manual de ayuda sobre cada instrucción de MySQL.

Para poder paginar sobre las entradas del manual, pantalla a pantalla, debemos ejecutar antes la aplicación **pager**. Dicho comando nos permite vincular una aplicación de paginación del terminal de Linux con el intérprete de MySQL (por ejemplo, el programa *more*).

```
mysql> pager more;

PAGER set to 'more'

mysql>  help DROP TABLE;

Name: 'DROP TABLE'
Description:
Syntax:
DROP [TEMPORARY] TABLE [IF EXISTS]
    tbl_name [, tbl_name] ...
    [RESTRICT | CASCADE]

DROP TABLE removes one or more tables. You must have the DROP privilege
for each table. All table data and the table definition are removed, so
be careful with this statement! If any of the tables named in the
argument list do not exist, MySQL returns an error indicating by name
which nonexisting tables it was unable to drop, but it also drops all
of the tables in the list that do exist.

--More--
```

Figura 49. Muestra del manual de ayuda de MySQL.
Fuente: elaboración propia.

El comando `source` permite ejecutar ficheros de texto que incluyen comandos MySQL con la secuencia de instrucciones precisas para realizar una determinada tarea. Por ejemplo, podemos editar un fichero de texto desde nuestro terminal con los primeros comandos que ejecutamos al entrar en el sistema:

```
USE catalogo;
SHOW TABLES;
DESCRIBE genes;
-----------------------------------------
mysql> source comandos.sql;

Database changed

+--------------------+
| Tables_in_catalogo |
+--------------------+
| anotaciones        |
| funciones          |
| genes              |
| genomas            |
+--------------------+
4 rows in set (0,00 sec)

+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| nombre    | varchar(20)  | NO   | PRI | NULL    |       |
| cromosoma | varchar(5)   | YES  |     | NULL    |       |
| hebra     | varchar(1)   | YES  |     | NULL    |       |
| inicio    | int(11)      | YES  |     | NULL    |       |
| final     | int(11)      | YES  |     | NULL    |       |
| proteina  | varchar(20)  | YES  |     | NULL    |       |
| especie   | varchar(100) | YES  | MUL | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
7 rows in set (0,00 sec)
```

Figura 50. Ejecución del fichero de comandos comandos.sql de MySQL.
Fuente: elaboración propia.

El comando `status` permite ver la configuración del sistema:

```
mysql> status;

mysql  Ver 14.14 Distrib 5.7.17, for Linux (i686) using  EditLine wrapper
Connection id:          7
Current database:       catalogo
Current user:           eblanco@localhost
SSL:                    Not in use
Current pager:          more
Using outfile:          ''
Using delimiter:        ;
Server version:         5.7.17-0ubuntu0.16.04.1 (Ubuntu)
Protocol version:       10
Connection:             Localhost via UNIX socket
Server characterset:    latin1
Db     characterset:    latin1
Client characterset:    utf8
Conn.  characterset:    utf8
UNIX socket:            /var/run/mysqld/mysqld.sock
Uptime:                 4 hours 14 min 12 sec
Threads: 1  Questions: 101  Slow queries: 0  Opens: 129
Flush tables: 1  Open tables: 42  Queries per second avg: 0.006
```

Figura 51. Muestra de la configuración actual de MySQL.
Fuente: elaboración propia.

Si en algún instante necesitamos acceder al terminal de Linux, podemos emplear el comando **system** para invocar sus comandos desde el interior de MySQL:

```
mysql> system (ls /home/eblanco);

Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos

mysql> system (cat comandos.sql);

USE catalogo;

SHOW TABLES;

DESCRIBE genes;
```

Figura 52. Ejecución del intérprete de comandos de Linux en MySQL.
Fuente: elaboración propia.

1.13. Copias de seguridad y restauración de BBDD

Es altamente recomendable llevar a cabo copias de seguridad de nuestros datos con cierta periodicidad. En el caso de las bases de datos gestionadas con MySQL, el programa **mysqldump**, ejecutado desde el terminal de Linux, realiza un volcado completo de su contenido hacia un fichero de texto. Posteriormente, en caso de ser necesario, este fichero de comandos puede ser ejecutado con la instrucción **source** para regenerar la base de datos completa, creando automáticamente las tablas e insertando los registros existentes en ese momento:

```
% mysqldump -vp catalogo > catalogo.sql

Enter password:
-- Connecting to localhost...
-- Retrieving table structure for table anotaciones...
-- Sending SELECT query...
-- Retrieving rows...
-- Retrieving table structure for table funciones...
-- Sending SELECT query...
-- Retrieving rows...
-- Retrieving table structure for table genes...
-- Sending SELECT query...
-- Retrieving rows...
-- Retrieving table structure for table genomas...
-- Sending SELECT query...
-- Retrieving rows...
-- Disconnecting from localhost...

% more catalogo.sql

-- MySQL dump 10.13 Distrib 5.7.17, for Linux (i686)
-- Host: localhost Database: catalogo
-- Server version 5.7.17-0ubuntu0.16.04.1
...
CREATE TABLE `anotaciones` (
...
INSERT INTO `anotaciones` VALUES ('ash2','GO:0003700','Computacional'),...
...
```

Figura 53. Realizar una copia de seguridad con el programa mysqldump.
Fuente: elaboración propia.

También podemos restaurar una base de datos MySQL desde el terminal a partir de un fichero *backup* así:

```
mysql -u usuario -p basededatos < basededatos.sql
```

1.14. Ejemplo práctico: gestión del catálogo de genes humanos

Para demostrar cómo extraer conocimiento útil de una base de datos relacional, os proponemos analizar con MySQL el contenido de un catálogo de genes humanos. Un gen es un fragmento de ADN ubicado en el genoma que contiene la información precisa para sintetizar una molécula de ARN. En los organismos eucariotas, un gen está constituido por una sucesión de fragmentos útiles denominados *exones*. En una proporción significativa de los genes humanos existen varias combinaciones alternativas de exones, dando lugar a distintas formas alternativas de un mismo gen, denominadas *transcritos alternativos*. Para codificar la información relativa a la localización de los genes en el genoma es frecuente utilizar ficheros de texto tabulado. Cada línea de estos ficheros contiene los valores de los atributos que caracterizan a un transcrito de un determinado gen. Básicamente, un transcrito de un gen posee una localización concreta, identificada por un cromosoma, una posición inicial/final y una dirección de lectura. Otras características que podemos recuperar sobre un transcrito son su código, el nombre del gen, el número de exones o sus coordenadas exactas.

Ved también

Para revisar los conceptos de *genoma*, *cromosoma*, *gen* y *proteína* os recomendamos la asignatura Fundamentos de biología molecular.

El navegador genómico de UCSC representa gráficamente los diferentes tipos de anotaciones existentes sobre el genoma humano en forma de cientos de pistas. Para administrar eficientemente este elevado volumen de información, una copia del SGBD MySQL está funcionando de forma transparente a los miles de usuarios que cada día visitan este servidor. De este modo, en el caso de que deseemos reproducir una pista en nuestro ordenador, disponemos en la sección de descargas de un fichero SQL para ser ejecutado con el comando `source` y un fichero de texto con el conjunto de datos que deben importarse con la instrucción `LOAD DATA`. En este ejercicio vamos a utilizar la anotación de los genes humanos distribuida por el consorcio **RefSeq** para el genoma humano. Este formato es común a todas las especies suministradas por el navegador.

Ved también

Es posible profundizar sobre el funcionamiento de los navegadores genómicos en la asignatura Genómica computacional.

Vamos a proceder ahora a descargar los dos ficheros asociados a la pista **refGene**, que contiene el catálogo de genes humanos anotados por el consorcio **RefSeq**, en su versión **hg38**.

Para ello, debemos utilizar el comando `wget` para transferir ambos ficheros a nuestro terminal.

`wget` http://hgdownload.soe.ucsc.edu/goldenPath/hg38/database/refGene.sql

`wget` http://hgdownload.soe.ucsc.edu/goldenPath/hg38/database/refGene.txt.gz

Mostramos a continuación el contenido del fichero **refGene.sql** que realiza la creación de la tabla **refGene**. Los atributos que consultaremos con mayor frecuencia serán (figura 54): **name** (código del transcrito), **chrom** (cromosoma), **strand** (hebra), **txStart** y **txEnd** (coordenadas de inicio y final), **exonCount** (número de exones) y **name2** (nombre del gen).

Es importante no confundir los campos de **name** y **name2**: un gen puede tener varios transcritos, pero un transcrito únicamente pertenece a un gen.

```
CREATE TABLE 'refGene' (
  'bin' smallint(5) unsigned NOT NULL,
  'name' varchar(255) NOT NULL,
  'chrom' varchar(255) NOT NULL,
  'strand' char(1) NOT NULL,
  'txStart' int(10) unsigned NOT NULL,
  'txEnd' int(10) unsigned NOT NULL,
  'cdsStart' int(10) unsigned NOT NULL,
  'cdsEnd' int(10) unsigned NOT NULL,
  'exonCount' int(10) unsigned NOT NULL,
  'exonStarts' longblob NOT NULL,
  'exonEnds' longblob NOT NULL,
  'score' int(11) DEFAULT NULL,
  'name2' varchar(255) NOT NULL,
  'cdsStartStat' enum('none','unk','incmpl','cmpl') NOT NULL,
  'cdsEndStat' enum('none','unk','incmpl','cmpl') NOT NULL,
  'exonFrames' longblob NOT NULL,
  KEY 'chrom' ('chrom','bin'),
  KEY 'name' ('name'),
  KEY 'name2' ('name2')
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Figura 54. Atributos de los transcritos anotados por el consorcio RefSeq.
Fuente: elaboración propia.

Pasaremos ahora a visualizar con el terminal el segundo fichero **refGene.txt**. Este archivo contiene los datos del catálogo completo de genes anotados en el genoma humano. Debemos cargar esta información en nuestra base de datos una vez esté creada la tabla **refGene**. En el contexto de este ejercicio, cada registro contiene información sobre un transcrito de un determinado gen. En el caso de que un gen posea varios transcritos, cada uno se codifica en registros separados (cada uno con su propio código y sus correspondientes coordenadas).

En primer lugar, debemos descomprimir el fichero con el comando **gzip**.

```
% gzip -d refGene.txt.gz

% head -5 refGene.txt

585 NR_046018 chr1 + 11873 14409 14409 14409 3 11873,12612,13220, 12227,12721,14409, 0 DDX11L1
unk unk -1,-1,-1,

585 NR_024540 chr1 - 14361 29370 29370 29370 11 14361,14969,15795,16606,16857,17232,17605,17914,
18267,24737,29320, 14829,15038,15947,16765,17055,17368,17742,18061,18366,24891,29370, 0 WASH7P
unk unk -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,

932 NR_104645 chrX + 45505387 45523644 45523644 45523644 3 45505387,45510496,45521607, 45505465,
45510595,45523644, 0 LINC01204 unk unk -1,-1,-1,

1078 NR_104148 chr7 + 64666082 64687830 64687830 64687830 4 64666082,64669036,64679176,64684334,
64666285,64669178,64679336,64687830, 0 ZNF107 unk unk -1,-1,-1,-1,

103 NR_120408 chr14 + 31561384 31861223 31861223 31861223 10 31561384,31562067,31565013,31599288,
31673354,31673483,31826628,31846470,31850118,31859117, 31561547,31562215,31565048,31599379,
31673394,31673574,31826714,31846591,31850201,31861223, 0NUBPL unk unk -1,-1,-1,-1,-1,-1,-1,-1,-1,
-1,
```

Figura 55. El catálogo refGene.txt de genes humanos.
Fuente: elaboración propia.

Las anotaciones de un genoma suelen actualizarse frecuentemente. Por este motivo, los datos mostrados en este tutorial pueden variar ligeramente con el paso del tiempo.

Una vez dentro del intérprete de MySQL, indicaremos que vamos a trabajar dentro de nuestra base de datos **catalogo**.

Ejecutaremos, posteriormente, el fichero **refGene.sql** con el comando `source` para crear la tabla **refGene**.

Para verificar que la instrucción anterior ha funcionado correctamente, podemos ver el listado de atributos de la tabla **refGene** con el comando `DESCRIBE`:

```
mysql> USE catalogo;

Database changed

mysql> source refGene.sql;

Query OK, 0 rows affected (0,00 sec)

mysql> DESCRIBE refGene;

+--------------+----------------------------------+------+-----+---------+-------+
| Field        | Type                             | Null | Key | Default | Extra |
+--------------+----------------------------------+------+-----+---------+-------+
| bin          | smallint(5) unsigned             | NO   |     | NULL    |       |
| name         | varchar(255)                     | NO   | MUL | NULL    |       |
| chrom        | varchar(255)                     | NO   | MUL | NULL    |       |
| strand       | char(1)                          | NO   |     | NULL    |       |
| txStart      | int(10) unsigned                 | NO   |     | NULL    |       |
| txEnd        | int(10) unsigned                 | NO   |     | NULL    |       |
| cdsStart     | int(10) unsigned                 | NO   |     | NULL    |       |
| cdsEnd       | int(10) unsigned                 | NO   |     | NULL    |       |
| exonCount    | int(10) unsigned                 | NO   |     | NULL    |       |
| exonStarts   | longblob                         | NO   |     | NULL    |       |
| exonEnds     | longblob                         | NO   |     | NULL    |       |
| score        | int(11)                          | YES  |     | NULL    |       |
| name2        | varchar(255)                     | NO   | MUL | NULL    |       |
| cdsStartStat | enum('none','unk','incmpl','cmpl') | NO   |     | NULL    |       |
| cdsEndStat   | enum('none','unk','incmpl','cmpl') | NO   |     | NULL    |       |
| exonFrames   | longblob                         | NO   |     | NULL    |       |
+--------------+----------------------------------+------+-----+---------+-------+
16 rows in set (0,14 sec)
```

Figura 56. Creación de la tabla refGene.
Fuente: elaboración propia.

Vamos a asumir que ambos ficheros (refGene.sql y refGene.txt) están guardados en la misma carpeta de trabajo desde la cual hemos invocado al programa MySQL, anteriormente.

El segundo paso consiste en poblar la tabla con las anotaciones de los genes humanos que hemos descargado dentro del fichero **refGene.txt**. Usando el comando **LOAD DATA** podemos volcar todo el contenido en la tabla **refGene**:

```
mysql> LOAD DATA LOCAL INFILE 'refGene.txt' INTO TABLE refGene;

Query OK, 69853 rows affected (2,54 sec)
Records: 69853  Deleted: 0  Skipped: 0  Warnings: 0
```

Figura 57. Poblar la tabla refGene.
Fuente: elaboración propia.

Nos encontramos en condiciones de comenzar a interrogar a la base de datos. Recordemos, nuevamente, que cada registro de la tabla **refGene** alberga la información asociada al transcrito de un gen en particular. Igualmente, es importante tener en cuenta que una elevada fracción de los genes humanos posee dos o más transcritos alternativos. Nuestra misión, a continuación, es mostrar el enorme potencial de las consultas de SQL a la hora de extraer nuevo conocimiento biológico de los datos almacenados en las tablas de nuestra base de datos.

Comenzaremos mostrando los primeros registros de nuestra tabla, incluyendo únicamente varios de sus atributos para favorecer la legibilidad de los valores de los registros por pantalla:

```
mysql> SELECT name2,name,chrom,strand,txStart,txEnd,exonCount
    -> FROM refGene ORDER BY name2 LIMIT 10;

+----------+--------------+-------+--------+----------+----------+-----------+
| name2    | name         | chrom | strand | txStart  | txEnd    | exonCount |
+----------+--------------+-------+--------+----------+----------+-----------+
| A1BG     | NM_130786    | chr19 | -      | 58346805 | 58353499 |         8 |
| A1BG-AS1 | NR_015380    | chr19 | +      | 58351969 | 58355183 |         4 |
| A1CF     | NM_001198819 | chr10 | -      | 50799408 | 50885675 |        15 |
| A1CF     | NM_014576    | chr10 | -      | 50799408 | 50885675 |        13 |
| A1CF     | NM_138932    | chr10 | -      | 50799408 | 50885675 |        13 |
| A1CF     | NM_001198820 | chr10 | -      | 50799408 | 50885675 |        14 |
| A1CF     | NM_001198818 | chr10 | -      | 50799408 | 50885675 |        14 |
| A1CF     | NM_138933    | chr10 | -      | 50799408 | 50885675 |        13 |
| A2M      | NM_001347423 | chr12 | -      |  9067707 |  9116229 |        37 |
| A2M      | NM_000014    | chr12 | -      |  9067707 |  9116229 |        36 |
+----------+--------------+-------+--------+----------+----------+-----------+
10 rows in set (0,00 sec)
```

Figura 58. Muestra del contenido de la tabla refGene.
Fuente: elaboración propia.

Dado que cada registro contiene la información de un transcrito, el número de transcritos conocidos en el genoma humano coincidirá con el número de registros almacenados en la tabla **refGene**. Este contaje es sencillo:

```
mysql> SELECT COUNT(*) FROM refGene;

+----------+
| COUNT(*) |
+----------+
|    69853 |
+----------+
1 row in set (0,00 sec)
```

Figura 59. Contar los transcritos de la tabla refGene.
Fuente: elaboración propia.

También podemos contar fácilmente el número total de genes codificados en el genoma humano. Si un gen posee varios transcritos alternativos, entonces encontraremos varios registros en nuestro catálogo que poseen un valor distinto del atributo **name**, pero que comparten el mismo valor para el atributo **name2**. Por tanto, empleando la cláusula `DISTINCT` sobre este último atributo, contaremos una única vez cada gen de nuestra tabla, aunque posea varias formas alternativas:

```
mysql> SELECT COUNT(DISTINCT name2) FROM refGene;

+-----------------------+
| COUNT(DISTINCT name2) |
+-----------------------+
|                 27656 |
+-----------------------+
1 row in set (0,07 sec)
```

Figura 60. Contar los genes de la tabla refGene.
Fuente: elaboración propia.

Si agrupamos los registros de la tabla por el atributo **name2**, podemos elaborar un inventario del número de transcritos alternativos anotados para cada gen.

```
mysql> SELECT name2,COUNT(name2)
    -> FROM refGene GROUP BY name2 LIMIT 10;

+----------+--------------+
| name2    | COUNT(name2) |
+----------+--------------+
| A1BG     |            1 |
| A1BG-AS1 |            1 |
| A1CF     |            6 |
| A2M      |            4 |
| A2M-AS1  |            3 |
| A2ML1    |            2 |
| A2MP1    |            1 |
| A3GALT2  |            1 |
| A4GALT   |            3 |
| A4GNT    |            1 |
+----------+--------------+
10 rows in set (0,00 sec)
```

Figura 61. Contar el número de transcritos de cada gen de la tabla refGene.
Fuente: elaboración propia.

Podemos obtener resultados interesantes aplicando la cláusula **WHERE** sobre los atributos de cada registro. Por ejemplo, imaginemos que deseamos conocer el número de transcritos ubicados en cada hebra de la molécula de ADN:

```
mysql> SELECT COUNT(*) FROM refGene WHERE strand LIKE '+';

+----------+
| COUNT(*) |
+----------+
|    35724 |
+----------+
1 row in set (0,10 sec)

mysql> SELECT COUNT(*) FROM refGene WHERE strand LIKE '-';

+----------+
| COUNT(*) |
+----------+
|    34129 |
+----------+
1 row in set (0,09 sec)
```

Figura 62. Contar transcritos en una hebra de ADN.
Fuente: elaboración propia.

También podemos contar el número de transcritos localizados en un cromosoma:

```
mysql> SELECT COUNT(*) FROM refGene WHERE chrom LIKE 'chr21';

+----------+
| COUNT(*) |
+----------+
|      961 |
+----------+
1 row in set (0,00 sec)
```

Figura 63. Contar los transcritos de un cromosoma.
Fuente: elaboración propia.

Nuevamente, jugando con el atributo **name2** podemos contar el número de genes codificados en el mismo cromosoma:

```
mysql> SELECT COUNT(DISTINCT name2)
    -> FROM refGene WHERE chrom LIKE 'chr21';

+-----------------------+
| COUNT(DISTINCT name2) |
+-----------------------+
|                   408 |
+-----------------------+
1 row in set (0,01 sec)
```

Figura 64. Contar los genes de un cromosoma.
Fuente: elaboración propia.

O identificar cuáles son los transcritos que poseen un mayor número de exones:

```
mysql> SELECT name2,name,exonCount
    -> FROM refGene ORDER BY exonCount DESC LIMIT 10;

+-------+--------------+-----------+
| name2 | name         | exonCount |
+-------+--------------+-----------+
| TTN   | NM_001267550 |       363 |
| TTN   | NM_001256850 |       313 |
| TTN   | NM_133378    |       312 |
| TTN   | NM_133437    |       192 |
| TTN   | NM_133432    |       192 |
| TTN   | NM_003319    |       191 |
| NEB   | NM_001271208 |       183 |
| NEB   | NM_001164507 |       182 |
| NEB   | NM_001164508 |       182 |
| MUC19 | NM_173600    |       174 |
+-------+--------------+-----------+
10 rows in set (0,11 sec)
```

Figura 65. Recuperar los transcritos con mayor número de exones.
Fuente: elaboración propia.

También podemos seleccionar aquellos transcritos que poseen un único exón:

```
mysql> SELECT name2,name,exonCount
     > FROM refGene WHERE exonCount=1
     > ORDER BY name2 LIMIT 10;

+-------------+--------------+-----------+
| name2       | name         | exonCount |
+-------------+--------------+-----------+
| AADACL2-AS1 | NR_110203    |         1 |
| ABALON      | NR_131907    |         1 |
| ABHD16B     | NM_080622    |         1 |
| ACKR1       | NM_001122951 |         1 |
| ACKR4       | NM_178445    |         1 |
| ACTBL2      | NM_001017992 |         1 |
| ACTG1P20    | NR_033926    |         1 |
| ACTG1P4     | NR_024438    |         1 |
| ACTL10      | NM_001024675 |         1 |
| ACTL7A      | NM_006687    |         1 |
+-------------+--------------+-----------+
10 rows in set (0,00 sec)
```

Figura 66. Recuperar los transcritos con un único exón.
Fuente: elaboración propia.

Es posible calcular el número de exones, en promedio, por cada transcrito:

```
mysql> SELECT AVG(exonCount) FROM refGene;

+----------------+
| AVG(exonCount) |
+----------------+
|         9.4126 |
+----------------+
1 row in set (0,11 sec)
```

Figura 67. Calcular el número de exones en promedio por cada transcrito.
Fuente: elaboración propia.

Y la longitud en promedio de los transcritos de los genes humanos:

```
mysql> SELECT AVG(txEnd-txStart+1) FROM refGene;

+----------------------+
| AVG(txEnd-txStart+1) |
+----------------------+
|           56983.2770 |
+----------------------+
1 row in set (0,10 sec)
```

Figura 68. Calcular la longitud en promedio de los genes.
Fuente: elaboración propia.

Finalmente, vamos a integrar en este análisis el genoma de ratón doméstico. Descargamos los ficheros **refGene.sql** y **refGene.txt** de esta especie en su versión mm9.

Para evitar sobrescribir las anotaciones humanas, debemos grabar ambos ficheros con un nombre diferente (por ejemplo, **refGene_mouse.sql** y **refGene_mouse.txt**). Posteriormente, es necesario editar el contenido del fichero SQL para modificar el nombre de la tabla, por la misma razón (figura 69).

Tras estas modificaciones, ya estamos en condiciones de lanzar la creación de la nueva tabla con el comando **source** y su repoblación con los datos relativos al genoma del ratón con el comando `LOAD DATA`.

```
DROP TABLE IF EXISTS 'refGene_mouse';

CREATE TABLE 'refGene_mouse' (
  'bin' smallint(5) unsigned NOT NULL,
  'name' varchar(255) NOT NULL,
  'chrom' varchar(255) NOT NULL,
  'strand' char(1) NOT NULL,
...
------------------------------------------

mysql> source 'refGene_mouse.sql';

Query OK, 0 rows affected (0,00 sec)

mysql> LOAD DATA LOCAL INFILE 'refGene_mouse.txt'
    -> INTO TABLE refGene_mouse;

Query OK, 34904 rows affected (1,23 sec)
Records: 34904  Deleted: 0  Skipped: 0  Warnings: 0
```

Figura 69. Incorporar los genes de ratón en nuestra base de datos.
Fuente: elaboración propia.

Comprobamos que los registros almacenados en la nueva tabla son correctos:

```
mysql> SELECT name2,name,chrom,strand,txStart,txEnd,exonCount
    -> FROM refGene_mouse ORDER BY name2 LIMIT 10;

+---------------+--------------+-------+--------+-----------+-----------+-----------+
| name2         | name         | chrom | strand | txStart   | txEnd     | exonCount |
+---------------+--------------+-------+--------+-----------+-----------+-----------+
| 0610005C13Rik | NR_038166    | chr7  | -      |  52823164 |  52830546 |         5 |
| 0610005C13Rik | NR_038165    | chr7  | -      |  52823164 |  52830546 |         4 |
| 0610007P14Rik | NM_021446    | chr12 | -      |  87156404 |  87165495 |         5 |
| 0610009B22Rik | NM_025319    | chr11 | -      |  51498886 |  51502136 |         2 |
| 0610009L18Rik | NR_038126    | chr11 | +      | 120209991 | 120212504 |         2 |
| 0610009O20Rik | NM_024179    | chr18 | +      |  38409902 |  38422283 |        13 |
| 0610010B08Rik | NM_001177543 | chr2  | -      | 175017505 | 175163713 |         6 |
| 0610010B08Rik | NM_001177543 | chr2  | -      | 174952492 | 175261278 |         6 |
| 0610010B08Rik | NM_001177543 | chr2  | +      | 175639522 | 175655901 |         5 |
| 0610010B08Rik | NM_001177543 | chr2  | +      | 175737073 | 175753460 |         5 |
+---------------+--------------+-------+--------+-----------+-----------+-----------+
10 rows in set (0,00 sec)
```

Figura 70. Visualizar los primeros transcritos del genoma del ratón doméstico.
Fuente: elaboración propia.

Ahora, si seleccionamos aquellos registros de las dos tablas que pertenecen al mismo gen en ambas especies, podemos construir un catálogo de genes homólogos.

Podemos llevar a cabo esta asociación porque SQL no distingue entre mayúsculas o minúsculas a la hora de comparar la columna **name2**.

```
mysql> SELECT DISTINCT refGene.name2,refGene.chrom,refGene.strand,
    -> refGene.txStart,refGene.txEnd,refGene.exonCount,
    -> refGene_mouse.name2,refGene_mouse.chrom,
    -> refGene_mouse.strand,refGene_mouse.txStart,
    -> refGene_mouse.txEnd,refGene_mouse.exonCount
    -> FROM refGene JOIN refGene_mouse
    -> ON refGene.name2 = refGene_mouse.name2
    -> ORDER BY refGene.name2 ASC LIMIT 10;

+---------+-------+--------+----------+----------+-----------+---------+-------+--------+-----------+-----------+-----------+
| name2   | chrom | strand | txStart  | txEnd    | exonCount | name2   | chrom | strand | txStart   | txEnd     | exonCount |
+---------+-------+--------+----------+----------+-----------+---------+-------+--------+-----------+-----------+-----------+
| A1BG    | chr19 | -      | 58346805 | 58353499 |         8 | A1bg    | chr15 | -      |  60749143 |  60752825 |         7 |
| A1CF    | chr10 | -      | 50799408 | 50885675 |        13 | A1cf    | chr19 | +      |  31943250 |  32023896 |        12 |
| A1CF    | chr10 | -      | 50799408 | 50885675 |        14 | A1cf    | chr19 | +      |  31943250 |  32023896 |        12 |
| A1CF    | chr10 | -      | 50799408 | 50885675 |        15 | A1cf    | chr19 | +      |  31943250 |  32023896 |        12 |
| A2M     | chr12 | -      |  9067707 |  9116229 |        35 | A2m     | chr6  | +      | 121586190 | 121629256 |        36 |
| A2M     | chr12 | -      |  9067707 |  9116229 |        36 | A2m     | chr6  | +      | 121586190 | 121629256 |        36 |
| A2M     | chr12 | -      |  9067707 |  9116229 |        37 | A2m     | chr6  | +      | 121586190 | 121629256 |        36 |
| A3GALT2 | chr1  | -      | 33306765 | 33321098 |         5 | A3galt2 | chr4  | +      | 128436501 | 128446542 |         5 |
| A4GALT  | chr22 | -      | 42692111 | 42720910 |         3 | A4galt  | chr15 | -      |  83057151 |  83082161 |         3 |
| A4GALT  | chr22 | -      | 42692111 | 42720910 |         3 | A4galt  | chr15 | -      |  83057151 |  83082204 |         3 |
+---------+-------+--------+----------+----------+-----------+---------+-------+--------+-----------+-----------+-----------+
10 rows in set (5,14 sec)
```

Figura 71. Listado de genes comunes entre el genoma humano y el genoma de ratón.
Fuente: elaboración propia.

1.15. *Triggers*, procedimientos y funciones

La mayoría de bases de datos relacionales ofrecen la posibilidad de almacenar subprogramas. Se denominan funciones, procedimientos y *triggers* o disparadores, y son muy útiles para automatizar tareas y guardar instrucciones SQL que se tienen que utilizar frecuentemente. Son objetos que contienen código SQL, cómo breves *scripts* de código SQL, que pueden aceptar parámetros y declarar variables.

Se asigna un nombre al subprograma y se ejecuta para que quede almacenado en la base de datos. Después lo podemos invocar o *llamar* por su nombre porque se ejecute el código almacenado en los subprogramas.

- **Procedimiento** almacenado. Es un objeto que se crea con la sentencia `CREATE PROCEDURE` y se invoca con la sentencia `CALL`. Los procedimientos pueden aceptar parámetros y no hacen ningún retorno, es decir, no devuelven ningún valor.

- **Función** almacenada. Es un objeto que se crea con la sentencia `CREATE FUNCTION` y se invoca con la sentencia `SELECT` o dentro de una expresión. Las **funciones** pueden aceptar parámetros y devuelven siempre un valor. Este valor devuelto puede ser un valor nulo y en este caso se comportaría como un procedimiento.

- ***Trigger***. Es un objeto que se crea con la sentencia `CREATE TRIGGER` y tiene que estar asociado a una tabla. Un *trigger* se activa, se dispara, cuando ocurre un evento de inserción, actualización o borrado, sobre la tabla a la que está asociado.

Veamos algunos ejemplos:

Procedimientos

1. Creamos un procedimiento en la base de datos **catalogo** para contar el número de genes diferentes de la tabla **refGene**, que denominamos **numGenes**.

```
CREATE PROCEDURE numGenes()
SELECT COUNT(distinct name2) FROM refGene;
```

Este procedimiento no tiene parámetros () y muestra por pantalla el número de genes diferentes que encontramos en el campo **name2** de la tabla **refGene**. Al ejecutar el código, el procedimiento se almacena en la base de datos como un objeto más, como las tablas, y no se ejecuta la consulta `SELECT` que contiene hasta que lo llamemos.

Para llamar el procedimiento **numGenes** escribimos:

```
CALL numGenes();
```

Si queremos ver los procedimientos almacenados a la base de datos **catalogo** escribimos:

```
SHOW PROCEDURE STATUS WHERE db = 'catalogo';
```

Si queremos eliminar un procedimiento creado escribimos:

```
DROP PROCEDURE IF EXISTS numGenes;
```

2. Ahora vamos a utilizar variables, el resultado del **SELECT** lo guardaremos en la variable local **gens** que tenemos que declarar y de la cual debemos definir el tipo de datos.

Como en el procedimiento hay sentencias que acaban en «;» primero asignaremos un nuevo delimitador. Escribimos:

```
DELIMITER //
```

Ahora MySQL interpretará que el final de las sentencias SQL es el símbolo //

Crearemos un nuevo procedimiento llamado **numGenes2**.

Para declarar la variable **gens** necesitamos poner `BEGIN` antes de `DECLARE` y acabar con `END`.

Al declarar las variables locales es necesario especificar qué tipo de dato van a guardar, INT, CHAR, VARCHAR, etc. En este caso es un INT.

```
DECLARE gens INT;
```

Para guardar el resultado del SELECT en la variable local **gens** hacemos:

```
INTO gens.
```

Para mostrar el contenido de la variable **gens** hacemos:

```
SELECT gens;
```

Vemos todo el código para crear el nuevo procedimiento usando variables:

```
CREATE PROCEDURE numGenes2()
BEGIN
DECLARE gens INT;
SELECT COUNT(distinct name2)
INTO gens
FROM refGene;
SELECT gens;
END //
```

Volvemos a cambiar el delimitador para poder usar «;» al finalizar la sentencia `DELIMITER ;`

Invocamos el procedimiento:

```
 CALL numGenes2();
```

3. Ahora pasaremos un parámetro al procedimiento y lo utilizaremos en la condición `WHERE` de la consulta `SELECT`.

Creamos un nuevo procedimiento llamado **numTrans** que nos servirá para contar el número de transcritos según el tipo de transcrito que le pasamos como parámetro.

Al igual que las variables locales, es necesario especificar a los parámetros qué tipo de datos esperan, INT, CHAR, VARCHAR... En este caso, `CHAR(50)`.

El parámetro lo ponemos junto al nombre del procedimiento entre paréntesis `numTrans(transcritType CHAR(50))`.

Asignamos otra vez un delimitador //:

```
DELIMITER //
```

Veamos el código:

```
CREATE PROCEDURE numTrans ( transcritType CHAR(50))
BEGIN
DECLARE numT INT;
SELECT COUNT(*) INTO numT FROM refGene
WHERE name LIKE transcritType;
```

```
SELECT numT;
END //
```

Cambiamos otra vez el delimitador:

```
DELIMITER ;
```

Invocamos el procedimiento pasándole el parámetro NR:

```
CALL numTrans('%NR%');
```

Ahora invocamos el procedimiento pasándole el parámetro NM:

```
CALL numTrans('%NM%');
```

Al usar el comodín % nos aseguramos que no perdemos ningún registro con el contenido del parámetro.

4. En vez de declarar una variable local dentro de un procedimiento, también podemos utilizar un parámetro como variable de salida. Indicamos que es un parámetro de salida con la cláusula OUT. Por defecto, los parámetros son solo de entrada, pero también los podemos indicar con la cláusula IN.

Asignamos un delimitador //:

```
DELIMITER //
```

Creamos un nuevo procedimiento llamado **numTrans2** que nos servirá también para contar el número de transcritos según el tipo de transcrito que le pasamos como parámetro, y con un segundo parámetro que nos servirá para guardar el resultado de la consulta.

Veamos el código:

```
CREATE PROCEDURE numTrans2(IN transcritType CHAR(50), OUT numT INT)
BEGIN
SELECT COUNT(*) INTO numT FROM refGene
WHERE name LIKE transcritType;
END //
```

Cambiamos el delimitador:

```
DELIMITER ;
```

Invocamos el procedimiento pasándole los dos parámetros. Usamos una variable definida por el usuario con @ llamada @transcritos para guardar el valor que devuelve el **SELECT** del procedimiento:

```
CALL numTrans2('%NR%', @transcrits);
```

Hacemos un **SELECT** de la variable @transcritos que contiene el número de transcritos:

```
SELECT @transcrits;
```

Funciones

Las funciones devuelven un valor, así que, para llamar una función almacenada, en vez de hacer CALL hacemos directamente SELECT nombre_de_la_función y nos muestra el valor que devuelve la función.

Asignamos un delimitador //:

```
DELIMITER //
```

Creamos una nueva función llamada **numTrans3** que nos servirá también para contar el número de transcritos según el tipo de transcrito que le pasamos como parámetro.

En las funciones tenemos que indicar, después del nombre y de los parámetros, el tipo de dato que devuelve la función, en este caso un INT, y escribimos:

```
RETURNS INT
```

Al final de la función le indicamos la variable que queremos devolver, en este caso:

```
RETURN numT;
```

Veamos el código íntegro de la función:

```
CREATE FUNCTION numTrans3 (transcritType CHAR(50)) RETURNS INT
BEGIN
DECLARE numT INT;
SELECT COUNT(*) INTO numT FROM refGene
WHERE name LIKE transcritType;
RETURN numT;
END //
```

Cambiamos el delimitador:

```
DELIMITER ;
```

Invocamos la función pasándole el parámetro, en este caso les pasamos el parámetro NR:

```
SELECT numTrans3('%NR%');
```

Si pasamos a la función el parámetro NM:

```
SELECT numTrans3('%NM%');
```

Si queremos ver las funciones almacenadas a la base de datos **catalogo** escribimos la sentencia:

```
SHOW FUNCTION STATUS WHERE db = 'catalogo';
```

Si queremos eliminar una función almacenada, escribimos:

```
DROP FUNCTION IF EXISTS numTrans3;
```

Triggers

Un *trigger* es un objeto almacenado en la base de datos que está asociado con una tabla y que se activa cuando ocurre un evento sobre la tabla.

Los eventos que pueden ocurrir sobre la tabla son:

- INSERT. El *trigger* se activa cuando se inserta una nueva fila sobre la tabla asociada.
- UPDATE. El *trigger* se activa cuando se actualiza una fila sobre la tabla asociada.
- DELETE. El *trigger* se activa cuando se elimina una fila sobre la tabla asociada.

El *trigger* se puede activar o disparar antes (BEFORE) del evento o después (AFTER) del evento.

Como ejemplos vamos a crear dos *triggers* asociados a la tabla *genes* de nuestra base de datos **catalogo**:

Un *trigger* con el nombre de **trig_check_genes_before_insert** que se asocia a la tabla **genes**. Se activa antes de una operación de inserción. Si el nuevo valor del campo inicio que se quiere insertar es negativo, se guarda como 0. Si el nuevo valor del campo **final** que se quiere insertar es menor que el valor del campo **inicio**, se guarda el valor del campo **inicio**.

Un *trigger* con el nombre de **trig_check_genes_before_update** que se asocia a la tabla **genes**. Se activa antes de una operación de modificación. Si el nuevo valor del campo **inicio** que se quiere modificar es negativo, se guarda como 0. Si el valor del campo **final** del registro que se quiere modificar es menor que el nuevo valor que queremos actualizar del campo **inicio** se guarda como 1.

Creamos el *trigger* **trig_check_genes_before_insert**

```
DELIMITER //
CREATE TRIGGER trig_check_genes_before_insert
BEFORE INSERT
ON genes FOR EACH ROW
BEGIN IF NEW.inicio < 0 THEN SET NEW.inicio = 0;
ELSEIF NEW.inicio > NEW.final THEN SET NEW.final = NEW.inicio;
END IF;
END//
```

Al ejecutar este código de creación del *trigger,* **trig_check_genes_before_insert** queda almacenado en nuestra base de datos, y solo actuará, se activará cuando el usuario ejecute una sentencia de inserción, por ejemplo:

Cambiamos el delimitador:

```
DELIMITER ;
```

Realizamos operaciones de inserción en la tabla genes para que se dispare el *trigger* **trig_check_genes_before_insert**:

```
INSERT INTO genes (nombre, cromosoma, hebra, inicio, final,
proteina, especie) VALUES ('WASH7P', 'chrX', '+', -2527305,
2575270,'NR_033380', 'H. Sapiens');

INSERT INTO genes (nombre, cromosoma, hebra, inicio, final,
proteina, especie) VALUES ('WASH7P2', 'chrX', '+', 252730599,
2575270,'NR_033381', 'H. Sapiens');
```

La variable compuesta NEW que utilizamos en el *trigger* almacena todos los valores que insertamos en cada operación INSERT. De esta forma podemos usarla en el *trigger* sin conocer *a priori* qué valores se van a insertar.

En nuestros ejemplos la variable **NEW.inicio** contiene en el primer INSERT el valor -2527305 y en el segundo INSERT la variable **NEW.inicio** contiene el valor 252730599 y la variable **NEW.final** contiene el valor 2575270

En el primer **INSERT** se cumple NEW.inicio < 0, esta condición dispara el *trigger* y en el campo **inicio** se guarda el valor 0.

En el segundo **INSERT** se cumple NEW.inicio > NEW.final, esta condición dispara el *trigger* y en el campo **final** se guarda el valor 252730599.

Ahora creamos el *trigger* **trig_check_genes_before_update**:

```
DELIMITER //
CREATE TRIGGER trig_check_genes_before_update
BEFORE UPDATE ON genes
FOR EACH ROW
BEGIN
IF NEW.inicio < 0 THEN SET NEW.inicio = 0;
ELSEIF NEW.inicio > OLD.final THEN SET NEW.inicio = 1;
END IF;
END //
```

Al ejecutar este código de creación del *trigger*, **trig_check_genes_before_update** queda almacenado en nuestra base de datos, y solo actuará, se activará cuando el usuario ejecute una sentencia de actualización, por ejemplo:

Cambiamos el delimitador:

```
DELIMITER ;
```

Realizamos operaciones de modificación en la tabla genes para que se dispare el *trigger* **trig_check_genes_before_update**:

```
UPDATE genes SET inicio = 228748314 WHERE nombre = 'MYC';

UPDATE genes SET inicio = -47643 WHERE nombre = 'cbt';
```

En este caso, la variable compuesta **NEW** que utilizamos en el *trigger* almacena el nuevo valor que queremos actualizar en la sentencia **UPDATE**. En el primer **UPDATE** la variable NEW.inicio contiene el valor 228748314 y en el segundo **UPDATE** la variable NEW.inicio contiene el valor –47643.

En cambio, la variable compuesta **OLD** almacena todos los valores viejos ya almacenados en la tabla del registro que se quiere actualizar. En el primer **UPDATE**, la variable OLD.final contiene el valor almacenado en el campo final del registro con clave primaria **'MYC'** en la tabla **genes**, y en el segundo **UPDATE**, la variable OLD.final contiene el valor almacenado en el campo final del registro con clave primaria **'cbt'**.

En el primer **UPDATE** se cumple NEW.inicio > OLD.final, esta condición dispara el *trigger* y en el campo inicio se guarda el valor 1.

En el segundo **UPDATE** se cumple NEW.inicio < 0, esta condición dispara el *trigger* y en el campo inicio se guarda el valor 0.

En las operaciones **DELETE** solo se utiliza la variable compuesta **OLD** capaz de almacenar todos los valores del registro que se quiere eliminar y acceder a ellos con el formato OLD.nombre_campo.

2. Bases de datos NoSQL

2.1. Introducción

Las bases de datos relacionales son muy eficientes y mantienen la integridad de los datos, pero tienen limitaciones para gestionar con rapidez grandes volúmenes de información.

Las bases de datos relacionales escalan de forma vertical. Para crecer necesitan que los servidores tengan más capacidad.

MySQL es uno de los SGBD más utilizados para proyectos web, pero las nuevas aplicaciones web se caracterizan por tener que gestionar un inmenso volumen de información y gran cantidad de datos.

Para afrontar este nuevo reto de gestión de los datos, aparecieron las bases de datos no relacionales, conocidas también como NoSQL o Not Only SQL, y llamadas así porque no dependen únicamente del lenguaje estructurado SQL.

Las bases de datos no relacionales pueden escalar de forma horizontal, permiten la distribución de los procesos de trabajo y conjuntos de datos en múltiples servidores. De esta forma es posible que la escalabilidad de estas bases de datos sea prácticamente ilimitada.

Las bases de datos NoSQL se clasifican como de clave/valor, orientadas a documentos, grafos, o de familias de columnas.

En este módulo nos centraremos en las bases de datos no relacionales orientadas a documentos, concretamente a documentos en formato JSON.

2. Bases de datos NoSQL

2.2. Ficheros JSON

JavaScript Object Notation (JSON) es un formato de datos basado en texto estándar para representar datos estructurados que sigue la sintaxis de objeto de JavaScript. Aunque es muy parecido a la sintaxis de objeto literal de JavaScript, puede ser utilizado independientemente de JavaScript.

Los ficheros para almacenar datos con formato JSON cada vez son más usados en la informática y también en el ámbito de la bioinformática.

Figura 72. Ejemplo de un documento con formato JSON.
Fuente: elaboración propia.

En este formato la forma de guardar los datos es parecida al modelo **clave/valor**, donde la clave sería el nombre del campo o atributo y a continuación tenemos el valor.

Los ficheros JSON en realidad almacenan una **colección** de documentos con este formato y cada documento es la representación o la instancia de una entidad.

Si hacemos el símil con la información que almacenamos en una tabla relacional, cada fila de la tabla corresponde a un documento, y todas las filas de la tabla serían una colección de documentos.

JSON requiere utilizar comillas dobles para las cadenas y los nombres de propiedades. Las comillas simples no son válidas.

Una coma o dos puntos mal ubicados pueden producir que un archivo JSON no funcione. Se debe ser cuidadoso para validar cualquier dato que se quiera usar. Es posible validar JSON empleando una aplicación como JSONLint.

Veamos cómo se almacena la información que tenemos guardada en la tabla *genomas* en un fichero JSON.

```
 {especie:"D. Melanogaster", nombre:"Mosca de la
fruta",descripcion:"Tambien denominada del
 vinagre"} {especie:"H. Sapiens", nombre:"Humano",descripcion:"Nuestra
propia especie"}
 {especie:"M. Musculus", nombre:"Raton",descripcion:"Otro organismo
modelo"}
```

En este caso tenemos 3 documentos que corresponden a las 3 filas de la tabla *genomas*. Cada documento empieza con el símbolo {, y finaliza con el símbolo }

En cada documento se repite el nombre del campo y a continuación su valor, separados por el símbolo :

Cada combinación **campo/valor** se separa por comas.

No se describen los tipos de datos, si el valor es una cadena de caracteres se usan las comillas dobles " ", si el valor es numérico no es necesario escribirlo entre comillas.

Otra forma muy habitual de guardar la colección de documentos es dentro de un **array** con los símbolos **[ ]** y separando los diferentes documentos por comas. Se considera toda la colección de documentos como un solo objeto, pues están todos en un **array**.

```
[{especie:"D. Melanogaster", nombre:"Mosca de la
fruta",descripcion:"Tambien denominada del vinagre" }, {especie:"H.
Sapiens", nombre:"Humano",descripcion:"Nuestra propia especie" },
{especie:"M. Musculus", nombre:"Raton",descripcion:"Otro organismo
modelo" }]
```

Un campo puede guardar un **array** de valores:

```
{dias:["lunes", "jueves", "sábado"]}
```

Un campo también puede guardar otro **documento JSON.** También se llama *documento incrustado*.

```
{direccion: {calle: "Valencia", número: 334, codigo: 08012}}
```

Un campo puede guardar un **array** de **documentos JSON**.

```
{amigos:

[{nombre: "Pedro", edat: 34, telefono: 666737211},

{nombre: "Soraya", edat: 31, telefono: 666737212},

{nombre: "Arnau", edat: 29, telefono: 666737213}

]}
```

A partir de la versión 8 de MySQL es posible trabajar con documentos JSON en las tablas SQL con el tipo de datos JSON.

Podemos almacenar todo un documento JSON en un campo de la tabla y realizar consultas con el comando `JSON_EXTRACT`, actualizaciones con `JSON_REPLACE` y eliminaciones con el comando `JSON_REMOVE`.

2. Bases de datos NoSQL

2.3. El SGDB MongoDB

Aunque también es posible trabajar con los datos almacenados en formato JSON con MySQL y otros SGBD relacionales como PostgreSQL, la mejor forma de gestionar los datos almacenados en los ficheros JSON es utilizando una base de datos NoSQL como MongoDB**.**

El SGBD MongoDB se publicó en el año 2009 y permite gestionar bases de datos orientadas a documentos. Guarda los documentos en BSON, que no es más que una implementación binaria del formato JSON.

MongoDB es la más popular de las bases de datos NoSQL. Básicamente, devuelve datos en JSON e incorpora los conceptos de colecciones (en lugar de tablas) y documentos (en lugar de filas), su API o lenguaje de consulta se conoce popularmente como MQL (MongoDB Query Language).

Para que tengamos más claro las diferencias entre el modelo relacional y MongoDB podemos consultar la tabla 4:

Tabla 4. Comparativa entre el modelo relacional y MongoDB.

Modelo relacional	MongoDB
Database	Database
Table	Collection
Register	Document o BSON document
Columna	Field
Index	Index
Table joins	Embedded documents and linking
Primary key	Primary key
Specify any unique column or column combination as primary key	the primary key is automatically set to the_id field
Aggregation	Aggregation pipeline

Fuente: elaboración propia.

Básicamente, la diferencia más sustancial es que mientras en un SGBD relacional como MySQL tenemos bases de datos, tablas y columnas de las tablas, en MongoDB y SGBD NoSQL basados en documentos tenemos también bases de datos, pero en vez de tablas con columnas tenemos colecciones de documentos, y en cada documento tenemos los nombres de los campos en vez de las columnas de las tablas.

2. Bases de datos NoSQL

2.4. Empezar a trabajar con el SGBD MongoDB

En la máquina virtual proporcionada por la UOC tenemos instalado un SGBD MongoDB.

Para conectarnos al servidor de MongoDB abrimos un terminal, escribimos `mongosh` y nos saldrá el cursor >

```
student@ubuntuM0151:~$ mongosh
Current Mongosh Log ID: 66f3c526cf082f27c4964032
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.
3.1
Using MongoDB:          7.0.14
Using Mongosh:          2.3.1

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2024-09-25T10:08:11.986+02:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. Se
e http://dochub.mongodb.org/core/prodnotes-filesystem
   2024-09-25T10:08:17.269+02:00: Access control is not enabled for the database. Read and write access to data and confi
guration is unrestricted
------
```

Figura 73. Ejemplo de conexión a MongoDB por terminal.
Fuente: elaboración propia.

Ya estamos conectados al servidor con el cliente de la línea de comandos y ya podremos escribir las órdenes y sentencias para interactuar con el servidor de MongoDB.

Para visualizar las bases de datos creadas:

`show dbs`

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
```

Figura 74. Mostrar las bases de datos creadas.
Fuente: elaboración propia.

Como aún no hemos creado ninguna base de datos, solo nos muestra las bases de datos del sistema.

Para crear una nueva base de datos vacía usamos la orden `use`.

Vamos a generar la base de datos **uoc**.

```
use uoc
```

La orden **use** sirve tanto para crear una base de datos nueva como para conectarse a una base de datos existente. La orden intenta conectarse a una base de datos existente y si no existe la crea.

Esta es una característica general de MongoDB, es muy flexible y da pocos errores. En otros sistemas de gestión de bases de datos como MySQL, al intentar conectar con una base de datos inexistente saltaría un error. En MongoDB no salta error y se crea la nueva base de datos. En realidad, no la crea, guarda espacio para crear en ella colecciones de documentos. Si generamos una nueva base de datos con el comando **use** y no creamos colecciones en ella, no queda almacenada.

Vemos en esta secuencia de comandos cómo consultamos las bases de datos existentes en el sistema con `show dbs`, generamos la base de datos **uoc** con `use uoc`, y cómo cuando volvemos a consultar las bases de datos con `show dbs` no

aparece la base de datos **uoc**.

```
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
> use uoc
switched to db uo
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
```

Figura 75. Utilizar una base de datos.
Fuente: elaboración propia.

Para que la nueva base de datos quede guardada en el sistema es necesario que tenga colecciones de documentos.

Vamos a crear ahora una nueva colección vacía en la base de datos **uoc** utilizando la instrucción `createCollection()`.

```
> use uoc
switched to db uoc
> db.createCollection(chr1);
uncaught exception: ReferenceError: chr1 is not defined :
@(shell):1:1
> db.createCollection("chr1");
{ "ok" : 1 }
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
uoc      0.000GB
>
```

Figura 76. Crear una colección de documentos.
Fuente: elaboración propia.

Volvemos a conectarnos a la base de datos **uoc** con `use uoc` y creamos la colección de documentos vacía llamada **chr1**.

Si os fijáis, primero ejecutamos la instrucción o función `db.createCollection(chr1)`; pero da error porque el parámetro que espera la función no está entre comillas simples: **«chr1»**.

Al escribir el parámetro entre comillas simples `db.createCollection("chr1")`; la nueva colección llamada **chr1** se crea correctamente en la base de datos **uoc**, que ya no está vacía, y al ejecutar `show dbs` ya nos muestra la base de datos **uoc**.

Vamos a analizar la instrucción `db.createCollection("chr1")`; y nos servirá para entender cómo funcionan las instrucciones en MongoDB. Como ya estamos en la base de datos **uoc**, con la parte de la instrucción `db….` se refiere a la base

de datos que estamos usando, y a continuación la instrucción `createCollection()` que en realidad es una función que puede usar parámetros entre paréntesis.

Como ya veremos, la mayoría de las instrucciones son funciones que pueden usar o no parámetros. Si la instrucción no necesita parámetros dejaremos los paréntesis vacíos **()**.

Vamos a ver algunos **comandos útiles** de MongoDB:

- `db.help()` Muestra la ayuda para los métodos de la base de datos.
- `db.<collection>.help()` Muestra la ayuda para los métodos de la colección. La <collection> puede ser el nombre de una colección ya creada o no creada.
- `Show db` Muestra la lista de todas las bases de datos del sistema.
- `use db` Cambia de la base de datos actual a la base de datos <db>.
- `show collections` Muestra la lista de todas las colecciones de la base de datos actual.
- `show users` Muestra la lista de todos los usuarios de la base de datos actual.
- `show databases` Nuevo a partir de la versión 2.4. Muestra la lista de todas las bases de datos disponibles.
- `db.<collection>.find()` Muestra todos los documentos de la colección <collection>.
- `db.<collection>.find().pretty()` Muestra todos los documentos de la colección <collection> en un formato JSON más leíble.

Para ver las colecciones creadas en la base de datos usamos la instrucción siguiente:

```
show collections
```

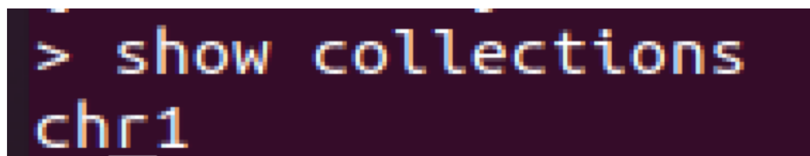


Figura 77. Mostrar las colecciones de documentos.
Fuente: elaboración propia.

2. Bases de datos NoSQL

2.5. Insertar documentos

Como tenemos la colección **chr1** vacía vamos a insertar documentos:

Vamos a insertar el siguiente documento con información del gen **uc001aaa.3**, localizado en el cromosoma 1:

```
{ "name": "uc001aaa.3", "chrom": "chr1", "strand": "+", "txStart":
11873, "txEnd": 14409, "cdsStart": 11873, "cdsEnd": 11873,
"exonCount": 3, "exonStarts": "11873,12612,13220,", "exonEnds":
"12227,12721,14409,", "proteinID": "", "alignID": "uc001aaa.3"}
```

Para insertar este documento en la colección **chr1** utilizaremos la instrucción `db.chr1.insert()`; donde se indica con db la base de datos a la que estamos conectados, `db.chr1`… , la colección donde insertamos el nuevo documento, la colección **chr1,** y la función `insert()` que espera como parámetro el documento que vamos a insertar:

```
db.chr1.insert({ "name": "uc001aaa.3", "chrom": "chr1", "strand": "+",
"txStart": 11873, "txEnd": 14409, "cdsStart": 11873, "cdsEnd": 11873,
"exonCount": 3, "exonStarts": "11873,12612,13220,", "exonEnds":
"12227,12721,14409,", "proteinID": "", "alignID": "uc001aaa.3"});
```

Si la inserción se ha realizado correctamente el sistema nos muestra el siguiente mensaje:

```
WriteResult({ "nInserted" : 1 })
```

Ahora vamos a insertar dos documentos más. Los documentos JSON con información sobre los genes **uc010nxr.1** y **uc009vit.3**, también localizados en el cromosoma 1.

```
db.chr1.insert({ "name": "uc010nxr.1", "chrom": "chr1", "strand": "+",
"txStart": 11873, "txEnd": 14409, "cdsStart": 11873, "cdsEnd": 11873,
"exonCount": 3, "exonStarts": "11873,12645,13220,", "exonEnds":
"12227,12697,14409,", "proteinID": "", "alignID": "uc010nxr.1"});


db.chr1.insert({ "name": "uc009vit.3", "chrom": "chr1", "strand": "-",
"txStart": 14361, "txEnd": 19759, "cdsStart": 14361, "cdsEnd": 14361,
"exonCount": 9, "exonStarts":
"14361,14969,15795,16606,16857,17232,17914,18267,18912,",
"exonEnds":
"14829,15038,15947,16765,17055,17742,18061,18366,19759,",
"proteinID": "", "alignID": "uc009vit.3"});
```

Ahora vamos a eliminar toda la colección **chr1** de documentos con la instrucción `db.chr1.drop()`;

Y la volvemos a crear:

```
db.createCollection("chr1");
```

para insertar los tres documentos a la vez con la instrucción `insertMany();`

```
> db.chr1.drop();
true
> db.createCollection("chr1");
{ "ok" : 1 }
> show collections
chr1
```

Figura 78. Eliminar una colección de documentos.
Fuente: elaboración propia.

En la parte de ficheros JSON vimos que muchas veces encontramos un fichero con una colección de documentos que están todos en un *array*.

Los tres documentos con la información de los genes de los ejemplos anteriores los tenemos ahora dentro de un *array*:

```
[{ "name": "uc001aaa.3", "chrom": "chr1", "strand": "+", "txStart":
11873, "txEnd": 14409, "cdsStart": 11873, "cdsEnd": 11873,
"exonCount": 3, "exonStarts": "11873,12612,13220,", "exonEnds":
"12227,12721,14409,", "proteinID": "", "alignID": "uc001aaa.3"},
{"name": "uc010nxr.1", "chrom": "chr1", "strand": "+", "txStart":
11873, "txEnd": 14409, "cdsStart": 11873, "cdsEnd": 11873,
"exonCount": 3, "exonStarts": "11873,12645,13220,", "exonEnds":
"12227,12697,14409,", "proteinID": "", "alignID": "uc010nxr.1"},
{"name": "uc009vit.3", "chrom": "chr1", "strand": "-", "txStart":
14361, "txEnd": 19759, "cdsStart": 14361, "cdsEnd": 14361,
"exonCount": 9, "exonStarts":
"14361,14969,15795,16606,16857,17232,17914,18267,18912,",
"exonEnds": "14829,15038,15947,16765,17055,17742,18061,18366,19759,",
"proteinID": "", "alignID": "uc009vit.3"}]
```

Y los vamos a inserir en la colección **chr1** con la instrucción `insertMany()`;

```
db.chr1.insertMany([{ "name": "uc001aaa.3", "chrom": "chr1",
"strand": "+", "txStart": 11873, "txEnd": 14409, "cdsStart": 11873,
"cdsEnd": 11873, "exonCount": 3, "exonStarts": "11873,12612,13220,",
"exonEnds": "12227,12721,14409,", "proteinID": "", "alignID":
"uc001aaa.3"}, {"name": "uc010nxr.1", "chrom": "chr1", "strand": "+",
"txStart": 11873, "txEnd": 14409, "cdsStart": 11873, "cdsEnd": 11873,
"exonCount": 3, "exonStarts": "11873,12645,13220,", "exonEnds":
"12227,12697,14409,", "proteinID": "", "alignID": "uc010nxr.1"},
{"name": "uc009vit.3", "chrom": "chr1", "strand": "-", "txStart":
14361, "txEnd": 19759, "cdsStart": 14361, "cdsEnd": 14361,
"exonCount": 9, "exonStarts":
"14361,14969,15795,16606,16857,17232,17914,18267,18912,",
"exonEnds":
"14829,15038,15947,16765,17055,17742,18061,18366,19759,",
"proteinID": "", "alignID": "uc009vit.3"}]);
```

Ya tenemos otra vez los tres documentos en la colección **chr1** de la base de datos **uoc,** y vamos a ver algunas instrucciones básicas para gestionar la información con MongoDB.

2.6. Buscar documentos

Primero contaremos cuántos documentos tiene la colección **chr1** con la instrucción

```
db.chr1.find().count();
```

Esta instrucción contiene dos funciones. La función `find()` que es la función que utilizamos para realizar búsquedas en la colección de documentos. Sería la instrucción `SELECT` que hemos visto en MySQL. Si no añadimos parámetros nos mostrará todos los documentos de la colección, si añadimos parámetros, estos serán los criterios de búsqueda o los filtros de la selección. En este caso complementamos la función `find()` con la función `count()` para contar los documentos encontrados. La función `count()` no requiere parámetros, pero es necesario escribir los paréntesis igualmente.

```
> db.chr1.find().count();
3
```

Figura 79. Contar todos los documentos de una colección.
Fuente: elaboración propia.

Al ejecutar la instrucción `db.chr1.find().count();` el sistema nos devuelve 3. La colección **chr1** tiene 3 documentos.

Vamos a utilizar la instrucción db.chr1.find(); para ver todos los documentos:

```
> db.chr1.find();
{ "_id" : ObjectId("64861afc32e0aa299185905c"), "name" : "uc001aaa
arts" : "11873,12612,13220,", "exonEnds" : "12227,12721,14409,", "
{ "_id" : ObjectId("64861df75d6ce7147d325b7e"), "name" : "uc010nxr
arts" : "11873,12645,13220,", "exonEnds" : "12227,12697,14409,", "
{ "_id" : ObjectId("64861e025d6ce7147d325b7f"), "name" : "uc009vit
arts" : "14361,14969,15795,16606,16857,17232,17914,18267,18912,",
>
```

Figura 80. Mostrar la información de todos los documentos de una colección.
Fuente: elaboración propia.

Para cada documento, el sistema nos muestra el campo **_id** con un valor generado aleatoriamente por la función del sistema `ObjectId()`;

En el campo **_id** se guarda la clave primaria del documento. Esta clave primaria es tan importante para el sistema que si el documento que insertamos no tiene el campo **_id** el sistema lo crea de forma automática. En nuestros ejemplos los tres documentos inseridos no tienen el campo **_id** y el sistema lo ha generado automáticamente.

En un documento puede haber muchos campos y, tal como los muestra la función `find();` , quedan poco legibles. Para que el documento se pueda leer mejor, la función `find();` se puede complementar con la función `pretty();`

```
db.chr1.find().pretty();
```

```
> db.chr1.find().pretty();
{
        "_id" : ObjectId("64861afc32e0aa299185905c"),
        "name" : "uc001aaa.3",
        "chrom" : "chr1",
        "strand" : "+",
        "txStart" : 11873,
        "txEnd" : 14409,
        "cdsStart" : 11873,
        "cdsEnd" : 11873,
        "exonCount" : 3,
        "exonStarts" : "11873,12612,13220,",
        "exonEnds" : "12227,12721,14409,",
        "proteinID" : "",
        "alignID" : "uc001aaa.3"
}
{
        "_id" : ObjectId("64861df75d6ce7147d325b7e"),
        "name" : "uc010nxr.1",
        "chrom" : "chr1",
        "strand" : "+",
        "txStart" : 11873,
        "txEnd" : 14409,
        "cdsStart" : 11873,
        "cdsEnd" : 11873,
        "exonCount" : 3,
        "exonStarts" : "11873,12645,13220,",
        "exonEnds" : "12227,12697,14409,",
        "proteinID" : "",
        "alignID" : "uc010nxr.1"
}
```

Figura 81. Mostrar la información de todos los documentos de una colección con un formato más legible.
Fuente: elaboración propia.

Como podemos observar, la función pretty(); nos permite ver los valores de cada campo de cada documento de una forma mucho más amigable.

ahora algunos ejemplos de utilización de la función find() con algunos parámetros para filtrar los resultados.

Vamos a mostrar solo los genes de nuestra colección que se encuentra en la hebra +:

```
db.chr1.find({"strand": "+"}).pretty();
```

```
> db.chr1.find({"strand": "+"}).pretty();
{
        "_id" : ObjectId("6487c3e05d6ce7147d325b83"),
        "name" : "uc001aaa.3",
        "chrom" : "chr1",
        "strand" : "+",
        "txStart" : 11873,
        "txEnd" : 14409,
        "cdsStart" : 11873,
        "cdsEnd" : 11873,
        "exonCount" : 3,
        "exonStarts" : "11873,12612,13220,",
        "exonEnds" : "12227,12721,14409,",
        "proteinID" : "",
        "alignID" : "uc001aaa.3"
}
{
        "_id" : ObjectId("6487c410235597cf4bc92782"),
        "name" : "uc010nxr.1",
        "chrom" : "chr1",
        "strand" : "+",
        "txStart" : 11873,
        "txEnd" : 14409,
        "cdsStart" : 11873,
        "cdsEnd" : 11873,
        "exonCount" : 3,
        "exonStarts" : "11873,12645,13220,",
        "exonEnds" : "12227,12697,14409,",
        "proteinID" : "",
        "alignID" : "uc010nxr.1"
}
```

Figura 82. Mostrar la información de los genes de la hebra +.
Fuente: elaboración propia.

El parámetro de la función es la condición de selección de la búsqueda `{"strand": "+"}`

Si no queremos mostrar algún campo, como por ejemplo el campo `_id`, escribimos

```
db.chr1.find({"strand": "+"}, {"_id":0}).pretty();
```

```
}
> db.chr1.find({"strand": "+"}, {"_id":0}).pretty();
{
        "name" : "uc001aaa.3",
        "chrom" : "chr1",
        "strand" : "+",
        "txStart" : 11873,
        "txEnd" : 14409,
        "cdsStart" : 11873,
        "cdsEnd" : 11873,
        "exonCount" : 3,
        "exonStarts" : "11873,12612,13220,",
        "exonEnds" : "12227,12721,14409,",
        "proteinID" : "",
        "alignID" : "uc001aaa.3"
}
{
        "name" : "uc010nxr.1",
        "chrom" : "chr1",
        "strand" : "+",
        "txStart" : 11873,
        "txEnd" : 14409,
        "cdsStart" : 11873,
        "cdsEnd" : 11873,
        "exonCount" : 3,
        "exonStarts" : "11873,12645,13220,",
        "exonEnds" : "12227,12697,14409,",
        "proteinID" : "",
        "alignID" : "uc010nxr.1"
}
```

Figura 83. Mostrar la información de los genes de la hebra + sin el campo _id.
Fuente: elaboración propia.

Si queremos mostrar un solo campo de los documentos seleccionados, como por ejemplo el campo **name**, escribimos

```
db.chr1.find({"strand": "+"},{"name":1, "_id":0 }).pretty();
```

```
> db.chr1.find({"strand": "+"},{"name":1, "_id":0 }).pretty();
{ "name" : "uc001aaa.3" }
{ "name" : "uc010nxr.1" }
```

Figura 84. Mostrar los genes de la hebra + sin mostrar el campo _id y mostrar el campo name.
Fuente: elaboración propia.

Las consultas con la función `find()` pueden ser mucho más elaboradas, usando varios campos como condición de búsqueda, o también usando operadores especiales como mayor que, menor que, etc.

Ejemplos de operadores especiales:

```
$gt, $gte, $lt, $lte, $ne, $in, $nin, $mod, $regex/$options,
```

```
$all, $size, $exists, $type, $not, $or, $nor, $elemMatch
```

Ejemplo de selección de documentos en los que el campo **txStart** sea mayor que 11873:

```
db.chr1.find({"txStart": {"$gt" : 11873 }}).pretty();
```

```
> db.chr1.find({"txStart": {"$gt" : 11873 }}).pretty();
{
        "_id" : ObjectId("6487c41c235597cf4bc92783"),
        "name" : "uc009vit.3",
        "chrom" : "chr1",
        "strand" : "-",
        "txStart" : 14361,
        "txEnd" : 19759,
        "cdsStart" : 14361,
        "cdsEnd" : 14361,
        "exonCount" : 9,
        "exonStarts" : "14361,14969,15795,16606,16857,17232,17914,18267,18912,",
        "exonEnds" : "14829,15038,15947,16765,17055,17742,18061,18366,19759,",
        "proteinID" : "",
        "alignID" : "uc009vit.3"
}
>
```

Figura 85. Mostrar los genes que el campo txStar sea mayor que 11873.
Fuente: elaboración propia.

O que cumpla dos condiciones, que se encuentren en la hebra + y que el campo **txStart** sea mayor o igual a 11873.

```
db.chr1.find({"strand": "+", "txStart": {"$gte" : 11873
}}).pretty();
```

2.7. Modificar documentos

Veamos ahora cómo se puede actualizar un documento con la función `update()`.

Vamos a modificar la hebra del gen uc001aaa.3.

```
db.chr1.update({"name" : "uc001aaa.3"}, { "$set" : { "strand": "-" }});
```

Y comprobamos el cambio:

```
db.chr1.find({"name" : "uc001aaa.3"},{"name":1,"strand":1,"_id":0
}).pretty();
```

```
> db.chr1.update({"name" : "uc001aaa.3"}, { "$set" : { "strand": "-" }});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.chr1.find({"name" : "uc001aaa.3"},{"name":1,"strand":1,"_id":0 }).pretty();
{ "name" : "uc001aaa.3", "strand" : "-" }
>
```

Figura 86. Modificar la hebra del gen uc001aaa.3.
Fuente: elaboración propia.

El operador `$set` modifica el valor de campo si este existe; si no existe el campo, lo incorpora al documento o documentos que coincidan con la selección.

El operador `$unset` elimina el campo del documento o documentos que coincidan con la selección.

También es posible añadir nuevos elementos a un campo *array* con el operador `$push` y eliminar elementos del *array* con los operadores `$pull`, `$pullAll`, `$pop`.

Los operadores del comando update son los siguientes:

```
$set, $unset, $inc, $push, $pushAll, $pull, $pullAll, $pop,
```

```
$addToSet, $rename, $bit, $ positional operator
```

2.8. Eliminar documentos

Veamos ahora cómo podemos eliminar un documento de la colección con la función `remove()`.

Eliminamos de la colección el documento con el nombre de gen **uc001aaa.3**

```
db.chr1.remove( { "name" : "uc001aaa.3"});
```

Y comprobamos si el documento se ha eliminado:

```
db.chr1.find().pretty();
```

```
> db.chr1.remove( { "name" : "uc001aaa.3"});
WriteResult({ "nRemoved" : 1 })
> db.chr1.find().pretty();
{
        "_id" : ObjectId("6487c410235597cf4bc92782"),
        "name" : "uc010nxr.1",
        "chrom" : "chr1",
        "strand" : "+",
        "txStart" : 11873,
        "txEnd" : 14409,
        "cdsStart" : 11873,
        "cdsEnd" : 11873,
        "exonCount" : 3,
        "exonStarts" : "11873,12645,13220,",
        "exonEnds" : "12227,12697,14409,",
        "proteinID" : "",
        "alignID" : "uc010nxr.1"
}
{
        "_id" : ObjectId("6487c41c235597cf4bc92783"),
        "name" : "uc009vit.3",
        "chrom" : "chr1",
        "strand" : "-",
        "txStart" : 14361,
        "txEnd" : 19759,
        "cdsStart" : 14361,
        "cdsEnd" : 14361,
        "exonCount" : 9,
        "exonStarts" : "14361,14969,15795,16606,16857,17232,17914,18267,18912,",
        "exonEnds" : "14829,15038,15947,16765,17055,17742,18061,18366,19759,",
        "proteinID" : "",
        "alignID" : "uc009vit.3"
}
```

Figura 87. Eliminar el gen uc001aaa.3.
Fuente: elaboración propia.

Ahora vamos a eliminar toda la colección con el comando `drop`

```
db.dropDatabase();
```

2. Bases de datos NoSQL

2.9. Importar ficheros JSON a MongoDB

Vamos a trabajar ahora importando un fichero en formato JSON que nos proporciona el navegador genómico UCSC.

Descargamos el fichero JSON del siguiente enlace:
https://api.genome.ucsc.edu/getData/track?genome=hg19;track=knownGene;chrom=chr1

Este fichero contiene la información de todos los genes conocidos del cromosoma 1.

Bajamos el fichero y lo guardamos con el nombre `hg19chr1.json`

Para importar el fichero a MongoDB abrimos un nuevo terminal en la carpeta de la máquina virtual donde hemos guardado el fichero y escribimos

```
mongoimport --db hg19 --collection chr1 --drop --file hg19chr1.json
```

Vamos a analizar esta instrucción:

El comando `mongoimport` se ejecuta **fuera** del cliente (mongo) y tiene varias opciones o parámetros

- `--db` indica la base de datos. Si existe la utiliza para crear la colección de documentos, si no existe la crea.
- `--collection db` indica la colección. Si existe, la utiliza para insertar en ella los documentos del fichero que importamos, si no existe la crea.
- `--drop elimina` los documentos previos de la colección, si existe.
- `--file` indica el fichero que vamos a importar. Si los documentos estuvieran dentro de un *array* tenemos que añadir la opción `--jsonArray`

Si nos conectamos ahora al servidor de MongoDB con el cliente mongo y miramos las bases de datos con `show dbs`, vemos cómo se ha creado la base de datos **hg19.**

Si nos conectamos a la base de datos **hg19** con `use hg19` y vemos las colecciones con `show collections` podemos ver la colección **chr1** que hemos creado y ya podemos trabajar con ella.

2.10. Buscar en un *array* de documentos

Si nos fijamos en la estructura del fichero JSON importado, solo contiene un documento JSON. Y en uno de los campos, el llamado **knownGene** contiene un *array* de documentos JSON, y cada documento del *array* contiene información de los genes del cromosoma 1: nombre, hebra, cromosoma, etc.

Esto nos obliga a conocer cómo se trabaja con los contenidos de los *arrays* si queremos gestionar correctamente esta información.

Tenemos que usar el `dot.notation`.

Por ejemplo, vamos a mostrar la información del gen `uc009vis.3`.

Esta es la instrucción:

```
db.chr1.find({ "knownGene.name": "uc009vis.3"}, {"knownGene.$":
1}).pretty();
```

```
> db.chr1.find(
... { "knownGene.name": "uc009vis.3"}, {"knownGene.$": 1}
... ).pretty();
{
        "_id" : ObjectId("6487db999faf672901337f0a"),
        "knownGene" : [
                {
                        "name" : "uc009vis.3",
                        "chrom" : "chr1",
                        "strand" : "-",
                        "txStart" : 14361,
                        "txEnd" : 16765,
                        "cdsStart" : 14361,
                        "cdsEnd" : 14361,
                        "exonCount" : 4,
                        "exonStarts" : "14361,14969,15795,16606,",
                        "exonEnds" : "14829,15038,15942,16765,",
                        "proteinID" : "",
                        "alignID" : "uc009vis.3"
                }
        ]
}
>
```

Figura 88. Mostrar la información del gen uc009vis.3.
Fuente: elaboración propia.

Vemos como, para referirnos al nombre del gen, escribimos `"knownGene.name":` o sea, el nombre del campo del documento que es un *array* de documentos JSON, el `knownGene`, punto y a continuación el nombre del campo de los documentos que se encuentran en el *array* `name`. Con `knownGene.$": 1` indicamos que nos muestre todos los campos de los documentos encontrados en el *array* `knownGene`.

El `dot.notation` también nos sirve para referirnos a campos de documentos embebidos.

2. Bases de datos NoSQL

2.11. Agregaciones en MongoDB

Para trabajar a fondo con los elementos de un *array* es mejor utilizar el `framework aggregation` que nos permite MongoDB muchas opciones y realizar tuberías *pipelines* para gestionar la información almacenada.

Operaciones de agregaciones

Las operaciones de agregación son herramientas de MQL que nos ayudan a procesar documentos y a retornar resultados calculados. Las operaciones de agregación se utilizan mayoritariamente para:

- Agrupar valores de varios documentos.
- Procesamiento y operaciones para el retorno de resultados.
- Analizar cambios de datos a lo largo del tiempo.

Tuberías y transformaciones

Para realizar el procesamiento de documentos, MongoDB se basa en el patrón de filtro de tubería, utilizado comúnmente en arquitecturas de software. Este patrón consta de una o más etapas, en donde cada etapa efectúa una operación con los datos de entrada, y la salida o resultado se la entrega a la siguiente etapa para su procesamiento.

Para aplicar este patrón, MongoDB utiliza una serie de operadores ya definidos para poder procesar documentos.

Los operadores más utilizados en tuberías:

- Filtrado de documentos con criterios: `$match`.
- Orden de documentos: `$sort`.
- Selección de campos en específico: `$project`.
- Agrupación de documentos: `$group`.
- Sacar los elementos de un *array*: `$unwind`.

Sería necesario otro curso para profundizar en todas las opciones que nos ofrece MongoDB.

Veamos un ejemplo de una operación de agregación. La función `aggregate()`.

Entre muchas otras opciones, `aggregate` nos permite sacar los elementos de un *array* con el operador `$unwind`, hacer agrupaciones con `$group` y contar con `$sum`.

Vamos a contar cuántos genes contiene cada hebra del ADN del cromosoma 1.

```
db.getCollection('chr1').aggregate([{$unwind:"$knownGene"},{$group:
{_id:"$knownGene.strand",cantidad:{$sum:1}}}])
```

```
> db.getCollection('chr1').aggregate([{$unwind:"$knownGene"},{$group:{_id:"$knownGene.strand",cantidad:{$sum:1}}}]);
{ "_id" : "-", "cantidad" : 3894 }
{ "_id" : "+", "cantidad" : 4073 }
>
>
>
```

Figura 89. Mostrar cuántos genes contiene cada hebra del ADN del cromosoma 1.
Fuente: elaboración propia.

Resumen

A nivel genómico, habitualmente trabajamos con miles de registros durante un análisis bioinformático convencional. Pese a que las herramientas de análisis de ficheros de texto basadas en el uso de comandos del terminal de GNU/Linux nos permiten acceder fácilmente a nuestros datos, los sistemas de gestión de bases de datos como MySQL o MongoDB resultan el medio ideal para gestionar grandes volúmenes de datos de forma eficiente y rápida.

En este módulo os hemos mostrado un gran abanico de comandos basados en el lenguaje SQL para consultar las tablas de nuestras bases de datos relacionales y algunos comandos específicos de MongoDB para gestionar la información de una base de datos no relacional NoSQL orientada a documentos.

Junto con este inventario de instrucciones, hemos aprendido también a modelar correctamente las entidades del problema real que deseamos aproximar mediante estrategias bioinformáticas.

Actividades

1. Llevamos a la práctica sobre tu propio SGBD MySQL los ejemplos mostrados durante este módulo. Intentamos enriquecer cada base de datos con nuevas tablas que modelen entidades o relaciones que no aparecían originalmente en los casos estudiados. Ampliamos el conjunto de atributos de las tablas para describir con más precisión las instancias reales que se muestran a lo largo del texto.

2. Diseñamos una base de datos para almacenar información sobre un entorno complejo natural que te gustaría modelar: un ecosistema, el cuerpo humano, la célula o el universo. Reflexionamos sobre las entidades, sus atributos y las relaciones entre estas, que son necesarias en cada caso para especificar dicho modelo. Repetimos el ejercicio con un entorno generado por el ser humano (por ejemplo, un automóvil).

3. Implementamos una base de datos en MySQL que permita llevar el registro de todas las actividades de un servidor web genérico que recibe peticiones para ejecutar una serie de servicios: páginas más visitadas, tipo de servicios solicitado, volumen de datos, tiempo de respuesta, usuarios, número de páginas consultadas por cliente, dirección IP y nombre de la máquina, país de procedencia, etc.

4. Ampliamos nuestro catálogo de genes pensando en futuras ampliaciones. Podemos añadir una entidad PROTEINAS para almacenar información estructural o sobre dominios funcionales. También sería interesante incorporar una entidad CROMOSOMAS, relacionada con la tabla GENOMAS, donde guardar otras características, como el número de bases o el contenido en G+C.

5. Importamos algunos ficheros JSON que nos proporciona UCSC Genome Browser https://genome.ucsc.edu/goldenPath/help/api.html#REST a MongoDB creando para cada uno de ellos una base de datos nueva y una nueva colección. Realizamos alguna consulta simple a las colecciones y alguna consulta en un documento situado en un campo *array* de documentos.

Ejercicios de autoevaluación

1. Definid el modelo entidad-relación en el diseño de bases de datos.
2. Describid las diferencias entre bases de datos y gestores de base de datos.
3. Describid qué es una tabla en el modelo relacional.
4. ¿Qué son las claves primarias?
5. ¿Qué son las claves foráneas?
6. Definid qué es una relación 1:N.
7. Recordad qué símbolo debéis emplear siempre al final de cada comando SQL.
8. Mencionad cinco comandos esenciales del lenguaje SQL.
9. Describid el comando de SQL para explorar el contenido de las tablas.
10. ¿Qué tipo de operaciones se realizan sobre registros agrupados?
11. Diferenciad entre los comandos DISTINCT y LIMIT.
12. Enumerad los tres tipos de uniones a realizar entre dos tablas.
13. Definid la utilidad de una subconsulta.
14. ¿Qué dos comandos son útiles para borrar tablas, en qué se diferencian?
15. Indicad la instrucción de SQL para ejecutar un fichero de comandos.
16. Indicad la instrucción de SQL para cargar un fichero de datos.
17. ¿Qué formato cumplen los ficheros de texto para ser empleados de ese modo?
18. Indicad el programa de GNU/Linux que copia una base de datos en un fichero.
19. ¿Cómo puede iniciarse el cliente de MySQL desde la línea de comandos?
20. ¿Cómo se conceden autorizaciones a un usuario sobre una base de datos MySQL?
21. ¿Cómo se muestra un documento con un formato más legible?
22. ¿Cómo se crea una nueva colección de documentos en MongoDB?
23. ¿Cómo se cuentan los documentos de una colección en MongoDB?
24. ¿Qué es la sintaxis dot.notation en MongoDB?
25. ¿Qué es un documento embebido en un fichero JSON?

Solucionario

1. El modelo entidad-relación se basa en el uso de dos tipos de elementos para modelar un entorno real. Las entidades modelan cada clase de elementos de la realidad. Las relaciones modelan las asociaciones entre las instancias de cada entidad en dicho entorno.

2. Una base de datos es un conjunto de informaciones organizadas para fomentar un acceso eficiente a estas; un gestor de base de datos es precisamente el programa que implementa el mantenimiento permanente de la base de datos, ofreciendo además mecanismos para acceder a estos.

3. Una tabla es una estructura que agrupa una colección de elementos (instancias) de la misma clase. Generalmente, una tabla modela una entidad junto con sus atributos, aunque puede ser necesaria también para implementar algunas relaciones entre entidades dentro de la base de datos.

4. La clave primaria de una tabla es el atributo que identifica de forma unívoca a cada instancia o elemento en su interior. Por ello, el valor de este atributo no puede repetirse entre instancias diferentes.

5. La clave foránea de una tabla es la clave primaria de otra tabla, asegurando la integridad del modelo, dado que cada instancia en la primera tabla deberá existir también en la segunda.

6. Una relación 1:N entre dos entidades indica que cada instancia de la primera entidad puede asociarse con *N* instancias en la segunda tabla. Puede implementarse en la segunda tabla directamente con un atributo que tome por valor alguno en el rango correcto para la primera.

7. Cualquier comando que introducimos en el intérprete de MySQL debe terminar obligatoriamente con el símbolo **;**, para ser ejecutado correctamente.

8. Por ejemplo: `CREATE DATABASE`, `CREATE TABLE`, `SELECT`, `LOAD DATA` y `GRANT`.

9. El comando `SELECT...FROM...WHERE` permite realizar una consulta sobre los registros de las tablas que cumplen ciertas condiciones.

10. Cuando se agrupan instancias con **GROUP BY**, es posible calcular medias aritméticas, mínimos, máximos o cuentas de totales (**AVG**, **MIN**, **MAX**, **COUNT**).

11. La cláusula DISTINCT sirve para eliminar valores repetidos. La cláusula LIMIT es útil para mostrar solo las primeras instancias de una tabla.

12. A la hora de comparar dos tablas con el comando `JOIN` empleando un atributo en común, podemos buscar las parejas de valores presentes en ambas tablas o aquellas que solo aparecen en una de ellas (**LEFT** o **RIGHT**).

13. Una subconsulta permite generar un grupo de resultados en forma de tabla temporal. Dicha tabla auxiliar podrá ser interrogada, a su vez, por la consulta principal que alberga la subconsulta en su interior.

14. El comando `DROP TABLE` elimina la tabla completamente (definición y contenido). El comando `DELETE`, en cambio, elimina únicamente determinados registros de acuerdo a una condición.

15. La instrucción `SOURCE`.

16. La instrucción `LOAD DATA`.

17. El fichero de texto debe estar tabulado en columnas, que corresponden a los atributos de las tablas.

18. El programa **mysqldump** realiza el volcado de la base de datos.

19. El comando sería: `mysql -u usuario -p`

20. Con la instrucción `GRANT`

21. Con el comando o función `pretty()`

22. Con la función `db.createCollection("nombreColeccion");`

23. Con la función `count();`

24. Es la forma de identificar un campo situado en un documento situado en campo *array* de documentos de la forma `nombre_campo_array.nombre_campo_documento`

25. Es un documento situado en un campo de un documento JSON.

Bibliografía

Conrad Bessant, Ian Shadforth y Darren Oakley (2009). *Building Bioinformatics Solutions: with Perl, R and MySQL*. Oxford University Press. ISBN: 0199230234.

Edgar F. Codd (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13, p. 377-387.

MySQL AB (2006). *Manual de referencia de MySQL 8.0*. http://dev.mysql.com

Paul DuBois (2008). *MySQL* (4th Edition). Addison-Wesley Professional. ISBN: 0672329387.

Peter Pin-Shan Chen (1976). The Entity-relationship Model–Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1, p. 9-36.

Steven Haddock y Casey Dunn (2011). *Practical Computing for Biologists*. Sinauer Associates. ISBN: 978-0-87893-391-4.

Vince Buffalo (2015). *Bioinformatics Data Skills.* O'Reilly Media. ISBN: 978-1-449-36737-4.

Webgrafía

Anaya Multimedia. Guía Práctica MySQL 5.1/Capítulo 11: Procedimientos almacenados. https://enreas.fandom.com/wiki/Gu%C3%ADa_Pr%C3%A1ctica_MySQL_5.1/Cap%C3%ADtulo_11:_Procedimientos_almacenados

COMANDOS MYSQL. Convertir consulta MySQL a JSON. https://thedevelopmentstages.com/convertir-consulta-mysql-a-json/

DelftStack. Cómo declarar y usar las variables en MySQL. https://www.delftstack.com/es/howto/mysql/mysql-declare-variable/

Guebs. MySQL 5.0 Reference Manual. Traducción. https://manuales.guebs.com/mysql-5.0/

Guebs. MySQL 5.0 Reference Manual. 19.2.1. CREATE PROCEDURE y CREATE FUNCTION. https://manuales.guebs.com/mysql-5.0/stored-procedures.html#create-procedure

JSONLint – The JSON Validator. https://jsonlint.com/

MongoDB. https://www.mongodb.com/

MongoDB. MongoDB CRUD Operations. https://www.mongodb.com/docs/manual/crud/

MongoDB Manual. Aggregation Operations. https://www.mongodb.com/docs/manual/aggregation/

MySQL. https://dev.mysql.com/

MySQL 8.0 Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/

MySQL 8.0 Reference Manual. 13.1.20.5 FOREIGN KEY Constraints. https://dev.mysql.com/doc/refman/8.0/en/create-table-foreign-keys.html

MySQL 8.0 Reference Manual. 13.1.17 CREATE PROCEDURE and CREATE FUNCTION Statements. https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html

MySQLTUTORIAL.MySQL Stored Procedures. https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx

MySQL 8.0 Reference Manual. 25.3.1 Trigger Syntax and Examples. https://dev.mysql.com/doc/refman/8.0/en/trigger-syntax.html

MySQL 8.0 Reference Manual. 11.5 The JSON Data Type. https://dev.mysql.com/doc/refman/8.0/en/json.html

Universidad de Sevilla. Introducción a PL/SQL en MySQL. https://dam.org.es/introduccion-a-pl-sql-en-mysql/