Administración de bases de datos: MySQL 5.1

Indice

1. Introducción....3
1.1 Bases de datos relacionales....3
1.2 Conceptos básicos....3
1.3 MySQL....4
1.3.1 Sistema de versiones....5
1.3.2 Licencia....6
1.4 Arquitectura LAMP....6
2. Arquitectura....8
3. SQL y MySQL....11
3.1 Tipos de instrucciones....11
3.2 Instrucciones básicas....11
3.2.1 Consultas (SELECT)....11
3.2.2 Seleccionar registros (WHERE)....13
3.2.3 Enlazar tablas (JOIN)....13
3.2.4 Modificar datos (INSERT, UPDATE, y DELETE)....15
3.2.5 Crear bases de datos, tablas e índices....18
3.2.6 Cambiar diseño de tablas....19
3.2.7 Borrar bases de datos y tablas....19
3.3 Cambios de diseño automáticos....20
3.4 Inspeccionar meta datos (SHOW)....20
3.5 Tablas INFORMATION_SCHEMA....22
4. Características específicas de MySQL....25
4.1 Tipos de tablas....25
4.1.1 MyISAM....26
4.1.2 InnoDB....28
4.1.3 MyISAM vs InnoDB....30
4.1.4 MEMORY....31
4.1.5 NBDCLUSTER....32
4.2 Tipos de datos....33
4.2.1 Enteros (xxxINT)....33
4.2.2 Enteros AUTO_INCREMENT....34
4.2.3 Datos binarios (BIT y BOOL)....35
4.2.4 Números en coma flotante (FLOAT y DOUBLE)....35
4.2.5 Números de coma fija (DECIMAL)....36
4.2.6 Fecha y hora (DATE, TIME, DATETIME, TIMESTAMP)....36
4.2.7 Cadenas de caracteres (CHAR, VARCHAR, xxxTEXT)....37
4.2.8 Datos binarios (xxxBLOB y BIT)....40
4.2.9 Otros tipos....40
4.2.10 Opciones y atributos....41
4.2.11 Datos GIS....41
5. Modos SQL....44
6. Diseño de bases de datos....50
6.1 Normalización....51
6.2 Integridad referencial....54
7. Índices....58

7.1 Conceptos básicos....58
7.2 Estructuras de datos....62
7.3 Índices y tipos de tablas....64
7.3.1 Tablas MyISAM....64
7.3.2 Tablas HEAP....65
7.3.3 Tablas InnoDB....65
7.4 Índices Full-text....65
8. Optimización de consultas....67
8.1 Gestión interna de las consultas....67
8.1.1 Cache de consultas....67
8.1.2 Análisis sintáctico y optimización....67
8.2 EXPLAIN SELECT....68
8.3 Características del optimizador....76
8.4 Identificar consultas lentas....78
8.5 Modificar el comportamiento de MySQL....79
9. Vistas....82
10. Procedimientos Almacenados....84
10.1 Sintaxis....87
10.2 Llamadas....88
10.3 Parámetros y valores de retorno....89
10.4 Variables....90
10.5 Encapsulación de instrucciones....91
10.6 Control de flujo....91
10.7 Bucles....92
10.8 Gestión de errores (handlers)....93
10.9 Cursores....94
11. Triggers....96
12. Transacciones....100
13. Respaldos (backups)....102
14. Administración de accesos y seguridad....105
14.1 Tipos de conexiones....105
14.2 Administración de accesos....105
14.3 Creación de bases de datos, usuarios y sus privilegios....106
14.4 Base de datos 'mysql'....107

1. Introducción

1.1 Bases de datos relacionales

Una base de datos relacional es un conjunto ordenado de datos, los cuales normalmente están almacenados en uno o más ficheros. Los datos están estructurados en tablas, y estas pueden tener referencias cruzadas. La existencia de estas referencias o relaciones es lo que da el nombre de *relacional* a este tipo de bases de datos.

Por ejemplo, una base de datos puede almacenar información a cerca de los clientes de una compañía. La bases de datos estaría compuesta de una tabla de clientes (nombre, dirección, ...etc), una tabla con los productos que la empresa ofrece, y finalmente, una tabla con los pedidos de la empresa. A través de la tabla de pedidos sería posible acceder a los datos de las otras dos tablas (por ejemplo, a través de los identificadores de cliente y producto).

Como ejemplos de *sistemas* de bases de datos relacionales tenemos MySQL, Oracle, PostgreSQL, Microsoft SQL Server, IBM DB2, ... etc. Estos sistemas proporcionan herramientas que permiten almacenar los datos de manera eficiente, procesar las *queries,* analizar y ordenar los datos, ... etc. Además, todo esto ha de ser posible que funcione sobre una red, por lo que normalmente hablaremos de un servidor de bases de datos.

Como contraposición a las bases de datos relacionales tenemos las bases de datos orientadas a objetos. Estas pueden almacenar objetos individuales sin estar estructurados en tablas. Estas bases de datos permiten acceder directamente a objetos definidos en el entorno de un lenguaje de programación (PHP, C++, Java, ...). A pesar de que han habido una serie de tendencias hacia bases de datos relacionales en los últimos años, estas solo han encontrado pequeños nichos de mercado.

1.2 Conceptos básicos

Ya hemos mencionado que las tablas son las estructuras donde se guardan los datos. Cada línea de una tabla se llama registro (*data record*), y su estructura viene determinada por la definición de la tabla. Por ejemplo, en una tabla de direcciones de usuarios cada registro contendrá campos (*fields*) para el nombre, apellido, calle, ... Para cada campo hay una serie de tipos de datos predefinidos que determinan con exactitud la información que puede ser almacenada.

La descripción de una base de datos consistente en varias tablas con todos sus campos, relaciones e índices se llama un modelo de base de datos (*database model*). Este modelo define como se tienen que construir las estructuras de datos y al mismo tiempo especifica el formato en que los datos deben ser almacenados.

Normalmente, las tablas almacenan los datos sin un orden concreto, y de hecho el orden suele ser en el que han sido introducidos. Sin embargo, para poder usar los datos de una manera útil se necesita que estos puedan ser ordenados siguiendo uno o más criterios. Por ejemplo, si disponemos de una tabla de clientes, podemos querer obtener la lista de clientes que han comprado un determinado producto en los últimos 12 meses, ordenados por el código postal.

Para crear una lista así se debe ejecutar una consulta (*query*) en la base de datos. El resultado es a su vez una tabla que se aloja en memoria RAM. Estas consultas se realizan en el lenguaje SQL (*Structured Query Language*). A pesar de ser un estandard para los sistemas de bases de datos relacionales, cada fabricante ofrece sus esxrtensiones particulares SQL, con lo que el objetivo de la compatibilidad no siempre se alcanza.

Cuando las tablas crecen hasta un tamaño considerable, la velocidad de las consultas depende significativamente de si existe un índice (*index*) que proporciona el orden de los datos de un determinado campo. Concretamente, un índice es una tabla que únicamente almacena información sobre el orden de los registros de una tabla. Un índice corresponde a un campo clave (*key*).

Los índices aceleran las consultas, pero a un coste. Primero, requieren un espacio de almacenamiento extra, y segundo, deben ser actualizados cada vez que la tabla es modificada.

Las claves y los índices pueden ser primarios, y en ese caso se tienen que cumplir la condición de que haya una única referencia para cada registro de la tabla. Normalmente las claves primarias se representan con un número cuyo valor es asignado automáticamente por la base de datos cada vez se crea un registro (*autoincrement*)

1.3 MySQL

Un poco de historia:

- El lenguaje SQL fue creado por IBM en 1981. Es un estándar de los sistemas relacionales. Está aprobado como estándar ANSI/ISO desde 1987.ç
- MySQL nace como un proyecto para crear un sistema de bases de datos de software libre por parte de la empresa sueca MySQL AB en 1995
- MySQL ha sido adquirida por Sun Microsystems en enero de 2008.

De aquí en adelante se entiende que toda le información correspondiente a MySQL se refiere a la versión 5.0. De todas maneras, la mayoría de las características se pueden aplicar a otras versiones, especialmente a la próxima versión 5.1.

MySQL es un sistema de bases de datos relacional. A pesar de su popularidad, este sistema todavía no tiene algunas de las características que poseen otros sistemas comerciales. Sin embargo, estas diferencias son cada vez más pequeñas. Algunas de las compañias que usan MySQL son Yahoo!, BBC News, CNET, Nokia, YouTube, Flickr, Google, ...

Las principales característica de MySQL son:

- Sistema de base de datos relacionales
- Arquitectura cliente/servidor
- Compatibilidad con SQL. En este momento soporta SQL:2003, aunque tiene muchas extensiones
- SubSELECTS. Es capaz de procesar consultas del tipo:

```
SELECT * FROM table1 WHERE x IN (SELECT y FROM table2)
```

- Vistas (*Views*). Son consultas SQL cuyos resultados son tratados como si fueran una

tabla diferente, permitiendo tener diferentes vistas de una misma tabla.

- Stored Procedures. Son consultas SQL que están almacenadas en la base de datos. Como olas vistas, pueden incrementar la eficiencia y simplificar la administración de la base de datos.
- Triggers. Son comandos SQL que se ejecutan automáticamente en ciertas operaciones (INSERT, UPDATE y DELETE).
- Unicode. Además, MySQL soporta casi cualquier tipo de codificación de caracteres, como Latin-1 y Latin-2.
- Full-text search. Esta característica acelera y simplifica la búsqueda de palabras que se encuentran almacenadas en campos de tipo texto.
- Replicación. Permite copiar el contenido de una base de datos a múltiples ordenadores. Esta característica se usa para mejorar la protección contra fallos, y parta acelerar las consultas.
- Transacciones. Una transacción es un conjunto de operaciones sobre una base de datos que funcionan como un bloque. Es sistema asegura que o bien se han ejecutado todas las operaciones, o ninguna. Es to es válido incluso si hay una caída del sistema, un fallo eléctrico, o cualquier otro desastre.
- Restricciones sobre claves externas (*foreign keys*). Son reglas con las que se asegura la integridad de las relaciones entre tablas.
- Lenguajes de programación. Hay un gran número de APIs y librerías para desarrollar aplicaciones MySQL. Por ejemplo, se pueden usar lenguajes como C, C++, Java, Perl, PHP, Python y Tcl.
- ODBC. Permite que MySQL sea usada con la interfície ODBC de Microsoft, lo que hace que pueda ser usada por lenguajes como Dephi, VisualBasic, ...
- Independiente de plataforma. El servidor MySQL puede funcionar sobre una variedad de sistemas operativos como Mac OS X, Linux, Microsoft Windows, y todas las variastes Unix (AIX, BSDI, FreeBSD, HP-UX, OpenBSD, SGI, Sun Solaris)

Sin embargo, a pesar de esta importante lista de características, MySQl tiene una serie de limitaciones:

- Cuando MySQL usa las tablas del tipo MyISAM, el sistema de bloqueo (*locking*) de datos solo funciona para tablas enteras. Eso significa que si queremos modificar una tabla y no queremos que nadie más pueda interferir en la operación, la única maneras es bloquear totalmente el acceso a la tabla entera. Este problema se puede resolver usando tablas del tipo InnoDB que soportan bloqueo por registro.
- MySQL no permite añadir tipos de datos definidos por el usuario.
- MySQL no soporta XML.
- MySQL no ofrece funcionalidad para aplicaciones en tiempo real.
- El sistema de triggers aún no está maduro.

1.3.1 Sistema de versiones

Las versiones de MySQL se identifican con los siguientes atributos:

- Alpha. Estas versiones están en pleno desarrollo y por lo tanto se pueden esperar nuevas funcionalidades, y cambios que las hagan incompatibles con versiones anteriores. También es probable que tengan errores severos.
- Beta. La versión está casi completa, pero no ha sido probada extensivamente. Ya no se introducirán grandes cambios.
- Gamma. La versión Beta ha alcanzado cierta estabilidad. Se resolverán los errores que aparezcan. También denominada *Release Candidate (RC)*.
- *Generally Available* (GA). Esta versión ya puede ser usada en producción y en entornos donde la fiabilidad es crítica. En estos momentos corresponde a la versión 5.0.

La historia de MySQL desde el punto de sus versiones ha sido la siguiente (actualizada a 24/04/2008):

- 3.23: Enero de 2001
- 4.0: Marzo de 2003
- 4.1: Octubre de 2004
- 5.0: Octubre 2005
    - Esta versión es actualmente la *Generally Available*, y por lo tanto la recomendada para entornos de producción.
    - En esta versión se incorporaron características tan importantes como triggers y stored procedures.
- 5.1: hoy (5.1.24)
    - Esta versión está en estado de *Release Candidate*
- 6.0: hoy (6.0.4)
    - Esta versión está en estado Alpha y por lo tanto lejos de poder usada.
    - Incluirá el nuevo motor de almacenamiento Falcon

1.3.2 Licencia

Una de las características más atractivas de MySQL es su licencia. MySQL es un proyecto de software libre. Desde Junio de 2000 se libera bajo la licencia GPL (*GNU Public License*). Sin embargo, si se quiere usar MySQL para un fin comercial sin la necesidad de proporcionar el código fuente de nuestra aplicación, entonces se puede comprar una licencia comercial.

1.4 Arquitectura LAMP

Una de las razones de la popularidad de MySQL reside en que es una pieza fundamental de la arquitectura LAMP (**L**inux + **A**pache + **M**ySQL + **P**HP). Esta arquitectura puede verse resumida en la Figura 1.

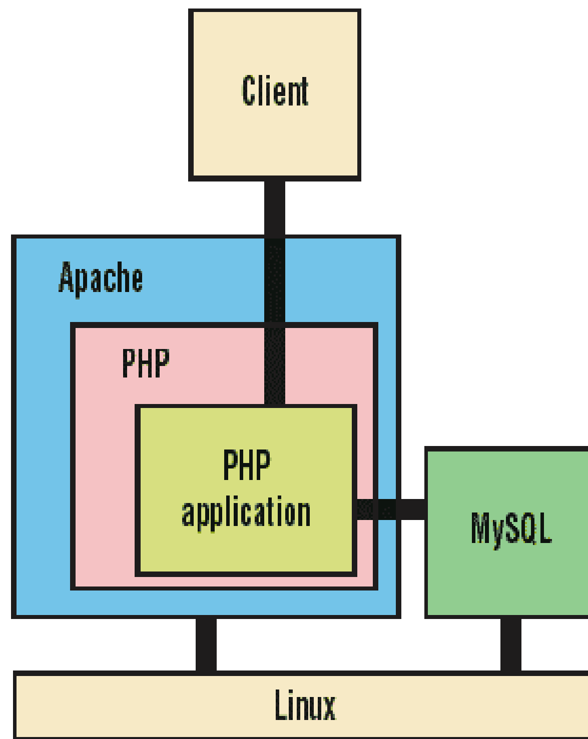


Figura 1: Arquitectura LAMP

Esta arquitectura es muy utilizada para servidores de aplicaciones web. Consiste en ordenadores funcionando con el sistema operativo GNU/Linux que ejecutan el servidor web Apache y el servidor MySQL. Sobre estos dos funcionan aplicaciones web escritas en el lenguaje PHP. Su gran difusión se debe a que todos los componentes son software libre y además a su eficiencia. La parte central de esta arquitectura se podría denominar LAM ya que Linux+Apache+MySQL es el denominador común sobre el que pueden funcionar no solo aplicaciones en PHP sino en multitud de lenguajes como Python o Java.

2. Arquitectura

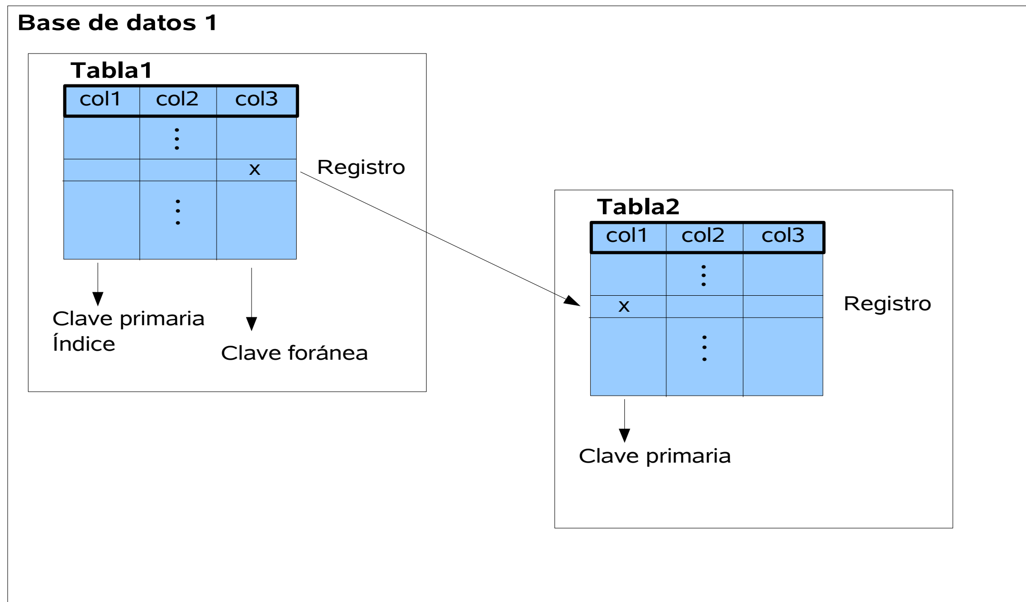


Ilustración 2: Esquema de una base de datos relacional

MySQL es un sistema de bases de datos relacional. Eso significa que la información está organizada en bases de datos, que están compuestas por tablas, que están compuestas por registros, que están compuestos por columnas. En la Figura 2 podemos ver dicha estructura.

Cada base de datos está compuesta por tablas. Las tablas a su vez se definen con columnas, cada una de un tipo de datos diferente, que conforman los registros. Además de los datos que almacenamos, las tablas pueden contener índices, y algunas de sus columnas tienen propiedades especiales como claves primarias y claves foráneas que permiten establecer relaciones entre las tablas.

Los sistemas que manejan estas estructuras se pueden describir en capas. En general, un sistema de bases de datos relacional tienen tres capas:

FIgura 3: las tres capas de un sistema de bases de datos relacional

La capa de aplicación es la parte más externa del sistema y es la interfice a través de la que los usuarios se comunican con el sistema.

La funcionalidad central del sistema está en la capa lógica. Es donde se realizan todas las operaciones del sistema. Esta capa se puede refinar de la siguiente manera:

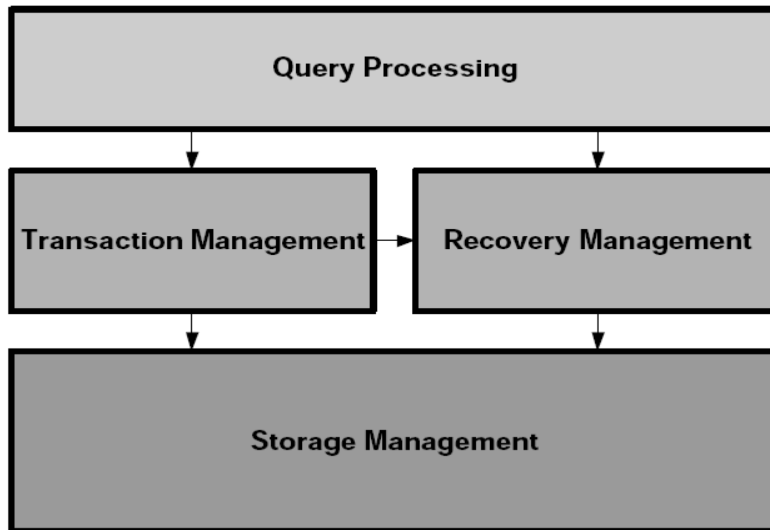


Figura 4: Descripción de la capa lógica

En esta capa se realizan varias operaciones. La primera y que sirve de interficie con la capa de aplicación es el procesamiento de las instrucciones SQL. Después hay dos módulos: el manejo de transacciones, y la recuperación después de errores. Finalmente está la parte que se encarga de traducir las instrucciones SQL en acciones sobre el almacenamiento de datos.

Finalmente, la capa física es donde están almacenados los datos.

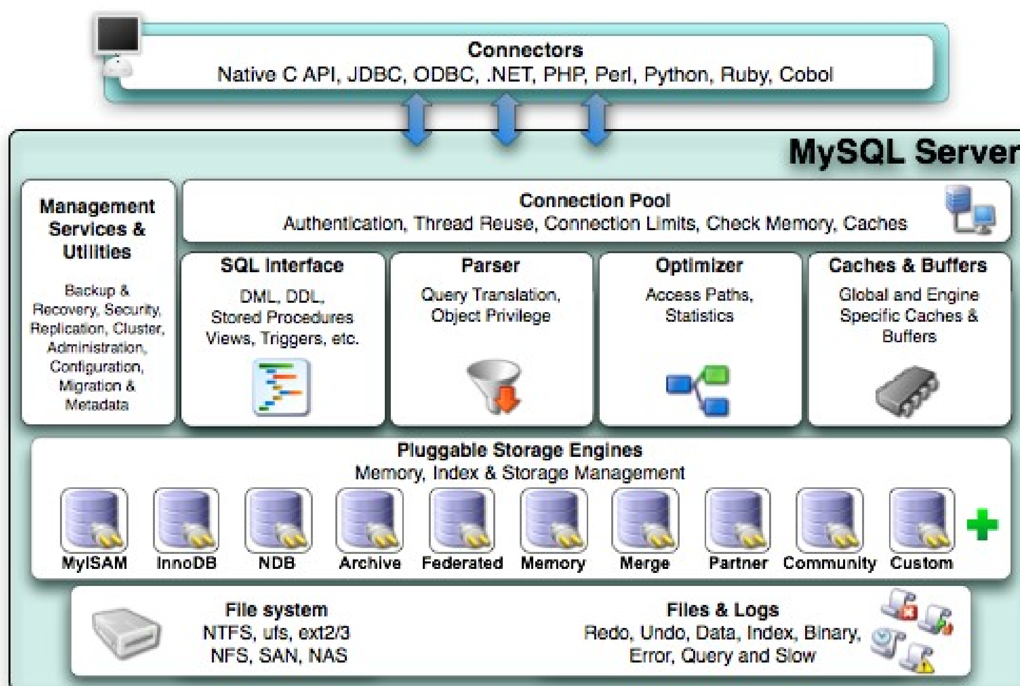


Figura 5: Arquitectura de MySQL

Esta arquitectura general sirve para MySQL, y en la Figura 5 podemos ver con más detalle los aspectos particulares del sistema.

En esta figura, los *Connectors* representan la API que MySQL expone al usuario, por lo que representaría la parte más cercana al sistema de la capa aplicación. MySQL dispone de APIs para muchos lenguajes de programación. En la parte más baja podemos ver los elementos *File system* y *Files & Logs* que representan la capa física.

Lo que queda entre medio es la capa lógica, donde reside la funcionalidad del servidor. La Figura 6 muestra como es el flujo de ejecución de una sentencia SQL dentro del servidor. Básicamente, las consultas entran a través de un protocolo cliente/servidor provenientes

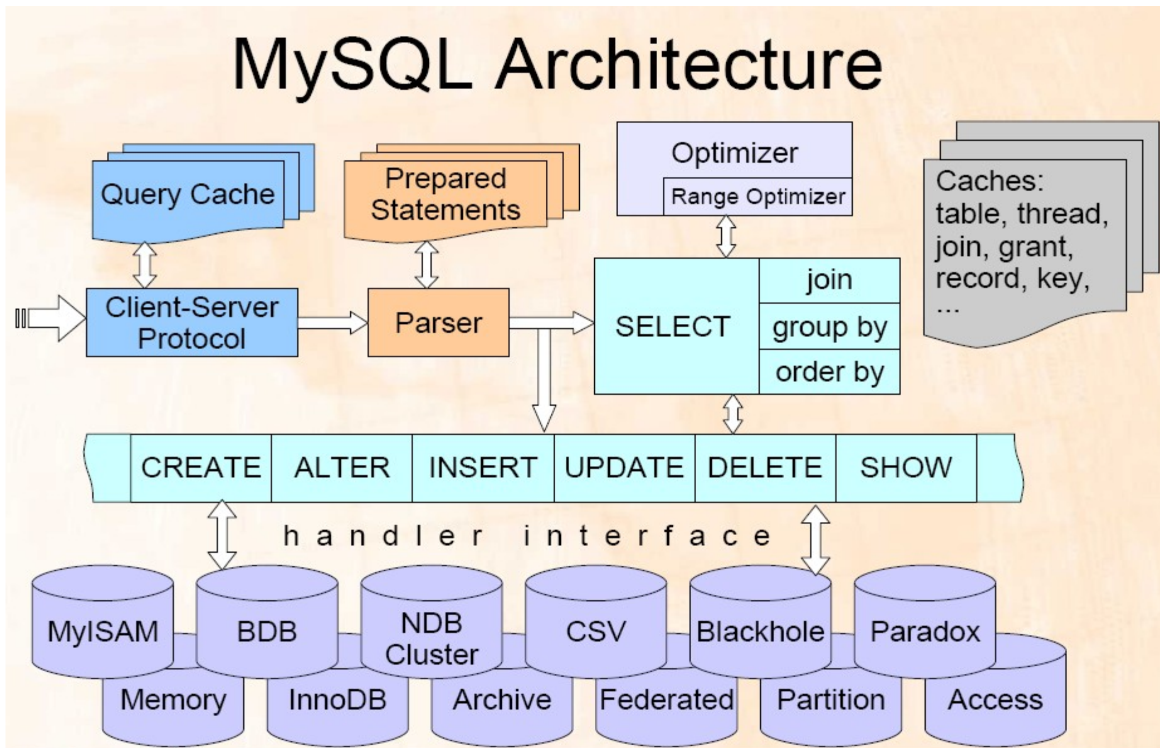


Figura 6: Flujo de ejecución del servidor MySQL

de las aplicaciones. Las consultas, pueden ser almacenas en una cache para su posterior reutilización. Después, si la consulta que ha entrado no se encuentra en la cache, se procede a su análisis sintáctico, que produce una serie de estructuras de datos que se almacenan en memoria. En esta fase, puede hacerse uso de las sentencias preparadas (*Prepared statements*) que sirven para aumentar la eficiencia de determinadas consultas. Una vez la consulta está preparada, su ejecución puede ser directa (no es un SELECT) o bien pasar por el optimizador si es de tipo SELECT. Esto es así ya que las instrucciones SELECT suelen ser las potencialmente más costosas ya que pueden necesitar acceder a tablas enteras o grandes porciones de la base de datos.

Una vez la sentencia está lista para ejecutarse, el acceso a los datos se hace a través de una interfície genérica de acceso a los datos físicos que es independiente del tipo de tabla que usemos. Y por último, la consulta se ejecuta en el subsistema correspondiente al tipo de tabla que estamos accediendo.

3. SQL y MySQL

En esta sección daremos un breve repaso al lenguaje SQL. Este lenguaje se usa para formular instrucciones al sistema de bases de datos, incluyendo consultas e instrucciones para cambiar o borrar objetos de la base de datos.

3.1 Tipos de instrucciones

Las instrucciones principales de SQL se pueden clasificar en tres grupos:

- ***Data Manipulation Language* (DML):** SELECT, INSERT, UPDATE y DELETE, y varias instrucciones más sirven para leer datos de las tablas, y para almacenar y modificarlos. Son la parte central del lenguaje.
- ***Data Definition Language*** (**DDL**): son las instrucciones que sirven para diseñar la base de datos: CREATE TABLE, ALTER TABLE, ...
- ***Data Control Language*** (**DCL**): son las instrucciones usadas para definir los mecanismos de seguridad de las base de datos: GRANT, REVOKE.

3.2 Instrucciones básicas

3.2.1 Consultas (SELECT)

El comando SELECT se utiliza para extraer información de las tablas. Por ejemplo:

```
mysql> SELECT * FROM editoriales;
+--------+-------------------+---------------------+
| editID | nombreEdit        | ts                  |
+--------+-------------------+---------------------+
|      1 | Grupo Anaya       | 2004-12-02 18:36:58 |
|      2 | Grupo Planeta     | 2004-12-02 18:36:58 |
|      3 | Crisol            | 2004-12-02 18:36:58 |
|      4 | Ediciones Destino | 2004-12-02 18:36:58 |
|      5 | Editorial Civitas | 2004-12-02 18:36:58 |
|      9 | Blume             | 2004-12-02 18:36:58 |
|     16 | Paradox Libros    | 2004-12-02 18:36:58 |
|     17 | Diaz de Santos    | 2004-12-02 18:36:58 |
|     19 | Anagrama          | 2004-12-02 18:36:58 |
|     20 | Editecnicas       | 2004-12-02 18:36:58 |
|     21 | Desnivel          | 2004-12-02 18:36:58 |
|     23 | Gredos            | 2004-12-02 18:36:58 |
|     24 | Herder            | 2004-12-02 18:36:58 |
+--------+-------------------+---------------------+
13 rows in set (0.00 sec)
```

Se puede usar para contar el número de registros:

```
mysql> SELECT COUNT(editID) FROM editoriales;
+---------------+
| COUNT(editID) |
+---------------+
|            13 |
+---------------+
1 row in set (0.00 sec)
```

O el número de registros únicos (DISTINCT):

```
mysql> SELECT COUNT(DISTINCT editID) FROM titulos;
+------------------------+
| COUNT(DISTINCT editID) |
+------------------------+
|                      7 |
+------------------------+
1 row in set (0.00 sec)
```

Se puede restringir las columnas que se seleccionan:

```
mysql> SELECT nombreEdit FROM editoriales;
+--------------------+
| nombreEdit         |
+--------------------+
| Grupo Anaya        |
| Grupo Planeta      |
| Crisol             |
| Ediciones Destino  |
| Editorial Civitas  |
| Blume              |
| Paradox Libros     |
| Diaz de Santos     |
| Anagrama           |
| Editecnicas        |
| Desnivel           |
| Gredos             |
| Herder             |
+--------------------+
13 rows in set (0.00 sec)
```

Se puede limitar el número de registros leídos:

```
mysql> SELECT nombreEdit FROM editoriales LIMIT 3;
+----------------+
| nombreEdit     |
+----------------+
| Grupo Anaya    |
| Grupo Planeta  |
| Crisol         |
+----------------+
3 rows in set (0.00 sec)
```

Se pueden ordenar los resultados:

```
mysql> SELECT * FROM editoriales ORDER BY nombreEdit;
+--------+-------------------+---------------------+
| editID | nombreEdit        | ts                  |
+--------+-------------------+---------------------+
|     19 | Anagrama          | 2004-12-02 18:36:58 |
|      9 | Blume             | 2004-12-02 18:36:58 |
|      3 | Crisol            | 2004-12-02 18:36:58 |
|     21 | Desnivel          | 2004-12-02 18:36:58 |
|     17 | Diaz de Santos    | 2004-12-02 18:36:58 |
|      4 | Ediciones Destino | 2004-12-02 18:36:58 |
|     20 | Editecnicas       | 2004-12-02 18:36:58 |
|      5 | Editorial Civitas | 2004-12-02 18:36:58 |
|     23 | Gredos            | 2004-12-02 18:36:58 |
|      1 | Grupo Anaya       | 2004-12-02 18:36:58 |
|      2 | Grupo Planeta     | 2004-12-02 18:36:58 |
|     24 | Herder            | 2004-12-02 18:36:58 |
|     16 | Paradox Libros    | 2004-12-02 18:36:58 |
+--------+-------------------+---------------------+
13 rows in set (0.00 sec)1
```

3.2.2 Seleccionar registros (WHERE)

Si queremos filtrar los resultados de un SELECT podemos poner condiciones:

```
mysql> SELECT nombreAutor FROM autores WHERE nombreAutor >= 'M';
+-----------------+
| nombreAutor     |
+-----------------+
| Martin Josep    |
| Molina Ana      |
| Molina Marc     |
| Montilla Belen  |
| Muntal Carmen   |
| Ortega Narcis   |
| Perez Miquel    |
| Puig David      |
| Reig Tomas      |
| Ruiz Sebastian  |
| Vila Robert     |
+-----------------+
11 rows in set (0.01 sec)
```

```
mysql> SELECT nombreAutor FROM autores WHERE nombreAutor LIKE '%ar%';
+-----------------+
| nombreAutor     |
+-----------------+
| Martin Josep    |
| Alvarez Tobias  |
| Gil Carles      |
| Garcia Jordi    |
| Ortega Narcis   |
| Molina Marc     |
| Colomer Gerard  |
| Garcia Simon    |
| Duarte Pablo    |
| Muntal Carmen   |
+-----------------+
10 rows in set (0.00 sec)
```

```
mysql> SELECT nombreAutor FROM autores WHERE IDautor IN (1, 7, 37);
+-----------------+
| nombreAutor     |
+-----------------+
| Perez Miquel    |
| Costa Pau       |
| Fontanella Teo  |
+-----------------+
3 rows in set (0.00 sec)
```

3.2.3 Enlazar tablas (JOIN)

Por ejemplo, si queremos obtener una lista de libros con su editorial correspondiente:

```
mysql> SELECT titulo, nombreEdit FROM titulos, editoriales WHERE
titulos.editID = editoriales.editID;
+-----------------------------------------------+--------------------+
| titulo                                        | nombreEdit         |
+-----------------------------------------------+--------------------+
| Linux                                         | Grupo Anaya        |
| Client/Server Survival Guide                  | Grupo Anaya        |
| A Guide to the SQL Standard                   | Grupo Anaya        |
```

```
| Visual Basic 6                                 | Grupo Anaya        |
| Excel 2000 programmieren                       | Grupo Anaya        |
| LaTeX                                          | Grupo Anaya        |
| Mathematica                                    | Grupo Anaya        |
| Maple                                          | Grupo Anaya        |
| VBA-Programación con Excel 7                   | Grupo Anaya        |
| Visual Basic: programación de bases de datos   | Grupo Anaya        |
| MySQL                                          | Grupo Anaya        |
| LaTeX en la web                                | Grupo Anaya        |
| Linux                                          | Grupo Anaya        |
| PostgreSQL                                     | Grupo Anaya        |
| Java                                           | Grupo Anaya        |
| The Definitive Guide to Excel VBA              | Grupo Planeta      |
| MySQL                                          | Grupo Planeta      |
| A Programmer's Introduction to PHP 4.0         | Grupo Planeta      |
| Web Application Development with PHP 4.0       | Crisol             |
| MySQL                                          | Crisol             |
| MySQL & mSQL                                   | Ediciones Destino  |
| Practical UNIX & Internet Security             | Ediciones Destino  |
| MySQL Cookbook                                 | Ediciones Destino  |
| PHP - Introduccion                             | Editorial Civitas  |
| Carlomagno                                     | Blume              |
| Comedia Infantil                               | Diaz de Santos     |
| Hunderna i Riga                                | Diaz de Santos     |
+------------------------------------------------+--------------------+
27 rows in set (0.00 sec)
```

Pero hay otra manera que consiste en utilizar JOIN:

```
SELECT titulo, nombreEdit
      FROM titulos LEFT JOIN editoriales
            ON titulos.editID = editoriales.editID;
```

Y si el nombre de la columna es el mismo en las dos tablas:

```
SELECT titulo, nombreEdit
      FROM titulos LEFT JOIN editoriales
      USING (editID);
```

MySQL SELECT

La sintaxis completa de SELECT en MYSQL es la siguiente:

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr, ...
    [FROM table_references
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [PROCEDURE procedure_name(argument_list)]
    [INTO OUTFILE 'file_name' export_options
      | INTO DUMPFILE 'file_name'
      | INTO var_name [, var_name]]
    [FOR UPDATE | LOCK IN SHARE MODE]]
```

En rojo están las partes de `SELECT` más comúnmente usadas. Ahora daremos una breve descripción de las opciones particulares de MySQL:

- ALL, DISTINCT, DISTINCTROW. Sirven para determinar qi queremos quedarnos con todos los registros (ALL, valor por defecto) o queremos eliminar los duplicados (DISTINCT, DISTINCTROW son sinónimos).
- HIGH_PRIORITY. Da prioridad al SELECT sobre instrucciones INSERT que se estén ejecutando al mismo tiempo. Solo afecta a tablas que solo disponen de bloqueo de tabla (MyISAM, MEMORY, MERGE).
- STRAIGHT_JOIN. Fuerza al optimizador a ejecutar el JOIN en el orden exacto en en que se especifica en la sentencia SELECT. Se usa cuando sabemos que el optimizador produce una orden que de hecho no es óptimo.
- SQL_BIG_RESULT. Se puede usar con GROUP BY y DISTINCT para decirle al optimizador que el resultado va a ser muy grande. Eso le dice al optimizador que debe usar tablas temporales en disco.
- SQL_BUFFER_RESULT. Fuerza a que los resultados sean almacenados en una tabla temporal. Esto ayuda a MySQL a liberar los bloqueos de tabla rápidamente y ayuda en casos en que tarda mucho tiempo en enviar el resultado al cliente.
- SQL_SMALL_RESULT puede usarse con GROUP BY o DISTINCT para decir al optimizador que el conjunto de resultados es pequeño. En este caso, MySQL usa tablas temporales rápidas para almacenar la tabla resultante en lugar de usar ordenación. En MySQL 5.0, esto no hará falta normalmente.
- SQL_CALC_FOUND_ROWS le dice a MySQL que calcule cuántos registros habrán en el conjunto de resultados, sin tener en cuenta ninguna cláusula LIMIT. El número de registros pueden encontrarse con SELECT FOUND_ROWS().
- SQL_CACHE le dice a MySQL que almacene el resultado de la consulta en la caché de consultas si está usando un valor de query_cache_type de 2 o DEMAND. Para una consulta que use UNION o subconsultas, esta opción afecta a cualquier SELECT en la consulta.
- SQL_NO_CACHE le dice a MySQL que no almacene los resultados de consulta en la caché de consultas. Para una consulta que use UNION o subconsultas esta opción afecta a cualquier SELECT en la consulta.

3.2.4 Modificar datos (INSERT, UPDATE, y DELETE)

Con la instrucción INSERT se pueden añadir registros a una tabla. Por ejemplo:

```
INSERT INTO titulos (titulo, año)
VALUES ('MySQL', 2007)
```

Se puede escoger que columnas rellenar, aunque siempre teniendo en cuenta que solo se pueden ignorar las que admiten un valor NULL, o las que son AUTO_INCREMENT. Si no se especifica que columnas se están rellenando, hay que dar valores para todas:

```
INSERT INTO titulos
VALUES (NULL, 'MySQL', '', 1, NULL, NULL, NULL, 2007, NULL, NULL, NULL)
```

También se pueden insertar varios registros a la vez:

```
INSERT INTO titulos (titulo, año)
VALUES ('tituloA', '2007'), ('tituloB', 2007), ('tituloC', 2007)
```

Con la instrucción UPDATE se pueden modificar registros ya existentes. En genral se usa de la forma:

```
UPDATE nombre_de_tabla
```

```
SET columna1=valor1, columna2=valor2, ...
WHERE id_columna=n
```

Y con el comando DELETE se pueden borrar registros de una tabla. Por ejemplo:

```
DELETE FROM nombre_de_tabla
WHERE id_columna=n
```

Para borrar tablas que están enlazadas con claves externas, se pueden borrar registros de diferentes tablas a la vez:

```
DELETE t1, t2 FROM t1, t2, t3 WHERE condicion1 AND condicion2 ...
```

Por ejemplo, usando el ejemplo de la biblioteca, si queremos borrar un titulo tenemos que tener en cuenta sus relaciones:

```
DELETE titulos FROM titulos, rel_titulo_autor, autores
WHERE titulos.tituloID = titulo_autor.tituloID
      AND autores.autorID = rel_titulo_autor.autorID
      AND autores.nombreAutor = 'Costa Pau'
```

INSERT

La sintaxis MySQL de esta instrucción es:

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name [(col_name,...)]
    VALUES ({expr | DEFAULT},...),(...),...
    [ ON DUPLICATE KEY UPDATE
      col_name=expr
        [, col_name=expr] ... ]
```

Las opciones en rojo son las más comúnmente usadas, y a continuación explicaremos el significado del resto de opciones:

- Si se usa la palabra `DELAYED`, el servidor pone el registro o registros a ser insertados en un búffer, y el cliente realizando el comando `INSERT DELAYED` puede continuar. Si la tabla está en uso, el servidor trata los registros. Cuando la tabla se libera, el servidor comienza a insertar registros, chequeando periódicamente para ver si hay alguna petición de lectura para la tabla. Si la hay, la cola de registros retardados se suspende hasta que la tabla se libera de nuevo.
- Si se usa la palabra `LOW_PRIORITY` , la ejecución de `INSERT` se retrasa hasta que no hay otros clientes leyendo de la tabla. Esto incluye a otros clientes que comiencen a leer mientras que los clientes existentes están leyendo, y mientras el comando `INSERT LOW_PRIORITY` está en espera. Es posible, por lo tanto, para un cliente que realice un comando `INSERT LOW_PRIORITY` esperar durante mucho tiempo (o incluso para siempre) en un entorno de muchas lecturas. (Esto es un contraste de `INSERT DELAYED`, que deja al cliente continuar. Tener en cuenta que `LOW_PRIORITY` no debe usarse normalmente con tablas `MyISAM` y que hacerlo deshabilita inserciones concurrentes.
- Si especifica `HIGH_PRIORITY`, deshabilita el efecto de la opción `--low-priority-updates` si el servidor se arrancó con esa opción. Hace que las inserciones concurrentes no se usen.
- Los valores afectados por un `INSERT` pueden usarse usando la función `mysql_affected_rows()` de la API de C.

- Si se usa la palabra IGNORE en un comando INSERT , los errores que ocurren mientras se ejecuta el comando se tratan como advertencias. Por ejemplo, sin IGNORE, un registro que duplique un índice UNIQUE existente o valor PRIMARY KEY en la tabla hace que un error de clave duplicada en el comando se aborte. Con IGNORE, el registro todavía no se inserta, pero no se muestra error. Las conversiones de datos dispararían errores y abortarían el comando si no se especificara IGNORE . Con IGNORE, los valores inválidos se ajustan al valor más cercano y se insertan; las advertencias se producen pero el comando no se aborta. Se puede determinar con la función mysql_info() de la API de C cuántos registros se insertan realmente en la tabla.
- Si se especifica ON DUPLICATE KEY UPDATE, y se inserta un registro que duplicaría un valor en un índice UNIQUE o PRIMARY KEY, se realiza un UPDATE del antiguo registro. Por ejemplo, si la columna a se declara como UNIQUE y contiene el valor 1, los siguientes dos comandos tienen efectos idénticos:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3)
-> ON DUPLICATE KEY UPDATE c=c+1;

mysql> UPDATE table SET c=c+1 WHERE a=1;
```

  El valor de registros afectados es 1 si el registros se inserta como un nuevo registro y 2 si un valor existente se actualiza.

  Nota: Si la columna b es única, el INSERT sería equivalente a este comando UPDATE :

```
mysql> UPDATE table SET c=c+1 WHERE a=1 OR b=2 LIMIT 1;
```

  Si a=1 OR b=2 se cumple para varios registros, sólo *un* registro se actualiza. En general, debería intentar evitar usar una cláusula ON DUPLICATE KEY en tablas con claves únicas múltiples.

  MySQL 5.0 permite el uso de la función VALUES(col_name) en la cláusula UPDATE que se refiere a los valores de columna de la porción INSERT del comando INSERT ... UPDATE . En otras palabras, VALUES(col_name) en la cláusula UPDATE se refiere al valor de col_name que se insertarían, no ocurre conflicto de clave duplicada. Esta función es especialmente útil en inserciones de múltiples registros. La función VALUES() tiene sentido sólo en comandos INSERT ... UPDATE y retorna NULL de otro modo.

  Ejemplo:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3),(4,5,6)
-> ON DUPLICATE KEY UPDATE c=VALUES(a)+VALUES(b);
```

  Este comando es idéntico a los siguientes dos comandos:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3)
-> ON DUPLICATE KEY UPDATE c=3;
mysql> INSERT INTO table (a,b,c) VALUES (4,5,6)
-> ON DUPLICATE KEY UPDATE c=9;
```

  Cuando usa ON DUPLICATE KEY UPDATE, la opción DELAYED se ignora.
- Puede encontrar el valor usado para una columna AUTO_INCREMENT usando la función SQL LAST_INSERT_ID() . Desde la API C, use la función

`mysql_insert_id()` . Sin embargo, debe tener en cuenta que las dos funciones no siempre se comportan idénticamente.

DELETE, UPDATE

La sintaxis de DELETE y UPDATE es:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
      [WHERE where_definition]
      [ORDER BY ...]
      [LIMIT row_count]

UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
    SET col_name1=expr1 [, col_name2=expr2 ...]
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

Estas instrucciones no tienen opciones específicas a parte de LOW_PRIORITY que ya la hemos visto antes. Si se usa la opción `QUICK` , el motor de almacenamiento no mezcla las hojas del índice durante el borrado, que puede acelerar algunos tipos de operaciones de borrado.

3.2.5 Crear bases de datos, tablas e índices

Normalmente usaremos una herramienta interactiva para crear bases de datos, tablas, índices. Sin embargo todas estas herramientas lo que hacen es usar instrucciones SQL. Por ejemplo, para crear una base de datos usaremos CREATE DATABASE:

```
CREATE DATABASE biblioteca
```

Opcionalmente se puede especificar la codificación de los caracteres:

```
CREATE DATABASE biblioteca
DEFAULT CHARACTER SET latin1 COLLATE latin1_general_ci
```

Para crear una tabla usaremos CREATE TABLE. La sintaxis simplificada es:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] nombre_tabla (
      nombre_columna1, tipo_columna opciones_de_columna referencia,
      nombre_columna2, tipo_columna opciones_de_columna referencia, ...
[ , index1, index2 ...] )
[ ENGINE = MyISAM|InnoDB|HEAP ]
[ DEFAULT CHARSET = csname [ COLLATE = collname ]]
```

Por ejemplo, para crear la tabla `titulos` de la base de datos `biblioteca` usaremos esto:

```
CREATE TABLE IF NOT EXISTS `titulos` (
  `tituloID` int(11) NOT NULL AUTO_INCREMENT,
  `titulo` varchar(100) COLLATE latin1_spanish_ci NOT NULL DEFAULT '',
  `subtitulo` varchar(100) COLLATE latin1_spanish_ci DEFAULT NULL,
  `edition` tinyint(4) DEFAULT NULL,
  `editID` int(11) DEFAULT NULL,
  `catID` int(11) DEFAULT NULL,
  `idiomaID` int(11) DEFAULT NULL,
  `año` int(11) DEFAULT NULL,
  `isbn` varchar(20) COLLATE latin1_spanish_ci DEFAULT NULL,
  `comentario` varchar(255) COLLATE latin1_spanish_ci DEFAULT NULL,
  `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
```

```
  `autores` varchar(255) COLLATE latin1_spanish_ci DEFAULT NULL,
  PRIMARY KEY (`tituloID`),
  KEY `publIdIndex` (`editID`),
  KEY `idiomaID` (`idiomaID`),
  KEY `catID` (`catID`),
  KEY `titulo` (`titulo`),
  INDEX idxTitulo ('titulo'),
  CONSTRAINT `titulos_ibfk_3` FOREIGN KEY (`idiomaID`) REFERENCES `idiomas`
      (`idiomaID`),
  CONSTRAINT `titulos_ibfk_1` FOREIGN KEY (`editID`) REFERENCES `editoriales`
      (`editID`),
  CONSTRAINT `titulos_ibfk_2` FOREIGN KEY (`catID`) REFERENCES `categorias`
      (`catID`);
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 COLLATE=latin1_spanish_ci;
```

Al crear esta tabla ya hemos creado los índices y las claves externas. Sin embargo esto se puede hacer después de crear la tabla con instrucciones separadas:

```
CREATE INDEX idxTitulo ON titulos (titulo)
o
ALTER TABLE titulos ADD INDEX idxTitulo (titulo)
```

También podemos crear nuevas tablas a partir de un SELECT:

```
CREATE TABLE libros_fantasia
SELECT * FROM titulos WHERE catID=65
```

Desde este momento la tabla `libros_fantasia` se puede usar de forma independiente. Y cuando ya no la necesitemos, la podemos borrar así:

```
DROP TABLE libros_fantasia
```

3.2.6 Cambiar diseño de tablas

Con ALTER TABLE podemos cambiar los detalles de una tabla como añadir o eliminar columnas, cambiar las propiedades de las columnas (como el tipo de datos), y definir o borrar índices. Por ejemplo, si queremos aumentar el número máximo de caracteres asignados a la columna `titulo` de la tabla `titulos` haremos lo siguiente:

```
ALTER TABLE titulos CHANGE titulo titulo VARCHAR(150) NOT NULL
```

Para añadir un columna, la sintaxis general es:

```
ALTER TABLE nombre_tabla ADD nombre_columna_nueva tipo_columna
opciones_columna
      [FIRST | AFTER columna_existente]
```

Para borrar una columna:

```
ALTER TABLE nombre_tabla DROP nombre_columna
```

3.2.7 Borrar bases de datos y tablas

Para borrar bases de datos y tablas:

```
DROP TABLE nombre_tabla
DROP DATABASE nombre_bd
```

3.3 Cambios de diseño automáticos

Cuando creamos una tabla con CREATE TABLE o hacemos un cambio con ALTER TABLE, MySQL puede, bajo determinadas circunstancias, hacer cambios en el diseño de la tabla. La razón es que la tabla, con estos cambios, será más eficiente, o que los cambios que queremos hacer no son posibles con MySQL. Para asegurarnos que el diseño de la tabla es exactamente lo que pensamos que es, podemos usar SHOW CREATE TABLE. Por ejemplo, si creamos la siguiente tabla:

```
CREATE TABLE test1 (col1 VARCHAR(20), col2 CHAR(20))
```

Sin embargo, si usamos SHOW CREATE TABLE obtendremos lo siguiente:

```
SHOW CREATE TABLE test1
      CREATE TABLE 'test1' (
             'col1' varchar(20) default NULL,
             'col2' varchar(20) default NULL
      ) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

MySQL ha transformado la columna 2 de `CHAR(20)` a `VARCHAR(20)`, y además ha añadido el atributo `DEFAULT NULL` a las dos columnas.

En general, estos son los cambios más importante que MySQL efectúa en el diseño de las tablas:

- Las columnas VARCHAR(n) con n<4 se transforman en CHAR(n)
- Las columnas CHAR(n) con n>3 se transforman en VARCHAR(n) si existe en la tablas alguna columna adicional del tipo VARCHAR, TEXT o BLOB. Si no, se deja como está.
- Las columnas de tipo TIMESTAMP no pueden almacenar valores NULL. Cuando se define una columna de este tipo con el atributo NULL, el efecto es que se almacena el valor 0000-00-00 00:00:00.
- Las columnas que son PRIMARY KEY siempre tienen asignado el atributo NOT NULL, aunque no lo hayamos especificado.
- Si no se define el valor por defecto de una columna, MySQL definirá uno apropiado (NULL, 0, una cadena de caracteres vacía).

3.4 Inspeccionar meta datos (SHOW)

La manera de recuperar información sobre los datos almacenados es el comando SHOW. Por ejemplo, SHOW DATABASES muestra una lista de las bases de datos del sistema, SHOW TABLES muestra una lista con las tablas existentes en una bases de datos, SHOW COLUMNS muestra la información sobre las columnas de una tabla.

Por ejemplo, podemos inspeccionar la estructura de una tabla:

```
mysql> SHOW COLUMNS FROM titulos;
+------------+--------------+------+-----+-------------------+-----------------------------+
| Field      | Type         | Null | Key | Default           | Extra                       |
+------------+--------------+------+-----+-------------------+-----------------------------+
| tituloID   | int(11)      | NO   | PRI | NULL              | auto_increment              |
| titulo     | varchar(100) | NO   | MUL |                   |                             |
| subtitulo  | varchar(100) | YES  |     | NULL              |                             |
| edition    | tinyint(4)   | YES  |     | NULL              |                             |
| editID     | int(11)      | YES  | MUL | NULL              |                             |
| catID      | int(11)      | YES  | MUL | NULL              |                             |
| idiomaID   | int(11)      | YES  | MUL | NULL              |                             |
| año        | int(11)      | YES  |     | NULL              |                             |
| isbn       | varchar(20)  | YES  |     | NULL              |                             |
| comentario | varchar(255) | YES  |     | NULL              |                             |
| ts         | timestamp    | NO   |     | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
| autores    | varchar(255) | YES  |     | NULL              |                             |
+------------+--------------+------+-----+-------------------+-----------------------------+
12 rows in set (0.00 sec)          'col1' varchar(20) default NULL,
```

Las formas más comunes de usar SHOW son:

Instrucción	Función
`SHOW DATABASES`	Muestra un lista de todas las bases de datos
`SHOW TABLES FROM nombre_db`	Muestra una lista de todas las tablas en la base de datos nombre_db
`SHOW [FULL] COLUMNS FROM nombre_tabla`	Muestra información detallada de todas las columnas de la tabla nombre_tabla
`SHOW INDEX FROM nombre_tabla`	Muestra una lista de los índices de la tabla nombre_tabla

DESCRIBE

Este comando es muy útil cuando no sabemos cual es la estructura de una tabla. Por ejemplo, si queremos saber cual es la estructura de la tabla `autores` de la base de datos `biblioteca` haremos esto:

```
mysql> DESCRIBE autores;
+-------------+-------------+------+-----+-------------------+-----------------------------+
| Field       | Type        | Null | Key | Default           | Extra                       |
+-------------+-------------+------+-----+-------------------+-----------------------------+
| autorID     | int(11)     | NO   | PRI | NULL              | auto_increment              |
| nombreAutor | varchar(60) | NO   | MUL |                   |                             |
| ts          | timestamp   | NO   |     | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
+-------------+-------------+------+-----+-------------------+-----------------------------+
```

Podemos ver como este comando nos da información sobre cada columna: nombre, tipo, NULL/NOT NULL, tipo de clave(PRI: primaria, MUL: no única), valor por defecto, e información adicional como si es AUTO_INCREMENT, ...

3.5 Tablas INFORMATION_SCHEMA

Todas las bases de datos relacionales almacenan datos acerca de ellas mismas en una colección de tablas de sistema no estándar. Estas tablas contienen metadatos, esto es, datos acerca de objetos en la base de datos. Es desaconsejable escribir consultas directamente sobre estas tablas ya que su estructura puede cambiar. Para evitar este problema desde SQL:92 se definió una base de datos estándar que contiene todos estos metadatos: INFORMATION_SCHEMA. Más precisamente, INFORMATION_SCHEMA es un conjunto de VIEWS con acceso de lectura únicamente.

MySQL trata INFOMATION_SCHEMA como si fuera una base de datos, por lo cual se le pueden dirigir consultas como si fuera una bases de datos normal. Esto es extremadamente útil ya que de esta manera este tipo de consultas son más fácilmente trasladables de MySQL a otros sistema de bases de datos que siga el estándar SQL

Por ejemplo:

```
mysql> use information_schema;
Database changed
mysql> show tables;
+---------------------------------------+
| Tables_in_information_schema          |
+---------------------------------------+
| CHARACTER_SETS                        |
| COLLATIONS                            |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS                               |
| COLUMN_PRIVILEGES                     |
| ENGINES                               |
| EVENTS                                |
| FILES                                 |
| GLOBAL_STATUS                         |
| GLOBAL_VARIABLES                      |
| KEY_COLUMN_USAGE                      |
| PARTITIONS                            |
| PLUGINS                               |
| PROCESSLIST                           |
| REFERENTIAL_CONSTRAINTS               |
| ROUTINES                              |
| SCHEMATA                              |
| SCHEMA_PRIVILEGES                     |
| SESSION_STATUS                        |
| SESSION_VARIABLES                     |
| STATISTICS                            |
| TABLES                                |
| TABLE_CONSTRAINTS                     |
| TABLE_PRIVILEGES                      |
| TRIGGERS                              |
| USER_PRIVILEGES                       |
| VIEWS                                 |
+---------------------------------------+
27 rows in set (0.00 sec)
```

Esta es la lista de tablas disponible en MySQL 5.1. Los detalles de los contenidos de estas tablas pueden encontrarse en:

http://dev.mysql.com/doc/refman/5.1/en/information-schema.html

Ilustración 7: Diagrama de relaciones entre las tablas INFORMATION_SCHEMA

A continuación daremos una breve descripción sobre las más importantes:

- SCHEMATA. Proporciona información sobre las bases de datos que hay en el sistema. Básicamente, los nombres y el juego de caracteres usado.
- TABLES. Describe las propiedades de todas las tablas.
- COLUMNS. Describe las propiedades de todas las columnas de todas las tablas.
- STATISTICS. Contiene la información sobre los índices existentes en el sistema.
- VIEWS. Describe las propiedades de todas las vistas del sistema. Eso incluye su definición, es decir, la sentencia SQL que se usó para crearla.
- TABLE_CONSTRAINTS. Contiene todas las restricciones de las tablas.
- KEY_COLUMN_USAGE. Contiene también todos los índices, pero con informació añadida sobre sus restricciones.
- REFERENTIAL_CONSTRAINTS. Contiene la información sobre todas las claves foráneas.
- USER_PRIVILEGES. Lista de todos los usuarios MySQL. La información de esta tabla viene de la tabla `mysql.user`. Contiene información sobre los privilegios de cada usuario dentro del sistema.
- SCHEMA_PRIVILEGES. Contiene información sobre los privilegios específicos sobre cada base de datos. La información de esta tabla viene de `mysql.db`.
- TABLE_PRIVILEGES. Contiene información sobre los privilegios específicos sobre cada tabla. La información de esta tabla viene de `mysql.tables_priv`.
- COLUMN_PRIVILEGES. Contiene información sobre los privilegios específicos sobre cada columna. La información de esta tabla viene de `mysql.columns_priv`.
- CHARACTER_SETS. Describe todos los juegos de caracteres disponibles.
- COLLATIONS. Especifica que esquemas de ordenación están disponibles para cada juego de caracteres.
- ROUTINES. Contiene información acerca de todos las funciones almacenadas (*stored procedures*).
- TRIGGERS. Contiene información sobre todos los triggers del sistema.

4. Características específicas de MySQL

En este capítulo veremos dos aspectos fundamentales de MySQL:

- Tipos de tablas o motores de almacenamiento. MySQL dispone de una API para que puedan desarrollarse diferentes métodos de almacenamiento para las tablas. Veremos cuales son los más importante y entraremos en profundidad en los tres más usados: MyISAM, InnoDB y MEMORY.
- Tipos de datos. MySQL dispone de una serie de tipos de datos preestablecidos para las columnas que no pueden ser extendidos por el usuario. Veremos las características de dichos tipos y de sus peculiaridades concretas en MySQL.

4.1 Tipos de tablas

Una peculiaridad de MySQL es que cuando se crea una nueva tabla se puede especificar su tipo. MySQL soporta una serie de tipos de tablas que se distinguen por tener propiedades diferenciadas. Las tres más importantes son MyISAM, InnoDB y MEMORY. Si al crear una tabla no se especifica el tipo, el servidor decide por nosotros basándose en su configuración.

Sin embargo, el tipo de tablas disponible en MySQL es muy amplio. Aquí mostramos un pequeño resumen:

- `MyISAM` trata tablas no transaccionales. Proporciona almacenamiento y recuperación de datos rápida, así como posibilidad de búsquedas fulltext. `MyISAM` se soporta en todas las configuraciones MySQL, y es el motor de almacenamiento por defecto a no ser que tenga una configuración distinta a la que viene por defecto con MySQL.
- El motor de almacenamiento `MEMORY` proporciona tablas en memoria. El motor de almacenamiento `MERGE` permite una colección de tablas `MyISAM` idénticas ser tratadas como una simple tabla. Como `MyISAM`, los motores de almacenamiento `MEMORY` y `MERGE` tratan tablas no transaccionales y ambos se incluyen en MySQL por defecto.

  Nota: El motor de almacenamiento `MEMORY` anteriormente se conocía como `HEAP`.
- Los motores de almacenamiento `InnoDB` y `BDB` proporcionan tablas transaccionales. `BDB` se incluye en la distribución binaria MySQL-Max en aquellos sistemas operativos que la soportan. `InnoDB` también se incluye por defecto en todas las distribuciones binarias de MySQL 5.0 . En distribuciones fuente, puede activar o desactivar estos motores de almacenamiento configurando MySQL a su gusto.
- El motor de almacenamiento `EXAMPLE` es un motor de almacenamiento "tonto" que no hace nada. Puede crear tablas con este motor, pero no puede almacenar datos ni recuperarlos. El objetivo es que sirva como ejemplo en el código MySQL para ilustrar cómo escribir un motor de almacenamiento. Como tal, su interés primario es para desarrolladores.
- `NDB Cluster` es el motor de almacenamiento usado por MySQL Cluster para implementar tablas que se particionan en varias máquinas. Está disponible en

distribuciones binarias MySQL-Max 5.0. Este motor de almacenamiento está disponible para Linux, Solaris, y Mac OS X . Se añadirá soporte para este motor de almacenamiento en otras plataformas, incluyendo Windows en próximas versiones.

- El motor de almacenamiento `ARCHIVE` se usa para guardar grandes cantidades de datos sin índices con una huella muy pequeña.
- El motor de almacenamiento `CSV` guarda datos en ficheros de texto usando formato de valores separados por comas.
- El motor de almacenamiento `FEDERATED` se añadió en MySQL 5.0.3. Este motor guarda datos en una base de datos remota. En esta versión sólo funciona con MySQL a través de la API MySQL C Client. En futuras versiones, será capaz de conectar con otras fuentes de datos usando otros drivers o métodos de conexión clientes.

4.1.1 MyISAM

Este tipo de tablas está maduro, es estable, y simple de utilizar. Si no tenemos una razón específica para elegir otro tipo de tabla, deberíamos escoger esta. Hay dos variantes de esta tabla, que MySQL escoge automáticamente según considere apropiado:

- **MyISAM static**: este tipo de usa cuando todas las tablas de la columna tienen un tamaño fijo y predeterminado. El acceso a estas tablas es muy eficiente, incluso si la tabla es modificada (INSERT, UPDATE, DELETE) con mucha frecuencia. Además, la seguridad de los datos es muy alta ya que si se produce una corrupción de los ficheros es muy fácil recuperar registros.
- **MyISAM dynamic**: si en la declaración de una tabla hay un solo valor de tipo VARCHAR, TEXT o BLOB, entonces MySQL escoge este tipo de tabla. La ventaja sobre el tipo static es que el espacio que se necesita para guardar estas tablas es menor ya que se guarda estrictamente lo necesario.

  Sin embargo, como los registros no tienen el mismo tamaño, si se modifican, su posición en la base de datos cambia y aparece un “agujero” en el fichero. También puede suceder que un registro no esté almacenado contiguamente. Todo esto hace que se incremente el tiempo de acceso conforme las tablas se van fragmentando con las modificaciones. Para evitar esta defragmentación se puede usar el comando OPTIMIZE TABLE, o un programa externo como `myisamchk`.

A parte de estas dos variantes, las tablas MyISAM pueden estar comprimidas con el programa externo `myisamchk.` Esto hace que el espacio de almacenamiento se reduzca, en promedio, a menos de la mitad. Sin embargo, cada vez que la tabla es accedida hay que efectuar un proceso de descompresión. La gran desventaja de estas tablas es que no pueden ser modificadas, es decir, son solo de lectura.

Las características de las tablas MyISAM son:

- Todos los datos se almacenan con el byte menor primero (*little-endian*). Esto hace que sean independientes de la máquina y el sistema operativo. El único requerimiento para portabilidad binaria es que la máquina use enteros con signo en complemento a dos (como todas las máquinas en los últimos 20 años) y formato en

coma flotante IEEE (también dominante en todas las máquinas). El único tipo de máquinas que pueden no soportar compatibilidad binaria son sistemas empotrados, que a veces tienen procesadores peculiares.

- No hay penalización de velocidad al almacenar el byte menor primero; los bytes en un registro de tabla normalmente no están alineados y no es un problema leer un byte no alineado en orden normal o inverso. Además, el código en el servidor que escoge los valores de las columnas no es crítico respecto a otro código en cuanto a velocidad.
- Ficheros grandes (hasta longitud de 63 bits) se soportan en sistemas de ficheros y sistemas operativos que soportan ficheros grandes.
- Registros de tamaño dinámico se fragmentan mucho menos cuando se mezclan borrados con actualizaciones e inserciones. Esto se hace combinando automáticamente bloques borrados adyacentes y extendiendo bloques si el siguiente bloque se borra.
- El máximo número de índices por tabla `MyISAM` en MySQL 5.0 es 64. Esto puede cambiarse recompilando. El máximo número de columnas por índice es 16.
- La longitud máxima de clave es 1000 bytes. Esto puede cambiarse recompilando. En caso de clave mayor a 250 bytes, se usa un tamaño de bloque mayor, de 1024 bytes.
- Las columnas `BLOB` y `TEXT` pueden indexarse.
- Valores `NULL` se permiten en columnas indexadas. Esto ocupa 0-1 bytes por clave.
- Todos los valores de clave numérico se almacenan con el byte mayor primero para mejor compresión de índice.
- Cuando se insertan registros en orden (como al usar columnas `AUTO_INCREMENT`), el árbol índice se divide de forma que el nodo mayor sólo contenga una clave. Esto mejora la utilización de espacio en el árbol índice.
- El tratamiento interno de una columna `AUTO_INCREMENT` por tabla. `MyISAM` actualiza automáticamente esta columna para operaciones `INSERT` y `UPDATE` . Esto hace las columnas `AUTO_INCREMENT` más rápidas (al menos 10%). Los valores iniciales de la secuencia no se reúsan tras ser borrados. (Cuando una columna `AUTO_INCREMENT` se define como la última columna de un índice de varias columnas, se reúsan los valores borrados iniciales de la secuencia.) El valor `AUTO_INCREMENT` puede cambiarse con `ALTER TABLE` o **myisamchk**.
- Si una tabla no tiene bloques libres en medio del fichero de datos, puede `INSERT` nuevos registros a la vez que otros flujos leen de la tabla. (Esto se conoce como inserciones concurrentes.) Un bloque libre puede ser resultado de borrar o actualizar registros de longitud dinámica con más datos que su contenido. Cuando todos los bloques libres se usan (se rellenan), las inserciones futuras vuelven a ser concurrentes.
- Puede tener el fichero de datos e índice en directorios distintos para obtener más velocidad con las opciones `DATA DIRECTORY` y `INDEX DIRECTORY` para `CREATE TABLE`.
- Cada columna de caracteres puede tener distintos juegos de caracteres.
- Hay un flag en el fichero índice `MyISAM` que indica si la tabla se ha cerrado correctamente. Si **mysqld** se arranca con la opción `--myisam-recover` , Las

tablas MyISAM se comprueban automáticamente al abrirse, y se reparan si la tabla no se cierra correctamente.

- **myisamchk** marca las tablas como chequeadas si se ejecuta con la opción --update-state . **myisamchk --fast** cheque sólo las tablas que no tienen esta marca.
- **myisamchk --analyze** almacena estadísticas para partes de las claves, así como para las claves enteras.
- **myisampack** puede comprimir columnas BLOB y VARCHAR .

Además de todo esto, MyISAM soporta las siguientes características:

- Soporte de un tipo VARCHAR auténtico; una columna VARCHAR comienza con la longitud almacenada en dos bytes.
- Tablas con VARCHAR pueden tener longitud de registro fija o dinámica.
- VARCHAR y CHAR pueden ser de hasta 64KB.
- Un índice hash puede usarse para UNIQUE. Esto le permite tener UNIQUE o cualquier combinación de columnas en una tabla . (Sin embargo, no puede buscar en un índice UNIQUE .)

4.1.2 InnoDB

MySQL soporta un segundo formato de tablas llamado InnoDB. Este motor de almacenamiento no es desarrollado por MySQL sino que pertenece a la empresa Innobase Oy que es la propietaria del software. InnoDB está bajo la licencia GPL v2. Es una alternativa moderna a las tablas MyISAM, que ofrece las siguientes funcionalidades adicionales:

- **Transacciones**. Las operaciones sobre las tablas InnoDB se pueden ejecutar como transacciones. Esto permite ejecutar varias operaciones SQL conectadas como una sola entidad. Si ocurre un error durante una transacción, todos los comandos de la transacción son anulados, no solo el que se estaba ejecutando en el monento de la transacción. Esto es muy útil para asegurar la seguridad de las aplicaciones sobre bases de datos.
- **Bloqueo a nivel de registro**. Para implementar transacciones, las tablas InnoDB usan internamente bloqueo a nivel de registro. Esto significa que durante una transacción no se necesita bloquear la tabla entera, y eso permite que otros usuarios puedan acceder a otros registros de la tabla simultáneamente. Esto permite una gran eficiencia en los casos en que muchos usuarios acceden a la vez a una misma tabla.
- **Restricciones sobre claves foráneas**. Cuando se definen relaciones entre tablas usando claves foráneas, las tablas InnoDB automáticamente aseguran la integridad referencial de los datos cuando se usan comandos DELETE. Por ejemplo, es imposible que un registro en una tabla A referencíe a otro inexistente en una tabla B.
- **Recuperación de datos perdidos**. Después de una caída del sistema, las tablas InnoDB se llevan a un estado consistente de forma automática y muy rápida

(siempre que el sistema de ficheros no haya quedado dañado).

Limitaciones de las tablas InnoDB:

- Administración del espacio de tablas. Mientras que las tablas MyISAM utilizan un fichero para cada tabla, que crece o se reduce según el uso, las tablas InnoDB almacenan todos los datos y los índices en un solo espacio de tablas, que puede estar en uno o varios ficheros, que forman una especie de sistema de ficheros virtual. Estos fichero no pueden reducirse. Tampoco es posible parar el servidor y copiar el fichero a otro servidor. Para trasladar una base de datos con tablas InnoDB se tiene que utilizar el comando `mysqldump`.
- Requerimiento de espacio. Las tablas InnoDB ocupan mucho más que las MyISAM.
- Indices full-text. Las tablas InnoDB no pueden usar índices full-text.
- Problema con `ANALIZE TABLE`. Es difícil establecer en un momento determinado el número excato de registros de una tabla. Además, ejecutar `SELECT COUNT(*) FROM TABLE` es mucho más lento que las tablas MyISAM. Esta limitación debería desaparecer en próximas versiones.
- Las columnas `AUTO_INCREMENT` llevan un índice por defecto, y no pueden formar parte de un índice multi-columna.
- Cuando se produce un INSERT en una tabla con columna AUTO_INCREMENT, se produce un bloqueo del índice entero llamado AUTO-INC. Durante este bloqueo no se pueden hacer nuevos INSERT hasta que termine el anterior. A partir de MySQL 5.1.22 las tablas InnoDB han mejorado mucho su velocidad a la hora de generar los valores `AUTO_INCREMENT`. Hasta dicha versión, las tablas InnoDB necesitaban de un bloqueo de tabla especial llamado `AUTO-INC` para garantizar la unicidad de los enteros generados. Este bloqueo era efectivo mientras duraba una instrucción `INSERT`, por lo que la inserción de muchas filas en una sola instrucción significaba el bloque entero de la tabla. Sin embargo, a partir de 5.1.22 se ha introducido un nuevo mecanismo que elimina la necesidad de usar el bloque de tabla AUTO_INC para INSERT de los que se sabe de antemano el número de registros a insertar (lo que ocurre la mayoría de las veces).
- Bloqueo de tablas. Las tablas InnoDB usan su propio algoritmo de bloqueo al ejecutar transacciones. Así pues, hay que evitar usar LOCK TABLE ... READ/WRITE. En su lugar hay que usar SELECT ... IN SHARE MODE o SELECT ... FOR UPDATE. Estos comandos tienen la ventaja adicional de que bloquean solo registros individuales, y no tablas enteras.
- Tablas mysql. Las tablas de la base de datos especial mysql no se pueden transformar a InnoDB y deben permanecer del tipo MyISAM.
- Hay un límite de 1023 transacciones concurrentes.
- Costes de licencia. Añadir soporte comercial de InnoDB a una licencia cuesta el doble.

A parte de todas estas características de tipo general que se encuentran en todas las versiones de MySQL, existen muchas características específicas que han aparecido en la

versión de InnoDB para MySQL 5.1 en su versión Plugin 1.0 (a partir de 5.1.23). Es importante recordar que la versión de InnoDB que viene con MySQL 5.1 no tiene estas características, y que solo la versión Plugin 1.0 las tiene:

- Creación rápida de índices. Si tenemos una tabla con datos, crear o borrar un índice es mucho más rápido. Hasta ahora, las instrucciones CREATE INDEX y DROP INDEX representaban crear una tabla nueva vacía para los nuevos índices, y entonces se copiaban uno a uno los nuevos.

  Con la nueva versión no se produce esta copia.

- Compresión de datos. No solo reduce el tamaño de las tablas en disco sino que permite tener más información efectiva en memoria reduciendo la entrada/salida.

- Columnas de longitud variable. Los registros de una tabla están guardados en un B-tree. Esta estructura de datos permite guardar elementos ordenados de manera que cada nodo puede contener un número variable de elementos. En la Figura 8 podemos ver un ejemplo.

Figura 8: Ejemplo de B-Tree

  La ordenación corresponde a la clave primaria. Los datos de cada registro se almacenan en los nodos. Los índices secundarios también son B-Trees que en sus nodos contienen referencias a los nodos del B-Tree “primario”.

  Sin embargo, cuando los registros tienen campos de tipo BLOB o VARCHAR que son demasiado grandes para almacenarlos en los nodos del árbol, se guardan en páginas de disco aparte (“overflow”) en forma de lista enlazada.

- Se introducen siete tablas nuevas en la base de datos INFORMATION_SCHEMA que permiten conocer el estado del motor InnoDB en un momento determinado. Estas tablas son `INFORMATION_SCHEMA` tables `INNODB_CMP`, `INNODB_CMP_RESET`, `INNODB_CMPMEM`, `INNODB_CMPMEM_RESET`, `INNODB_TRX`, `INNODB_LOCKS` y `INNODB_LOCK_WAITS` contienen información puntual sobre las tablas comprimidas, el buffer pool comprimido, las transacciones que se están ejecutando en el momento, los bloqueos correspondientes a las transacciones, y a qué bloqueo está esperando cada transacción.

4.1.3 MyISAM vs InnoDB

En MySQL se pueden especificar los tipos de tabla individualmente, de manera que diferentes tipos de tabla pueden ser usados simultáneamente en una misma base de datos. Eso permite escoger el tipo óptimo para cada tabla dependiendo de la aplicación que las vaya a usar.

Las tablas MyISAM están recomendadas para casos en los que se necesita el máximo de velocidad y el espacio de almacenamiento es pequeño. Por otro lado, las tablas InnoDB están recomendadas cuando nuestra aplicación necesita usar transacciones, necesita más seguridad sobre los datos, o va a ser accedida por muchos usuarios a la vez para hacer modificaciones.

No hay una respuesta general sobre la pregunta de qué tipo de tabla ofrece una respuesta más rápida. En principio, debido a que las transacciones requieren un tiempo extra y las tablas InnoDB ocupan más espacio en disco, MyISAM deberían tener ventaja. Pero con las tablas InnoDB se puede evitar el bloqueo de tablas, lo que le da una ventaja a InnoDB en ciertas situaciones.

Además, la velocidad de una aplicación depende mucho en el tipo de hardware donde se ejecuta, la configuración del servidor MySQL y otros factores. Normalmente, la única manera de obtener una respuesta para una determinada aplicación es hacer pruebas de velocidad extensivas usando ambos tipos de tablas.

Sin embargo, la gran diferencia entre estos dos tipos de tablas que puede afectar seriamente a la velocidad es el hecho de que InnoDB dispone de bloqueo por registro mientras que MyISAM solo lo tiene por tabla. Si nuestra aplicación realiza muchos INSERT o UPDATE mientras a la vez también hay muchos SELECT,el uso de MyISAM puede verse muy penalizado ya que los bloqueos por tabla por cada modificación hará que se bloqueen las lecturas. En este caso las tablas InnoDB son más recomendables. SI por otro lado, nuestra aplicación tiene muchos menos INSERT y UPDATE comparados con los SELECT, los bloqueos de tabla en MyISAM se notarán poco y nos puede compensar la velocidad extra en los SELECT de este tipo de tabla.

Otra diferencia importante es la integridad de los datos cuando hay caídas del sistema. El motor MyISAM no tiene ningún mecanismo para recuperarse, mientras que InnoDb sí.

En general, las ventajas de InnoDB son:

- **Prestaciones**. A pesar de considerarse históricamente más lento que MyISAM, las recientes optimizaciones han hecho que la diferencia sea muy pequeña, a pesar de ser un motor transaccional.
- **Concurrencia**. Sobre todo cuando se mezclan INSERT y SELECT. El mecanismo de bloqueo por registro da una enorme ventaja a InnoDB.
- **Fiabilidad**. El tener transacciones ACID y la capacidad para recuperarse de caídas del sistema es una gran ventaja en este motor.
- **Seguridad de los datos**. InnoDB puede hacer respaldos con muchos menos bloqueos que las tablas MyISAM

Y las ventajas de MyISAM son:

- **Simplicidad**. El motor es muy sencillo y fácil de entender lo que facilita desarrollar complementos externos para él.
- **Optimización**. Al ser el motor que más tiempo lleva con MySQL está muy optimizado y muchas aplicaciones están escritas pensando en este motor. Y la búsqueda fulltext les una gran ventaja sobre InnoDB.
- **Uso de recursos**. En general consumen menos CPU y menos espacio de disco.

4.1.4 MEMORY

Las tablas MEMORY son un tipo de tablas muy específico cuya característica fundamental es que siempre residen en memoria RAM, y nunca en el disco. Usan un índice hash que les da un tiempo de acceso muy rápido a registros individuales. Estas tablas se usan normalmente para almacenar datos temporales.

En comparación con las tablas normales, las tablas MEMORY presentan un gran número de restricciones, de la cual la más importante es que no pueden usar columnas del tipo TEXT o BLOB. Los registros solo se pueden buscar usando = ó <=>. AUTO_INCREMENT no se puede usar. Los índices solo se pueden crear para columnas NOT NULL.

Estas tablas solo deberían usarse cuando se necesite máxima velocidad para acceder a un conjunto pequeño de datos. Al ser tablas que se guardan en memoria RAM, desaparecen cuando MySQL termina. El tamaño máximo de estas tablas está determinado en el fichero de configuración por el parámetro `max_heap_table_size`.

4.1.5 NBDCLUSTER

Este motor está pensado para aplicaciones que necesitan un gran tráfico de datos, con mucha concurrencia, y con niveles muy altos de disponibilidad y seguridad. Se construye usando un conjunto de ordenadores entre los que no hay compartición de recursos. La característica fundamental es que los datos se almacenan en memoria.

Desde 5.1.24 este motor no se distribuye en la versión estándar de MySQL, sino que se hace únicamente en la versión MySQL-Cluster. Podemos ver la arquitectura de este sistema en la Figura 9.

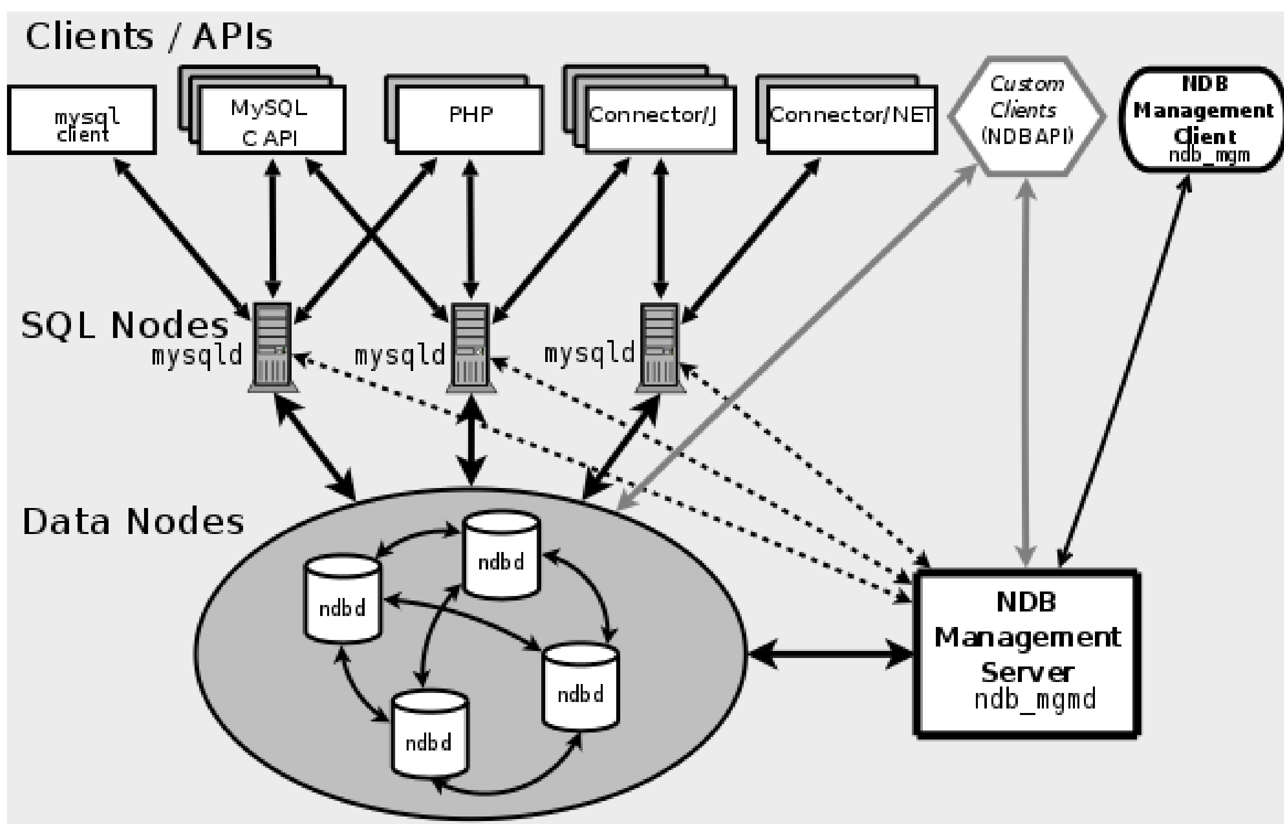


Figura 9: Arquitectura de MySQL-Cluster

MySQL está compuesto de una serie de nodos que realizan diferentes funciones. Hay tres tipos de nodos:

- Nodos de control. Normalmente hay solo uno excepto para configuraciones muy grandes. Este nodo se encarga de controlar a los demás, de iniciarlos y pararlos, de hacer respaldos, y de las tareas administrativas en general. Es el primero en iniciarse y corresponde al programa `ndb_mgmd`.

- Nodos de datos. Son los que guardan los datos de las tablas en memoria. Cada nodo guarda una copia de una parte de la base de datos. Se inicia con el programa `ndbd`.
- Nodos SQL. Son nodos que ejecutan servidores MySQL con motor `NDB` y que hacer de interfície para las aplicaciones de manera transparente. Se inicia con el programa `ndbdcluster`.

Comparación con replicación

Este sistema tiene parecidos con hacer replicación, pero hay una serie de diferencias muy importantes. Al hacer replicación tenemos un servidor maestro y N esclavos, con lo cual, en situaciones de mucha carga, todo ha de pasar por un solo nodo maestro. Con MySQL-Cluster no existe este cuello de botella ya que el trabajo está distribuido entre muchos nodos.

Después está el problema de que con replicación todas las transacciones se ejecutan de forma secuencial, lo que impide tener concurrencia a ese nivel. En MySQL-Cluster las transacciones se ejecutan de manera concurrente.

Otra de las ventajas de MySQL-Cluster sobre replicación es que garantiza la sincronización de los los datos.

4.2 Tipos de datos

Cada tabla está compuesta por un número de columnas. Para cada columna hay que especificar un tipo de datos. Los tipos de datos en MySQL se pueden clasificar en cuatro tipos:

1. Numéricos
2. Fecha y tiempo
3. Cadenas de caracteres
4. GIS

En esta sección daremos una breve descripción de los tipos específicos dentro de cada categoría.

4.2.1 Enteros (xxxINT)

Es un tipo numérico con el que se pueden representar número enteros positivos y negativos. Con el atributo UNSIGNED el rango se restringe a números positivos. MySQL soporta los tipos enteros estándar de SQL: `INTEGER`, `SMALLINT`, `DECIMAL`, `NUMERIC`. Como extensión, MySQL soporta los tipos adicionales `TINYINT`, `MEDIUMINT`, y `BIGINT`. Los rangos de valores de cada tipo están detallados en la Tabla 1.

Tipo MySQL	Significado
TINYINT(m)	Entero de 8 bits (1 byte, -128 a +127); el argumento m es el ancho de la columna en los resultados SELECT, pero no influye en el rango de números representados.
SMALLINT(m)	Entero de 16-bits (2 bytes, -32.768 a +32.767)
MEDIUMINT(m)	Entero de 24-bits (3 bytes, -8.388.608 a +8.388.607)
INT(m), INTEGER(m)	Entero de 32-bits (4 bytes, -2.147.483.648 a +2.147.483.647)
BIGINT(m)	Entero de 64-bits (8 bytes, ±9.22*10^18)
SERIAL	Sinónimo de BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY

Tabla 1: Tipos de enteros

Otra extensión de MySQL es uso de la forma INT(n) en el que se determina el número mínimo n de caracteres que se usará para mostrar ese entero. Si el entero tiene menos de n cifras, se rellenarán con espacios. SI se ha especificado la opción ZEROFILL entonces se rellenará con ceros. Por ejemplo, una columna de tipo INT(5) ZEROFILL mostrará el valor 5 como 00005. Además, ZEROFILL implica automáticamente UNSIGNED.

4.2.2 Enteros AUTO_INCREMENT

Cuando se usa el atributo opcional AUTO_INCREMENT se consigue que MySQL, cada vez que se crea un nuevo registro, automáticamente inserte un valor que es 1 más que el valor más grande en la columna correspondiente. Esto se usa generalmente para definir columnas que se utilizarán como claves primarias de la tabla. Las reglas que rigen este atributo son:

- Solo se puede usar cuando uno de los atributos NOT NULL, PRIMARY KEY, o UNIQUE se usa también.
- Una tablas solo puede tener una columna AUTO_INCREMENT.
- La generación automática de un nuevo valor solo funciona cuando se crean nuevos registros usando INSERT y no se proporciona un valor específico o NULL.
- Para saber el último valor generado automáticamente después de una instrucción INSERT, ejecutar el comando SELECT LAST_INSERT_ID(). Un detalle importante es que en un INSERT múltiple, esta función retorna el entero generado para el primer registro.
- Si el contador de AUTO_INCREMENT alcanza el valor máximo, dependiendo del tipo de entero que estemos usando, no se incrementará más. No se permitirán más operaciones de INSERT. Con tablas sobre las que se realizan muchos INSERT y DELETE, puede ocurrir que un entero de 32-bits INT alcance su valor máximo, independientemente de que haya pocos registros en la tabla. En estos casos hay que usar un BIGINT.

En las tablas MyISAM se puede especificar una columna secundaria como AUTO_INCREMENT cuando forma parte de un índice multicolumna. En este caso, el valor generado por la columna AUTO_INCREMENT se calcula como:

```
MAX(columa_auto_increment) + 1 WHERE prefix=given-prefix
```

Esto es útil cuando se quiere almacenar datos ordenados por grupos. Por ejemplo:

```
CREATE TABLE animals (
    grp ENUM('fish','mammal','bird') NOT NULL,
    id MEDIUMINT NOT NULL AUTO_INCREMENT,
    name CHAR(30) NOT NULL,
    PRIMARY KEY (grp,id)
);

INSERT INTO animals (grp,name) VALUES
    ('mammal','dog'),('mammal','cat'),
    ('bird','penguin'),('fish','lax'),('mammal','whale'),
    ('bird','ostrich');

SELECT * FROM animals ORDER BY grp,id;
```

Que retorna:

```
+--------+----+---------+
| grp    | id | name    |
+--------+----+---------+
| fish   |  1 | lax     |
| mammal |  1 | dog     |
| mammal |  2 | cat     |
| mammal |  3 | whale   |
| bird   |  1 | penguin |
| bird   |  2 | ostrich |
+--------+----+---------+
```

4.2.3 Datos binarios (BIT y BOOL)

En MySQL, el tipo de datos BOOL es sinónimo de TINYINT. Esto también era cierto con BIT hasta la versión 5.0.2. A partir de esa versión BIT es un tipo de datos aparte para representar información binaria hasta 64 bits.

4.2.4 Números en coma flotante (FLOAT y DOUBLE)

Desde la versión 3.23 de MySQL, los tipos `FLOAT` y `DOUBLE` corresponden a los tipos numéricos IEEE de precisión normal y doble (*single and double*) que usan la mayoría de lenguajes de programación.

Opcionalmente, el número de dígitos en `FLOAT` y `DOUBLE` se puede determinar con los parámetros `m` y `d`. En ese caso, `m` especifica el número de dígitos antes del punto decimal, y d determina el número de dígitos después del punto digital. Las características generales de estos tipos están resumidas en la Tabla 2.

Tipo de datos	Significado
`FLOAT(m, d)`	Número en coma flotante, precisión de 4 bytes. Los valores `m` y `d` solo afectan a como son presentados los valores en un SELECT, pero no afectan a su valor almacenado.
`DOUBLE(m, d)`	Número en coma flotante, precisión de 8 bytes.
`REAL(m, d)`	Sinónimo de DOUBLE

Tabla 2: Número en coma flotante

4.2.5 Números de coma fija (DECIMAL)

Este tipo se usa cuando el redondeo que se produce en operaciones con los tipos FLOAT y DOUBLE es inaceptable, por ejemplo, cuando manejamos precios. Los números son almacenados como cadenas de caracteres y por ello el coste de almacenamiento es mucho más alto. Además, el rango de valores es más pequeño ya que no se puede usar la notación exponencial.

4.2.6 Fecha y hora (DATE, TIME, DATETIME, TIMESTAMP)

Los tipos de datos para almacenar datos temporales están resumidos en la Tabla 3.

Tipo de datos	Significado
`DATE`	Fecha en la forma '2008-12-31', rango desde 1000-01-01 hasta 9999-12-31 (3 bytes)
`TIME`	Tiempo en la forma '23:59:59', rango ±838:59:59 (3 bytes)
`DATETIME`	Combinación de `DATE` más `TIME`
`YEAR`	Año 1900-2155 (1 byte)
`TIMESPAMP`	Lo mismo que `DATETIME`, pero con inicialización automática al día y hora en que se hace la modificación

Tabla 3: Especificación de tipos que representan fecha y hora

A partir de la versión 5.0.2 de MySQL hay un mecanismo muy potente de validación de fechas, de forma que solo fechas válidas pueden ser almacenadas. Aun así se permite colocar un 0 para el día o el mes, y también la fecha 0000-00-00. Sin embargo, hay atributos que permiten anular estas posibilidades. La Tabla 4 resume estos atributos.

Atributo	Significado
ALLOW_INVALID_DATES	Permite fechas incorrectas, como 2008-02-31.
NO_ZERO_DATE	No permite la fecha 0000-00-00.
NO_ZERO_IN_DATE	No permite el 0 como mes o día.

Tabla 4: Atributos para DATE

El tipo TIMESTAMP tiene una serie de particularidades. La fundamentas es que es modificado cada vez que el registro al que pertenece es modificado y de esa manera refleja el momento de la última modificación. Por eso, este tipo solo suele usarse para el control interno de los datos, no como dato “real”. Hay mucha librerías que solo funcionarán si cada tabla tiene una columna del tipo TIMESTAMP.

La actualización automática no requiere de ninguna función especial, y la columna no debe de tener ningún valor explícito almacenado, ni un NULL.

4.2.7 Cadenas de caracteres (CHAR, VARCHAR, xxxTEXT)

La Tabla 5 resume los diferentes tipos de datos que se usan para almacenar cadenas de caracteres.

Tipo	Significado
CHAR(n)	Cadena de caracteres con longitud específica n, máximo 255.
VARCHAR(n)	Cadena de caracteres de longitud variable, máximo n caracteres, con n<256 en versiones hasta 4.1, y n<65536 para versiones 5.0.3 o superiores
TINYTEXT	Cadena de caracteres de longitud variable, máximo 255 bytes.
TEXT	Cadena de caracteres de longitud variable, máximo $2^{16}-1$
MEDIUMTEXT	Cadena de caracteres de longitud variable, máximo $2^{24}-1$
LONGTEXT	Cadena de caracteres de longitud variable, máximo $2^{32}-1$

Tabla 5: Tipos de cadenas de caracteres

Con CHAR, la longitud de la cadena es fija y está definida estrictamente. Por ejemplo, CHAR(20) guardará 20 bytes en cada registro, independientemente de si todos se usan. Al contrario, las cadenas VARCHAR tienen longitud variable y los requerimientos de almacenamiento varían según el tamaño real de la cadena.

Aunque los tipos VARCHAR y TEXT puedan parecer idénticos, tienen una diferencia significativa. En el caso de VARCHAR, el tamaño máximo tiene que ser especificado por el usuario en el momento de definir la tabla. Las cadenas que sean más largas serán truncadas sin ningún aviso.

A partir de MySQL 5.0 hay dos novedades importantes para VARCHAR:

- El tamaño máximo es 65535 bytes, mientras que antes era de 255. Es importante

resaltar que el tamaño máximo está medido en bytes, por lo que el máximo de caracteres puede ser menor ya que según el tipo de codificación se necesita más de 1 byte para codificar un carácter.

- Los espacios al principio y final de la cadena ahora son almacenados, mientras que antes eran eliminados antes de almacenar la cadena.

Los tipos CHAR y VARCHAR pueden llevar el atributo BINARY. En este caso la columna se comporta esencialmente como un BLOB (ver más delante). Este atributo puede ser útil ya que cuando se usa, el criterio de ordenación se rije exclusivamente por el valor binario de los bytes de la cadena, y no depende del juego de caracteres que estemos usando.

Juegos de caracteres

En las columnas de tipo texto se puede usar el atributo adicional:

```
CHARACTER_SET nombre_juego_caracteres COLLATE criterio_ordenación
```

Los juegos de caracteres especifican la manera en que los caracteres son codificados y el criterio para ordenar las cadenas de caracteres. La mayoría de los juegos de caracteres tienen en común la codificación de los 128 caracteres “ingleses” de ASCII. El problema comienza con la codificación de los caracteres internacionales. Desde la perspectiva “euro-anglosajona”, hay dos grandes familias de juegos de caracteres:

- **Juegos de caracteres latinos**. En el pasado, cada región desarrolló su propia codificación de 1 byte, de las cuales el juego *Latin* ha sido el más extendido: *Latin1* (ISO-8859-1) los caracteres más comunes de Europa occidental (äöüXßáàå etc). *Latin2* (ISO-8859-2) contiene caracteres de idiomas de la Europa de este y oriental. *Latin0* (o *Latin9*, ISO-8859-15) es el mismo que *Latin1* pero con el símbolo del Euro añadido (€).

  El problema con estos juegos es que ninguno contiene todos los caracteres de los idiomas europeos.

- **Unicode**. Para resolver el problema de los juegos Latin se creó el juego Unicode que usa 2 bytes por carácter. Con 65536 caracteres posibles, cubre no solo Europa sino la mayoría de los idiomas asiáticos.

  Pero no podía ser tan fácil ... Unicode solo determina qué código está asociado a cada carácter, no como los códigos se guardan internamente. Por ello hay diferentes variantes, de las cuales UCS-2 (Universal Character Set) y UTF-8 (Unicode transfer format) son las más importantes.

  UCS-2, también llamado UTF-16, representa lo que aparentemente es la solución más simple, que es usar 2 bytes para codificar cada carácter. Sin embargo, tiene dos inconvenientes: el espacio para almacenar cadenas de caracteres se duplica, incluso cuando se están representando cadenas de idiomas europeos. Segundo, el segundo byte usualmente es 0, especialmente cuando se representan cadenas en inglés. Muchos programas escritos en C consideran que un carácter 0 significa el final de una cadena, lo cual puede dar lugar a problemas.

  Por esto, UTF-8 es la alternativa más popular a UTF-16. En este caso, los caracteres ASCII se representan con un solo byte, cuyo primer bit es 0. El resto de caracteres Unicode se representan con 2 ó 4 bytes. La desventaja de este formato es que no hay una relación directa entre el tamaño en bytes de una cadena y su

tamaño en caracteres. Este formato es el más estándar para representar Unicode.

A pesar de las ventajas de Unicode, no todo es sencillo. El principal problema es que las cadenas de caracteres en Unicode son incompatibles con las clásicas en ASCII. Además, el soporte a Unicode en algunas herramientas web no está presente. Por ejemplo, solo la última versión de PHP (5.2) incorpora soporte para Unicode.

Una vez se ha escogido un juego de caracteres, también se puede escoger el criterio de ordenación. Esto es debido a que un mismo juego de caracteres contiene elementos de muchos idiomas al mismo tiempo, por lo que cuando queremos ordenar alfabéticamente un conjunto de cadenas de caracteres, dependerá del idioma querremos un resultado u otro. Para determinar el orden alfabético a usar con un determinado juego de caracteres usaremos el atributo COLLATE al definir un juego de caracteres. Para ver todas las opciones que tiene nuestro servidor usaremos SHOW COLLATE:

```
mysql> show collation;
+----------------------+----------+-----+---------+----------+---------+
| Collation            | Charset  | Id  | Default | Compiled | Sortlen |
+----------------------+----------+-----+---------+----------+---------+
| big5_chinese_ci      | big5     |   1 | Yes     | Yes      |       1 |
| big5_bin             | big5     |  84 |         | Yes      |       1 |
| dec8_swedish_ci      | dec8     |   3 | Yes     | Yes      |       1 |
| dec8_bin             | dec8     |  69 |         | Yes      |       1 |
| cp850_general_ci     | cp850    |   4 | Yes     | Yes      |       1 |
| cp850_bin            | cp850    |  80 |         | Yes      |       1 |
| hp8_english_ci       | hp8      |   6 | Yes     | Yes      |       1 |
| hp8_bin              | hp8      |  72 |         | Yes      |       1 |
| koi8r_general_ci     | koi8r    |   7 | Yes     | Yes      |       1 |
| koi8r_bin            | koi8r    |  74 |         | Yes      |       1 |
| latin1_german1_ci    | latin1   |   5 |         | Yes      |       1 |
| latin1_swedish_ci    | latin1   |   8 | Yes     | Yes      |       1 |
| latin1_danish_ci     | latin1   |  15 |         | Yes      |       1 |
| latin1_german2_ci    | latin1   |  31 |         | Yes      |       2 |
| latin1_bin           | latin1   |  47 |         | Yes      |       1 |
| latin1_general_ci    | latin1   |  48 |         | Yes      |       1 |
| latin1_general_cs    | latin1   |  49 |         | Yes      |       1 |
| latin1_spanish_ci    | latin1   |  94 |         | Yes      |       1 |
| latin2_czech_cs      | latin2   |   2 |         | Yes      |       4 |
| latin2_general_ci    | latin2   |   9 | Yes     | Yes      |       1 |
| latin2_hungarian_ci  | latin2   |  21 |         | Yes      |       1 |
| latin2_croatian_ci   | latin2   |  27 |         | Yes      |       1 |
| latin2_bin           | latin2   |  77 |         | Yes      |       1 |
| swe7_swedish_ci      | swe7     |  10 | Yes     | Yes      |       1 |
| swe7_bin             | swe7     |  82 |         | Yes      |       1 |
| ascii_general_ci     | ascii    |  11 | Yes     | Yes      |       1 |
| ascii_bin            | ascii    |  65 |         | Yes      |       1 |
| ujis_japanese_ci     | ujis     |  12 | Yes     | Yes      |       1 |
| ujis_bin             | ujis     |  91 |         | Yes      |       1 |
| sjis_japanese_ci     | sjis     |  13 | Yes     | Yes      |       1 |
| sjis_bin             | sjis     |  88 |         | Yes      |       1 |
| hebrew_general_ci    | hebrew   |  16 | Yes     | Yes      |       1 |
| hebrew_bin           | hebrew   |  71 |         | Yes      |       1 |
| tis620_thai_ci       | tis620   |  18 | Yes     | Yes      |       4 |
| tis620_bin           | tis620   |  89 |         | Yes      |       1 |
| euckr_korean_ci      | euckr    |  19 | Yes     | Yes      |       1 |
| euckr_bin            | euckr    |  85 |         | Yes      |       1 |
| koi8u_general_ci     | koi8u    |  22 | Yes     | Yes      |       1 |
| koi8u_bin            | koi8u    |  75 |         | Yes      |       1 |
| gb2312_chinese_ci    | gb2312   |  24 | Yes     | Yes      |       1 |
| gb2312_bin           | gb2312   |  86 |         | Yes      |       1 |
| greek_general_ci     | greek    |  25 | Yes     | Yes      |       1 |
| greek_bin            | greek    |  70 |         | Yes      |       1 |
| cp1250_general_ci    | cp1250   |  26 | Yes     | Yes      |       1 |
| cp1250_czech_cs      | cp1250   |  34 |         | Yes      |       2 |
```

```
| cp1250_croatian_ci   | cp1250   |  44 |     | Yes |       1 |
| cp1250_bin           | cp1250   |  66 |     | Yes |       1 |
| cp1250_polish_ci     | cp1250   |  99 |     | Yes |       1 |
| gbk_chinese_ci       | gbk      |  28 | Yes | Yes |       1 |
| gbk_bin              | gbk      |  87 |     | Yes |       1 |
| latin5_turkish_ci    | latin5   |  30 | Yes | Yes |       1 |
| latin5_bin           | latin5   |  78 |     | Yes |       1 |
| armscii8_general_ci  | armscii8 |  32 | Yes | Yes |       1 |
| armscii8_bin         | armscii8 |  64 |     | Yes |       1 |
| utf8_general_ci      | utf8     |  33 | Yes | Yes |       1 |
| utf8_bin             | utf8     |  83 |     | Yes |       1 |
| utf8_unicode_ci      | utf8     | 192 |     | Yes |       8 |
...
```

Como se puede ver, para cada juego de caracteres existen diferentes opciones de orden, normalmente una por idioma.

4.2.8 Datos binarios (xxxBLOB y BIT)

Para el almacenamiento de datos binarios hay cuatro variantes del tipo BLOB, de manera similar al tipo TEXT. La diferencia es que los datos binarios siempre se ordenan y se compran usando directamente su valor sin mediar ninguna codificación.

El uso de este tipo tiene ventajas y desventajas. Si queremos almacenar imágenes o sonido, podemos usar el tipo BLOB. La ventaja es que los datos están integrados en la base datos, con lo cual entran dentro de los backups, de las recuperaciones cuando hay caídas del sistema, .. etc. Sin embargo, ralentiza el funcionamiento de la base de datos. Otra desventaja es que normalmente los datos BLOB solo se pueden leer enteros, es decir, no se pueden leer partes de ellos.

La alternativa a usar el tipo BLOB es tener los datos binarios en ficheros externos.

La Tabla 6 muestra las características de las variantes del tipo BLOB.

Tipo	Significado
`BIT(n)`	Datos en bits, donde n es el número de bits (máximo 64)
`TINYBLOB`	Datos binarios de tamaño variable, máximo 255 bytes
`BLOB`	Datos binarios de tamaño variable, máximo $2^{16}-1$ bytes
`MEDIUMBLOB`	Datos binarios de tamaño variable, máximo $2^{24}-1$ bytes
`LONGBLOB`	Datos binarios de tamaño variable, máximo $2^{32}-1$ bytes

Tabla 6: Tipos para datos binarios

4.2.9 Otros tipos

Hay dos tipos de datos que son particulares de MySQL: ENUM y SET. Permiten el manejo eficiente de conjuntos y enumeraciones.

Con ENUM se puede representar una lista de hasta 65535 cadenas de caracteres a las que se asignan números enteros consecutivos (similar al tipo ENUM de C).

El tipo SET es parecido, pero además distingue el orden en que los elementos están

dispuesto, permitiendo la representación de combinaciones. Requiere más espacio de almacenamiento y además solo puede manejar como máximo 64 valores.

4.2.10 Opciones y atributos

Hay una gran variedad de opciones y atributos que se pueden especificar cunado una columna es creada. En la Tabla podemos ver una lista de los más importantes. Hay que considerar que no todos los atributos se pueden aplicar a todos los tipos de datos.

Opción o atributo	Significado
`NULL`	La columna puede contener valores nulos (funciona por defecto).
`NOT NULL`	El valor NULL no está permitido.
`DEFAULT xxx`	El valor por defecto xxx se usará si no se especifica uno.
`DEFAULT CURRENT_TIMESTAMP`	Para columnas TIMESTAMP, la hora actual se almacenará cuando se creen nuevos registros
`ON UPDATE CURRENT_TIMESTAMP`	Para columnas TIMESTAMP, la hora actual se almacenará cuando se produzca una modificación del registro.
`PRIMARY_KEY`	Define la columna como clave primaria.
`AUTO_INCREMENT`	Se asigna un entero secuencialmente usar con columnas tipo INTEGER. Además, ha de ir acompañado de NOT NULL y PRIMARY_KEY.
`UNSIGNED`	Para valores INTEGER, no permite valores negativos.
`CHARACTER SET nombre [COLLATE orden]`	Para cadenas de caracteres, especifica el juego de caracteres, y opcionalmente el orden alfabético a usar.

4.2.11 Datos GIS

GIS significa *Geographical Information System,* decir, sistemas de información geográfica. MySQL tiene una serie de tipos de datos que sirven para almacenar datos que representan información geográfica, como puntos, líneas, polígonos, ...

MySQL implementa un subconjunto de ***SQL with Geometry Types*** que representa un entorno para almacenar información espacial en bases de datos relacionales. Las especificaciones de este entorno han sido publicadas por el consorcio OpenGIS y pueden ser consultadas aquí:

http://www.opengis.org/docs/99-049.pdf

Los objetos de tipo GIS tienen dos características fundamentales:

- Deben estar asociados a un sistema de referencia espacial.
- Deben ser parte de una jerarquía de objetos geométricos.

La jerarquía disponible en MySQL es la siguiente:

- Geometry (no instanciable)
    - Point (instanciable)
    - Curve (no instanciable)
        - LineString (instanciable)
            - Line (instanciable)
            - LineRing (instanciable)
    - Surface (no instanciable)
        - Polygon (instanciable)
    - GeometryCollection (instanciable)
        - MultiPoint (instanciable)
        - MultiCurve (no instanciable)
            - MultiLineString (instanciable)
        - MultiSurface (no instanciable)
            - MultiPolygon (instanciable)

El calificativo de instanciable para un tipo se refiere a que es posible declarar una columna de dicho tipo. Si es no instanciable quiere decir que no se puede declarar una columna de ese tipo.

Los datos de tipo GIS se pueden manejar en dos formatos definidos por OpenGIS:

- WKT (*Well Known Text*): un formato de texto
- WKB (*Well Known Binary*): un formato binario

Estos formatos se utilizan para manejar los datos que recuperamos de MySQL, pero el servidor los almacena en otro formato diferente internamente.

El formato WKT representa los objetos geométricos de la siguiente manera:

```
POINT(15 20)
LINESTRING(0 0, 10 10, 20 25, 50 60)
POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))
MULTIPOINT(0 0, 20 20, 60 60)
MULTILINESTRING((10 10, 20 20), (15 15, 30 15))
MULTIPOLYGON(((0 0,10 0,10 10,0 10,0 0)),((5 5,7 5,7 7,5 7, 5 5)))
GEOMETRYCOLLECTION(POINT(10 10), POINT(30 30), LINESTRING(15 15, 20 20))
```

Y el formato WKB utiliza enteros de 1 byte y de 4 bytes, y número en coma flotante de 64 bits. Por ejemplo, el objeto 'POINT(1 1)' se representa como:

```
0101000000000000000000F03F000000000000F03F

Que corresponde a:

Byte order : 01
WKB type   : 01000000
X          : 000000000000F03F
Y          : 000000000000F03F
```

El primer bytes es 0 ó 1 para indicar si los datos se almacenan en *little-endian* o *big-endian*. Los cuatro bytes siguientes son un entero que representa el tipo de objeto. Los 16 bytes siguiente son dos número en coma flotante con precisión de 64 bits.

Estos tipos de datos se pueden usar de la misma manera que el resto de tipos de MySQL. Para crear una tabla con columnas de tipo GIS haremos:

```
CREATE TABLE geom (g GEOMETRY)
```

Añadir y eliminar columnas:

```
ALTER TABLE geom ADD pt POINT
ALTER TABLE geom DROP pt
```

Insertar registros:

```
INSERT INTO geom VALUES (GeomFromText('POINT(1 1)'));

SET @g = GeomFromText('POINT(1 1)');
INSERT INTO geom VALUES (@g);

SET @g = 'POINT(1 1)';
INSERT INTO geom VALUES (PointFromText(@g))
```

Leer registros:

```
SELECT g FROM geom
SELECT AsText(g) FROM geom
SELECT AsBinary(g) FROM geom
```

5. Modos SQL

Cuando se arranca el servidor `mysqld` se puede hacer en diferentes modos, y se puede aplicar estos modos de forma distinta a diferentes clientes. Esto permite que cada aplicación ajuste el modo de operación del servidor a sus propios requerimientos.

Los modos definen qué sintaxis SQL debe soportar MySQL y que clase de comprobaciones de validación de datos debe realizar. Esto hace más fácil de usar MySQL en distintos entornos y usar MySQL junto con otros servidores de bases de datos.

Se puede especificar el modo SQL por defecto arrancando `mysqld` con la opción `--sql-mode="modes"`. El valor puede dejarse en blanco (`--sql-mode=""`) si se desea resetearlo.

A partir de MySQL 5.0, también se puede cambiar el modo SQL una vez arrancado el servidor cambiando la variable `sql_mode` usando el comando `SET [SESSION|GLOBAL] sql_mode='modes'` . Asignar la variable `GLOBAL` requiere el privilegio `SUPER` y afecta las operaciones de todos los clientes que conecten a partir de entonces. Asignar la variable `SESSION` afecta sólo al cliente actual. Cualquier cliente puede cambiar el valor de `sql_mode` en su sesión en cualquier momento.

`modes` es una lista de los diferentes modos separados por comas (`','`) . Se puede consultar el modo actual mediante el comando `SELECT @@sql_mode` . El valor por defecto es vacío (sin modo seleccionado).

Los valores de los modos `sql_mode` más importantes son los siguientes:

- `ANSI:` Cambia el comportamiento y la sintaxis para cumplir mejor el estándar SQL.
- `STRICT_TRANS_TABLES:` Si un valor no puede insertarse tal y como se da en una tabla transaccional, se aborta el comando. Para tablas no transaccionales, aborta el comando si el valor se encuentra en un comando que implique un sólo registro o el primer registro de un comando de varios registros.
- `TRADITIONAL:` Hace que MySQL se comporte como un sistema de bases de datos SQL ``tradicional''. Una simple descripción de este modo es `` da un error en lugar de un aviso'' cuando se inserta un valor incorrecto en la columna. **Nota**: `INSERT/UPDATE` aborta así que se detecta un error. Puede que no sea lo que se quiere si está usando un motor de almacenamiento no transaccional, ya que los cambios en los datos anteriores al error no se deshacen, resultando en una actualización ``parcial'' .

Cuando nos referimos al “modo estricto”, implica un modo donde al menos `STRICT_TRANS_TABLES` o `STRICT_ALL_TABLES` está activado..

La siguiente lista describe todos los modos soportados:

- `ALLOW_INVALID_DATES`

  No hace una comprobación total de los datos en modo estricto. Comprueba sólo que los meses se encuentran en el rango de 1 a 12 y que los días están en el rango de 1 a 31. Esto es muy conveniente para aplicaciones Web donde obtenemos un año, mes y día en tres campos distintos y quiere guardar exactamente lo que inserta el usuario (sin validación de datos). Este modo se

aplica a columnas DATE y DATETIME . No se aplica a columnas TIMESTAMP , que siempre requieren una fecha válida.

Este modo se implementó en MySQL 5.0.2. Antes de 5.0.2, este era el modo por defecto de MySQL para tratar datos. Desde 5.0.2, el permitir el modo estricto provoca que el servidor requiera que el mes y día se evalúen como valores legales y no simplemente en los rangos de 1 a 12 y de 1 a 31, respectivamente. Por ejemplo, '2004-04-31' es legal con el modo estricto desactivado, pero ilegal con el modo estricto activado. Para permitir tales fechas en modo estricto, habilite ALLOW_INVALID_DATES también.

- ANSI_QUOTES

  Trata '"' como un identificador delimitador de carácter (como '`' ) y no como un delimitador de cadenas de caracteres. Puede usar '`' para delimitar identificadores en modo ANSI. Con ANSI_QUOTES activado, puede usar doble delimitadores para delimitar una cadena de caracteres literales, ya que se interpreta como un identificador.

- ERROR_FOR_DIVISION_BY_ZERO

  Produce un error en modo estricto (de otra forma una advertencia) cuando encuentra una división por cero (o MOD(X,0)) durante un INSERT o UPDATE, o en cualquier expresión (por ejemplo, en una lista de SELECT o cláusula WHERE) que implica datos de tablas y una división por cero. Si este modo no se da, MySQL retorna NULL para una división por cero. Si se usa INSERT IGNORE o UPDATE IGNORE, MySQL genera una advertencia de división por cero, pero el resultado de la operación es NULL. (Implementado en MySQL 5.0.2)

- HIGH_NOT_PRECEDENCE

  Desde MySQL 5.0.2 , la precedencia del operador NOT se trata tal que expresiones como NOT a BETWEEN b AND c se parsean como NOT (a BETWEEN b AND c). Antes de MySQL 5.0.2, la expresión se parseaba como (NOT a) BETWEEN b AND c. El antiguo comportamiento de mayor-precedencia puede obtenerse permitiendo el modo SQL HIGH_NOT_PRECEDENCE . (Añadido en MySQL 5.0.2)

```
mysql> SET sql_mode = '';
mysql> SELECT NOT 1 BETWEEN -5 AND 5;
        -> 0
mysql> SET sql_mode = 'broken_not';
mysql> SELECT NOT 1 BETWEEN -5 AND 5;
        -> 1
```

- IGNORE_SPACE

  Permite nombres entre el nombre de función y el carácter '(' . Esto fuerza que todos los nombres de función se traten como palabras reservadas. Como resultado, si quiere acceder a cualquier base de datos, tabla, o nombre de columna que sea una palabra reservada, debe delimitarla. Por ejemplo, y como hay una función USER() , el nombre de la tabla user en la base de datos mysql y la columna User en esa table se reserva, así que debe delimitarla:

```
SELECT "User" FROM mysql."user";
```

- NO_AUTO_CREATE_USER

  Previene que GRANT cree automáticamente nuevos usuarios si de otra forma se

haría, a no ser que se especifique un usuario. (Añadido en MySQL 5.0.2)

- NO_AUTO_VALUE_ON_ZERO afecta el tratamiento de las columnas AUTO_INCREMENT . Normalmente, genera el siguiente número de secuencia para la columna insertando NULL o 0 en ella. NO_AUTO_VALUE_ON_ZERO suprime este comportamiento para 0 de forma que sólo NULL genera el siguiente número de secuencia.

  Este modo puede ser útil si 0 se ha almacenado en una tabla con columnas AUTO_INCREMENT . (Esta no es una práctica recomendada, de todos modos.) Por ejemplo, si vuelca la tabla con **mysqldump** y posteriormente la recarga, normalmente MySQL genera un nuevo número de secuencia cuando encuentra los valores 0 , resultando en una tabla con distinto contenido que la que fue volcada. Activar NO_AUTO_VALUE_ON_ZERO antes de recargar el fichero con el volcado resuelve el problema. En MySQL 5.0, **mysqldump** incluye automáticamente en su salida un comando permitiendo NO_AUTO_VALUE_ON_ZERO.

- NO_BACKSLASH_ESCAPES

  Desactiva el uso del carácter de barra invertida ('\') como carácter de escape en cadenas de caracteres. Con este modo activado, la barra invertida se convierte en un carácter ordinario como cualquier otro. (Implementado en MySQL 5.0.1)

- NO_DIR_IN_CREATE

  Cuando crea una tabla, ignora todas las directivas INDEX DIRECTORY y DATA DIRECTORY. Este opción es útil en servidores de replicación esclavos.

- NO_ENGINE_SUBSTITUTION

  SI no está activado, cuando se intenta crear una tabla con un motor de almacenamiento no disponible, MySQL usa el motor por defecto y da un aviso. Si se intenta cambiar de motor a uno no disponible con en ALTER TABLE, el sistema da un aviso y el ALTER no tiene efecto. SI la opción está activada, se obtiene en ambos casos un error.

- NO_FIELD_OPTIONS

  No muestra opciones específicas para columnas de MySQL en la salida de SHOW CREATE TABLE. Este modo se usa con **mysqldump** en modo de portabilidad.

- NO_KEY_OPTIONS

  No muestra opciones específicas para índices de MySQL en la salida de SHOW CREATE TABLE. Este modo se usa con **mysqldump** en modo de portabilidad.

- NO_TABLE_OPTIONS

  No muestra opciones específicas para tablas (tales como ENGINE) en la salida de SHOW CREATE TABLE. Este modo se usa con **mysqldump** en modo de portabilidad.

- NO_UNSIGNED_SUBTRACTION

  En operaciones de resta, no marca el resultado como UNSIGNED si uno de los operandos no tiene signo. Note que esto hace que UNSIGNED BIGINT no sea 100% usable en todos los contextos.

- NO_ZERO_DATE

En modo estricto, no permite `'0000-00-00'` como fecha válida. Puede insertar fechas 0 con la opción `IGNORE` . Cuando no está en modo estricto, la fecha se acepta pero se genera una advertencia. (Añadido en MySQL 5.0.2)

- `NO_ZERO_IN_DATE`

  En modo estricto, no acepta fechas la parte del mes o día es 0. Se usa con la opción `IGNORE` , inserta una fecha `'0000-00-00'` para cualquiera de estas fechas. Cuando no está en modo estricto, la fecha se acepta pero se genera una advertencia. (Añadido en MySQL 5.0.2)

- `ONLY_FULL_GROUP_BY`

  No permite consultas que en la parte del `GROUP BY` se refieran a una columna que no aparezcan en la parte SELECT o HAVING.

- `PAD_CHAR_TO_FULL_LENGTH`

  Por defecto, los espacios al final de las cadenas de caracteres son eliminados en las columnas CHAR cuando se leen de los registros. Si esta opción está activada, esto no ocurre y se conservan. Este modo no se aplica para columnas VARCHAR que siempre conservan esos espacios finales. Este modo se añadió en 5.1.20.

```
mysql> CREATE TABLE t1 (c1 CHAR(10));
Query OK, 0 rows affected (0.37 sec)

mysql> INSERT INTO t1 (c1) VALUES('xy');
Query OK, 1 row affected (0.01 sec)

mysql> SET sql_mode = '';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT c1, CHAR_LENGTH(c1) FROM t1;
+------+-----------------+
| c1   | CHAR_LENGTH(c1) |
+------+-----------------+
| xy   |               2 |
+------+-----------------+
1 row in set (0.00 sec)

mysql> SET sql_mode = 'PAD_CHAR_TO_FULL_LENGTH';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT c1, CHAR_LENGTH(c1) FROM t1;
+------------+-----------------+
| c1         | CHAR_LENGTH(c1) |
+------------+-----------------+
| xy         |              10 |
+------------+-----------------+
1 row in set (0.00 sec)
```

- `PIPES_AS_CONCAT`

  Trata `||` como un concatenador de columnas de caracteres (lo mismo que `CONCAT()`) en lugar de como sinónimo de `OR`.

- `REAL_AS_FLOAT`

  Trata `REAL` como un sinónimo de `FLOAT` en lugar de sinónimo de `DOUBLE`.

- `STRICT_ALL_TABLES`

  Activa el modo estricto para todos los motores de almacenamiento. Rechaza los datos inválidos. Detalles adicionales a continuación. (Añadido en MySQL 5.0.2)

- STRICT_TRANS_TABLES

  Habilita el modo estricto para motores de almacenamiento transaccionales, y cuando sea posible también para los no transaccionales.

El modo estricto controla cómo MySQL trata los valores de entrada inválidos o no presentes. Un valor puede ser inválido por distintas razones. Por ejemplo, puede tener un tipo de datos incorrecto para la columna, o puede estar fuera de rango. Un valor no está presente cuando el registro a insertarse no tiene un valor para una columna que no tiene la cláusula DEFAULT explícita en su definición.

Para tablas transaccionales, se genera un error para valores inválidos o no presentes en un comando con los modos STRICT_ALL_TABLES o STRICT_TRANS_TABLES habilitados. El comando se aborta y deshace.

Para tablas no transaccionales, el comportamiento es el mismo para cualquier modo, si un valor incorrecto se encuentra en el primer registro a insertar o actualizar. El comando se aborta y la tabla continúa igual. Si el comando inserta o modifica varios registros y el valor incorrecto aparece en el segundo o posterior registro, el resultado depende de qué modo estricto esté habilitado:

- Para STRICT_ALL_TABLES, MySQL devuelve un error e ignora el resto de los registros. Sin embargo, en este caso, los primeros registros se insertan o actualizan. Esto significa que puede producirse una actualización parcial, que puede no ser lo que desea. Para evitarlo, es mejor usar comandos de un único registro ya que pueden abortarse sin cambiar la tabla.
- Para STRICT_TRANS_TABLES, MySQL convierte los valores inválidos en el valor válido más próximo para la columna e inserta el nuevo valor. Si un valor no está presente, MySQL inserta el valor por defecto implícito para el tipo de la columna. En ese caso, MySQL genera una advertencia en lugar de un error y continúa procesando el comando.

El modo estricto no permite fechas inválidas como '2004-04-31'. Esto sigue permitiendo fechas con partes con ceros, como 2004-04-00' o fechas ``cero''. Para no permitirlas tampoco, active los modos SQL NO_ZERO_IN_DATE y NO_ZERO_DATE además del modo estricto.

Si no usa el modo estricto (esto es, ni STRICT_TRANS_TABLES ni STRICT_ALL_TABLES están activados), MySQL inserta valores ajustados para valores inválidos o no presentes y produce advertencias. En modo estricto, puede producir este comportamiento usando INSERT IGNORE o UPDATE IGNORE.

Los siguientes modos especiales se proporcionan como abreviaciones de combinaciones de modos de la lista precedente. Todos están disponibles en MySQL 5.0 empezando en la versión 5.0.0, excepto para TRADITIONAL, que se implementó en MySQL 5.0.2.

- La descripción incluye todos los modos que están disponibles en la versión más reciente de MySQL. Para versiones anteriores, un modo de combinación no incluye todos los modos individuales que sólo están disponibles en las versiones más recientes.
- ANSI: Equivalente a REAL_AS_FLOAT, PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE. Antes de MySQL 5.0.3, ANSI también incluye ONLY_FULL_GROUP_BY. Consulte Sección 1.7.3, “Ejecutar MySQL en modo ANSI”.

- DB2: Equivalente a PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, NO_FIELD_OPTIONS.
- MAXDB: Equivalente a PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, NO_FIELD_OPTIONS, NO_AUTO_CREATE_USER.
- MSSQL: Equivalente a PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, NO_FIELD_OPTIONS.
- MYSQL323: Equivalente a NO_FIELD_OPTIONS, HIGH_NOT_PRECEDENCE.
- MYSQL40: Equivalente a NO_FIELD_OPTIONS, HIGH_NOT_PRECEDENCE.
- ORACLE: Equivalente a PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, NO_FIELD_OPTIONS, NO_AUTO_CREATE_USER.
- POSTGRESQL: Equivalente a PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, NO_FIELD_OPTIONS.
- TRADITIONAL: Equivalente a STRICT_TRANS_TABLES, STRICT_ALL_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE, ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER.

6. Diseño de bases de datos

A la hora de diseñar una base datos hay que tener en cuenta una serie de reglas generales:

- Las tablas no deben contener información redundante. Si hay datos repetidos, es que el diseño es incorrecto.
- Las tablas no deberían tener columnas como *pedido1*, *pedido2*, *pedido3*, ... Aunque tengamos 10 columnas para almacenar pedidos, llegará un día en que un cliente hará 11 pedidos y nos forzará a cambiar la estructura de la tabla.
- Se debería limitar al máximo el espacio necesario para almacenar los datos.
- Las consultas más frecuentes a la base de datos deberían ser lo más simples y eficientes posible. Hay que pensar siempre que podemos llegar a tener miles o millones de registros en una tabla a la hora de diseñar consultas.

También es necesario seguir una serie de normas a la hora de poner nombres a las tablas y las columnas:

- MySQL distingue entre mayúsculas y minúsculas en los nombres de bases de datos y tablas, pero no para los nombres de columnas. Por ello, es importante que los nombres de las bases de datos y las tablas tengan el mismo patrón de uso de mayúsculas y minúsculas.
- Los nombres las bases de datos y tablas tienen un máximo de 64 caracteres.
- Evitar el uso de caracteres especiales en los nombres (p.e. Üàû). Aunque MySQL permite usar cualquier carácter, diferentes sistemas operativos o versiones de Linux usan diferentes juegos de caracteres y ello puede dar lugar a problemas.
- Escoger nombres que describan con precisión los datos que representan. Por ejemplo `nombreAutor` es mejor que nombre.
- Utilizar un sistema coherente para poner nombres. Si usamos `nombreAutor` en una tabla, no usar `nombre_editorial` en otra.
- Otro detalle importante es decidir si usaremos plural o singular para los nombres de las tablas. Elegir uno u otro, pero ser siempre coherentes.

En general, no es fácil definir una estructura de tablas que represente la información que deseamos almacenar de manera eficiente tanto en velocidad de acceso como en espacio. Hay dos reglas generales a seguir:

- Empezar el diseño de las tablas usando una metodología clara. En particular, usar UML (Unified Modeling Language) para establecer que tablas usar y qué relaciones hay entre ellas.
- Empezar con un conjunto de datos pequeño pero representativo para realizar pruebas, antes de trabajar con datos reales.

En el resto del capítulo veremos una serie de técnicas para diseñar una base de datos de forma eficiente. Durante todo el capítulo utilizaremos la base de datos `biblioteca` que ya hemos visto en ejemplos anteriores.

6.1 Normalización

En la base de datos biblioteca guardaremos la información sobre una lista de libros. En principio esta es una parte de la información que queremos almacenar:

Título	Editorial	Año	Autor1	Autor2	Autor3
Linux	Grupo Anaya	2004	Miquel Pérez		
The Definitive ...	Grupo Planeta	2003	Miquel Pérez	David Puig	
Client/Server	Grupo Anaya	1997	Josep Martín	Dan López	Robert Vila
Web Application ...	Crisol	2000	Pau Costa	Tobías Álvarez	

Tabla 7: Primer intento con biblioteca

Para optimizar la manera en la cual estructuramos la información seguiremos el proceso denominado normalización. Este proceso está compuesto de tres formas normales.

Primera forma normal

La primera forma normal consiste en aplicar las siguientes reglas:

- Las columnas con contenido similar deben ser eliminadas ya que limitan los datos que podemos almacenar.
- Se debe crear una tabla para cada tipo de información.
- Cada registro se tiene que poder identificar con una clave primaria.

En nuestro ejemplo, La primera regla se aplica a las columnas AutorN.

La segunda regla parece no aplicable ya que nuestra base de datos guarda únicamente libros. Más tarde veremos que no no es así.

Para aplicar la tercera regla, lo que haremos es añadir una columna de tipo entero que utilizaremos para identificar cada libro.

Así pues, nuestra tabla quedaría de la siguiente manera:

tituloID	Título	Editorial	Año	Autor
1	Linux	Grupo Anaya	2004	Miquel Pérez
2	The Definitive ...	Grupo Planeta	2003	Miquel Pérez
3	The Definitive ...	Grupo Planeta	2003	David Puig
4	Client/Server	Grupo Anaya	1997	Josep Martín
5	Client/Server	Grupo Anaya	1997	Dan López
6	Client/Server	Grupo Anaya	1997	Robert Vila
7	Web Application ...	Crisol	2000	Pau Costa
8	Web Application ...	Crisol	2000	Tobías Álvarez

El problema de las columnas AutorN está resuelto ya que ahora podemos representar cualquier número de autores. Sin embargo, el precio que pagamos es que el resto de la información se repite por cada autor que añadimos.

Segunda forma normal

Consiste en aplicar dos reglas:

- Cada vez que el contenido de una columna se repita quiere decir que esa tabla debe ser dividida en otras tablas.
- Estas tablas deben ser enlazadas usando claves foráneas.

En nuestro ejemplo vemos que prácticamente todas las columnas repiten información. El problema viene de que cada título tiene varios autores, y eso hace que cada autor que añadimos replique toda la información sobre el libro. Así pues, lo que haremos será crear una tabla de autores de manera que nuestras tablas quedará así:

tituloID	**Título**	**Editorial**	**Año**
1	Linux	Grupo Anaya	2004
2	The Definitive ...	Grupo Planeta	2003
3	Client/Server	Grupo Anaya	1997
4	Web Application ...	Crisol	2000

autorID	**tituloID**	**Autor**
1	1	Miquel Pérez
2	2	Miquel Pérez
3	2	David Puig
4	3	Josep Martín
5	3	Dan López
6	3	Robert Vila
7	4	Pau Costa
8	4	Tobías Álvarez

Cada fila de esta tabla representa la aparición de un autor en un libro. La columna tituloID enlaza con los datos del libro correspondiente.

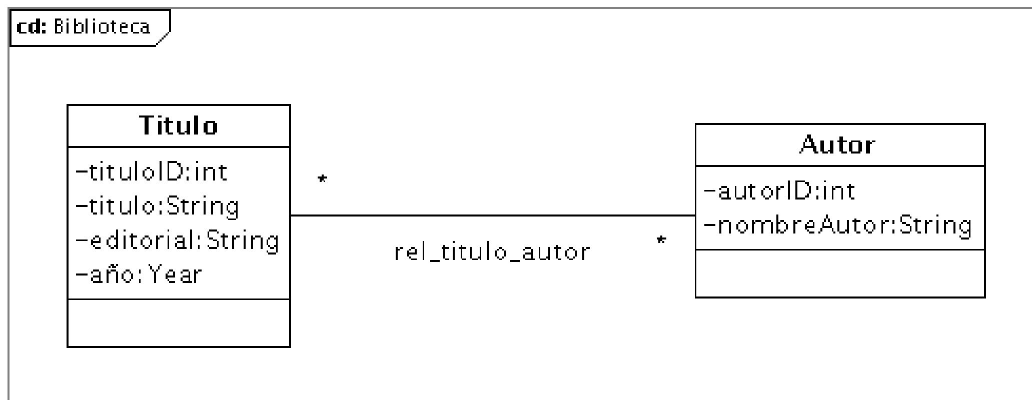
Sin embargo, siguen habiendo repeticiones en la tabla de autores, ya que los nombres se repiten por cada vez que un autor sale en un libro. La única solución es crear una tabla de autores en que los datos de cada autor solo se almacenen una vez, y crear una tabla más que almacene la relación entre libro y autor.

tituloID	Título	Editorial	Año
1	Linux	Grupo Anaya	2004
2	The Definitive ...	Grupo Planeta	2003
3	Client/Server	Grupo Anaya	1997
4	Web Application ...	Crisol	2000

autorID	Autor
1	Miquel Pérez
2	David Puig
3	Josep Martín
4	Dan López
5	Robert Vila
6	Pau Costa
7	Tobías Álvarez

tituloID	autorID
1	1
2	1
2	2
3	3
3	4
3	5
4	6
4	7

Esta última tabla representa la relación entre títulos de libros y autores. De hecho, es la representación de una relación n:m que en UML se representaría como:



Tercera forma normal

Consiste en aplicar esta regla:

- Las columnas que no están directamente relacionadas con la clave primaria deben ser eliminadas y su información traspasada a una tabla propia.

En nuestro caso, en la tabla `titulos` la columna `editorial` no forma parte de la clave primaria ya que varios títulos pueden compartir una misma editorial. Así pues, la tabla `titulos` y la nueva `editoriales` quedarían así:

tituloID	Título	editID	Año
1	Linux	1	2004
2	The Definitive ...	2	2003
3	Client/Server	1	1997
4	Web Application ...	3	2000

editID	nombre
1	Grupo Anaya
2	Grupo Planeta
3	Crisol

6.2 Integridad referencial

Las relaciones entre las clases se representan mediante claves primarias y foráneas.

Las claves primarias sirven para localizar, lo más rápido posible, un registro en una tabla. Además, una clave primaria identifica de manera univoca un registro en un tabla. Las claves primarias están formadas por una o más columnas de una tabla. Tiene que cumplir

las siguientes propiedades:

- Debe ser única.
- Debe ser lo más compacta posible. Primero, hay que mantener un índice para la clave primaria, con lo cual contra más compacta sea la clave primaria más eficiente (en velocidad y espacio) será el índice. Por ello se suelen usar enteros.

  Segundo, la clave primaria puede ser usada como foránea en otras tablas, por lo que la eficiencia no solo se refiere a la propia tabla donde está la clave primaria, sino a todas las tablas relacionadas a través de claves foráneas.

En la mayoría de sistemas de bases de datos la práctica habitual es usar enteros de 32 ó 64 bits generados automáticamente como una secuencia (AUTO_INCREMENT). De esta manera el programador no se tiene que preocupar de generar valores. En SQL:

```
CREATE TABLE editoriales
      (editID INT NOT NULL AUTO_INCREMENT,
      otras columnas ...,
      PRIMARY KEY (pubID))
```

Si creemos que el número de registros puede ser muy elevado, o que vamos a tener muchos INSERT y DELETE, podemos usar `BIGINT` en lugar de `INT`.

Por otro lado, las claves foráneas se usan para referenciar registros de otras tablas. Estas solo se usan cuando se tiene que consultar la base de datos. Por ejemplo, si queremos mostrar los títulos junto con el nombre de la editorial, ordenados alfabéticamente por el título, haremos los siguiente:

```
SELECT titulos.titulo, editoriales.nombreEditorial FROM titulos, editoriales
WHERE titulos.editID = editoriales.editID
ORDER BY titulo
```

Esta clave foránea se define de la siguiente manera:

```
CREATE TABLE titulos
      (columna1, columna2, ...,
      editID INT,
      FOREIGN KEY (editID) REFERENCES editoriales (editID)
```

Esto quiere decir que `titulos.editID` es una clave foránea que referencia la clave primaria `editoriales.editID`. Las reglas de integridad referencial que se aplicarán automáticamente serán:

- En `titulos`, no se puede insertar un registro con un `editID` que no exista en la tabla `editoriales`.
- No se puede cambiar el valor de `titulos.editID` si no existe en `editoriales`.
- No se puede borrar un registro de `editoriales` que esté referenciado por la tabla `titulos`.

Además, estas reglas afectan al orden en que se realizan las operaciones. Por ejemplo, si queremos añadir un nuevo título de una editorial que no existe, primero deberemos insertar el registro de la editorial, y luego el del título.

La sintaxis general de una clave foránea es:

```
FOREIGN KEY [nombre] (columna1) REFERENCES tabla2 (columna2)
      [ON DELETE {CASCADE | SET NULL | NO ACTION | RESTRICT}]
      [ON UPDATE {CASCADE | SET NULL | NO ACTION | RESTRICT}]
```

La parte opcional determina el comportamiento del sistema cuando se borra o actualiza un registro de `tabla2`. Hay cuatro posibilidades:

- RESRICT. Es el comportamiento por defecto. La instrucción DELETE causa un error y el registro no se borra.
- SET NULL. Se permite borrar el registro de `tabla2`. En `tabla1`, todos los registros que referencian al registro borrado tendrán NULL en `columna1`. Se asume que NULL es un valor permitido para `columna1`.
- CASCADE. El registro de `tabla2` es borrado, y además, también se borran todos los registros de `tabla1` que referencian al que estamos borrando en `tabla2`.
- NO ACTION. No se hace nada y se pierde la integridad referencial.

A pesar de haber explicado estas cuatro opciones para el caso de DELETE, en el caso de UPDATE el comportamiento es análogo.

Cuando añadimos una restricción a las reglas de integridad referencial, se deben de cumplir una serie de condiciones:

- Las columnas `tabla1.columna1` y `tabla2.columna2` deben de tener al menos un índice ordinario. El índice no se crea automáticamente al usar FOREIGN KEY y por lo tanto debe ser definido explícitamente.
- Los tipos de datos de `tabla1.columna1` y `tabla2.columna2` deben ser compatibles de manera que se puedan comparar sin requerir ninguna transformación. Los más fácil es que ambas sean INT o BIGINT. También las dos deben de ser del mismo signo (las dos SIGNED o las dos UNSIGNED).
- Si se ha usado la opción SET NULL, entonces la columna `tabla1.columna1` debe permitir valores NULL.
- Las restricciones deben cumplirse desde el inicio: si las tablas ya están creadas y se se está añadiendo una restricción con ALTER TABLE, si no se cumplen las condiciones con los datos existentes, se producirá un error.

Si al añadir una restricción obtenemos un error, para poder localizarlo hay que encontrar los registros que no cumplen la integridad referencial. Usando el caso de la biblioteca, supongamos que añadimos la restricción de que titulos.editID referencia editoriales.editID. Si obtenemos un error, la manera de ver qué registros lo han provocado será:

```
SELECT tituloID, editID FROM titulos
WHERE editID NOT IN (SELECT editID FROM editoriales)
```

Lo que hacemos es buscar todos los registros de titulos cuya columna `editID` contiene un valor que no existe en ninguno de los valores de la columna `editID` de `editoriales`.

Para borrar una restricción:

```
ALTER TABLE nombre_tabla DROP FOREIGN KEY foreign_key_id
```

Para conocer el nombre que MySQL asigna a las restricciones (`foreign_key_id`) podemos usar SHOW CREATE TABLE.

Si en algún momento tenemos problemas con la integridad referencial y queremos desactivarla, usaremos:

```
SET foreign_key_checks=0
```

7. Índices

Uno de los elementos de una base de datos que más afecta a su rendimiento son los índices. Por eso es importante saber qué son y qué particularidades tiene MySQL con respecto a ellos. Ese conocimiento nos permitirá optimizar el rendimiento de nuestras aplicaciones.

7.1 Conceptos básicos

Para entender la utilidad de los índices lo mejor es fijarse en como funciona una consulta. Imaginemos que tenemos una tabla `listin_telefonico` que contiene los nombre y teléfonos de los habitantes de un país. Eso puede suponer decenas de millones de registros, y recordemos que estos no se guardan de manera ordenada en una tabla. Ahora consideremos la siguiente consulta:

```
SELECT * FROM listin_telefonico WHERE apellido1='Martín'
```

Sino disponemos de ningún índice esto supone que MySQL deberá leer todos los registros de la tabla (varios millones) y seleccionar aquellos que cumplan la condición. Evidentemente la consulta será muy ineficiente ya que tiene un coste lineal u O(n).

Sin embargo, si pensamos en un listín de teléfonos real, lo que haríamos nosotros para buscar a alguien con el apellido 'Martín' sería ir directamente a la página donde están los apellidos que empiezan por 'M' y luego usar el orden alfabético para ir casi directamente a donde empiezan los 'Martín'.

Un índice, en el fondo, es una lista ordenada que nos permite, al igual que en el caso del listín, reducir la cantidad de registros a leer. El ejemplo anterior lo que haríamos es crear un índice para la columna 'apellido1' de esta manera:

```
ALTER TABLE listin_telefonico ADD INDEX (apellido1)
```

Con esto hacemos que MySQL cree una lista ordenada de apellidos de la tabla listin_telefonico. Con cada apellido diferente se guardará la lista de posiciones de los registros correspondientes (esto no es exactamente así ya que MySQL tiene estrategias para reducir el tamaño de los índices).

Los problemas comienzan cuando consideramos tablas que son modificadas frecuentemente. Eso quiere decir que hay que mantener el índice, que requiere un esfuerzo mantenerlo ordenado. Esto quiere decir que los índices requieren espacio de almacenamiento extra y tiempo de CPU extra. A cambio, pueden acelerar tremendamente determinadas consultas. Debido al coste extra, deberemos saber escoger qué columnas llevarán índices, intentando que sean las menos posibles.

Índices parciales

Los índices son un compromiso entre el espacio extra que requieren y la aceleración que proporcionan. Pero hay veces que el espacio en disco puede ser un factor crítico e indexar una columna que ocupa mucho puede significar índices muy grandes. Afortunadamente MySQL tiene un mecanismo que permite limitar la cantidad de información que se utilizada de una columna para formar el índice.

Imaginemos que en el ejemplo anterior el tamaño medio de apellido1 es 8 bytes y que

tenemos 100 millones de números de teléfono almacenados. Eso supone aproximadamente un índice de unos 8GB. Si consideramos que es demasiado, podemos reducirlo a la mitad creando el índice de la siguiente manera:

```
ALTER TABLE listin_telefonico ADD INDEX (apellido1(4))
```

De esta manera le decimos a MySQL que solo use los 4 primeros bytes de la columna apellido1 para conformar el índice. Cual es el problema ? Que los apellidos 'Marti', 'Martín', 'Martínez', ... quedan agrupados en un mismo grupo. Eso reduce la eficiencia del índice al hacer búsquedas ya que se eliminan menos registros.

Índices multicolumna

Como muchas bases de datos relacionales, MySQL permite tener índices compuestos de varias columnas.

```
ALTER TABLE listin_telefonico ADD INDEX (nombre, apellido1)
```

Esto puede ser útil si hacemos muchas consultas en las que aparecen las dos columnas en la cláusula WHERE, o si una columna no tiene suficiente variedad. Por supuesto, se pueden combinar los índices parciales con los multicolumna para limitar su tamaño:

```
ALTER TABLE listin_telefonico ADD INDEX (nombre(4), apellido1(4))
```

Esto acelerará mucho consultas del tipo:

```
SELECT * FROM listin_telefonico WHERE nombre='Ignacio' AND apellido1='Martin'
```

Esto es debido a que el número de registros que cumplirán la condición será muy pequeño, e incluso con el índice parcial, se leerán muy pocos registros de la tabla.

En este punto podríamos preguntarnos porqué no crear dos índices, uno para nombre y otro para apellido1. Podríamos hacerlo, pero MySQL nunca usaría los dos a la vez. Es más, MySQL solo usará un índice por tabla por consulta. Este dato es extremadamente importante ya que es el factor principal a la hora de considerar que índices se usarán en cada consulta.

Cuando hay más de un índice posible para acceder a una tabla, entonces MySQL decide cual es el mejor. Eso lo hace en base a unas estadísticas que se guardan internamente. Pero está decisión puede no ser óptima, especialmente cuando la distribución de los registros en el índice es muy heterogénea.

Ordenar con índices

Si se usa un sistema de bases de datos relacional, los resultados de una consulta pueden ordenarse, tanto en sentido ascendente como descendente. MySQL no ofrece ningún tipo de control sobre como gestiona la ordenación internamente. Y no hace falta que lo haga ya que es muy eficiente en cualquier tipo de ordenación.

Por ejemplo, hay sistemas de bases de datos que ejecutarán muy eficientemente esta consulta:

```
SELECT * FROM listin_telefonico WHERE apellido1='Martin'
  ORDER BY nombre DESC
```

y sin embargo esta otra consulta puede ser muy ineficiente:

```
SELECT * FROM listin_telefonico WHERE apellido1='Martin'
  ORDER BY nombre ASC
```

Esto es debido a que algunos sistemas guardan el índice ordenado de manera descendente y está optimizado para que los resultados sean retornados en ese orden. Y cuando se solicita el orden inverso hay que dar un segundo paso adicional para ordenar los resultados en sentido contrario.

Sin embargo, MySQL está diseñada para ser eficiente independientemente de cual sea el orden solicitado.

Los índices como restricciones

Los índices no solo se usan para localizar registros rápidamente. Un índice del tipo UNIQUE garantiza que cada valor solo aparece en un único registro. Por ejemplo, podríamos crear un índice de este tipo si quisiéramos tener un solo registro por cada número de teléfono:

```
ALTER TABLE listin_telefonico ADD UNIQUE (numero)
```

En este caso el índice tienen una función doble. Por una parte acelera las consultas del tipo:

```
SELECT * FROM listin_telefonico WHERE numero='972123456'
```

Y a la vez nos asegura que si intentamos insertar un registro con un número que ya existe el sistema no nos dejará.

El problema de esta doble función es que no se puede separar. Es decir, que si solo queremos utilizar la restricción, pero no queremos crear el índice porque no lo necesitamos para las consultas, no se puede hacer. Para MySQL la restricción y el índice UNIQUE son sinónimos. MySQL quiere modificar este comportamiento para separar el índice de la restricción. De hecho las tablas MyISAM ya lo hacen internamente pero no exponen la funcionalidad a nivel de SQL.

Índices clusterizados y secundarios

Las tablas MyISAM guardan los índices en ficheros separados que contienen la clave primaria (y las posibles secundarias). Además por cada valor de la clave se guarda la posición del registro en la tabla, ya que los registros se guardan sin ningún tipo de orden.

Los índices clusterizados funcionan al revés. El índice y los registros se guardan juntos y en el orden de la clave primaria. Este es el sistema que utiliza InnoDB. Cuando los datos son accedidos casi siempre por la clave primaria, las consultas pueden llegar a ser muy eficientes. Eso es debido a que solo se necesita realizar una lectura para obtener el registro ya que al leer la clave primaria también leemos el registro (en la cache). Mientras que las tablas MyISAM necesitan dos lecturas, una para la clave y otra para el registro. Es importante recalcar que esta eficiencia solo funciona cuando la consulta usa la clave primaria.

Sin embargo, los índices clusterizados pueden dar problemas serios de almacenamiento. Esto es debido a que los índices secundarios no apuntan directamente al registro (lo que no necesitaría mucho espacio, un entero de 64 bits) sino que guardan una copia de la clave primaria para acceder al registro. Si la clave primaria tiene un tamaño considerable,

los índices secundarios pueden llegara ocupar mucho espacio.

Otro problema menos común surge cuando se modifica la clave primaria de un registro. Eso implica realizar las siguiente operaciones:

- Actualizar el registro
- Determinar la nueva clave primaria
- Mover el registro a su nueva posición
- Actualizar todos los índices secundarios

Todas estas operaciones pueden llegar a ser muy costosas por lo que es muy importante escoger bien las claves primarias y asegurarse de que sea casi imposible que cambien. Por ejemplo, usar NIFs, número de la seguridad social, ... etc.

UNIQUE vs PRIMARY KEY

En las tablas MyISAM prácticamente no hay diferencia entre las dos opciones. La única diferencia es que los índices de clave primaria no pueden contener valores NULL. De esa manera, un índice PRIMARY es simplemente un índice NOT NULL UNIQUE.

Por el contrario, las tablas InnoDB siempre llevan una clave primaria. No hace falta especificar una explícitamente, ya que si no se hace MySQL crea una oculta automáticamente. En ambos casos las claves primarias son enteros AUTO_INCREMENT. Si se añade una clave primaria después de crear una tabla, entonces MySQL destruirá la oculta que creó por defecto.

Indexar NULLs

Es importante recalcar que MySQL utiliza una lógica de tres estados cuando realiza operaciones lógicas. Cuando se hace una comparación hay tres resultados posibles: cierto si los valores son equivalentes, falso si los valores no son equivalentes, y NULL si alguno de los valores a comparar es NULL.

Como se tratan los valores NULL en los índices ? Los valores NULL se pueden usar en índices no UNIQUE. Debido que al comparar dos valores NULL no se obtiene cierto, sino NULL, este valor no se puede utilizar en índices UNIQUE.

Sin embargo, el valor NULL puede aparecer una sola vez en un índice de clave primaria. Este comportamiento es estándar SQL. Es una de las pocas cosas que diferencian un índice UNIQUE de uno de clave primaria. Y para terminar, el usar NULL en un índice de clave primaria no afecta al rendimiento.

Índices en tablas InnoDB

Los índices son más importantes en tablas InnoDB que en las MyISAM. En las primeras, los índices no solo son usados para buscar registros sino también para poder realizar bloqueos a nivel de registro. Esto afecta, entre otros comandos, a SELECT ... LOCK IN SHARE MODE, SELECT ... FOR UPDATE, e INSERT, UPDATE y DELETE. El etiquetado de qué registros están bloqueados no se almacena en la tabla sino en los índices. Pero para ello debe haber un índice disponible.

Limitaciones

- MySQL no puede usar índices cuando se usan operadores de desigualdad (WHERE columna != ...)

- MySQL no puede usar índices cuando se usan comparaciones que contienen evaluaciones de funciones (WHERE DAY(columna) = ...)
- En operaciones JOIN los índices solo se pueden usar cuando la clave primaria y las foráneas son del mismo tipo de datos.
- Si se usan los operadores LIKE y REGEXP en las comparaciones, los índices solo se usaran cuando no haya ningún comodín al principio del patrón de búsqueda. Por ejemplo, con LIKE 'abc%' sí que se usarán índices, mientras que con LIKE '%abc' no.
- Los índices son ineficientes si la columna correspondiente contiene muchos valores repetidos. Por ejemplo, es completamente inútil indexar una columna que guarda 0/1 o SI/NO.

Índices Full-text

Un índice ordinario definido en una columna de tipo texto solo sirve para buscar caracteres que se encuentran al principio de cada campo. La consecuencia es que si tenemos textos largos con muchas palabras y queremos buscar palabras usando LIKE '%word%' la consulta es muy poco eficiente ya que el índice no ayuda a encontrar las palabras que buscamos.

En estos casos es útil usar un índice Full-Text. Con este tipo de índice, MySQL crea una lista de todas las palabras que aparecen en el texto. El índice puede ser creado al crear la tabla, o posteriormente:

```
ALTER TABLE nombre_tabla ADD FULLTEXT(columna1, columna2)
```

En una consulta, se pueden buscar registros que contengan una o más palabras:

```
SELECT * FROM nombre_tabla
WHERE MATCH(columna1, columna2) AGAINST ('palabra1', palabra2', 'palabra3')
```

La desventaja de este tipo de índices es que solo funciona para tablas del tipo MyISAM.

7.2 Estructuras de datos

En esta sección veremos las estructuras de datos más usadas para almacenar índices. No son específicas de MySQL por lo que se pueden encontrar en otros sistemas de bases de datos.

Índices B-Tree

Básicamente son árboles balanceados, y es la estructura de datos más común para almacenar índices. Casi todos los sistemas de bases de datos ofrecen esta estructura de datos para almacenar los índices. Ello es debido a la buena combinación de flexibilidad, tamaño y eficiencia general.

Los nodos del árbol se guarda ordenados por la clave que se almacena. la clave está en que el árbol se mantiene balanceado después de añadir y borrar nodos. Ofrecen un tiempo de acceso O(log n) para búsquedas de un solo registro. A diferencia de los árboles binarios, cada nodo guarda un número indeterminado de valores. .

También ofrecen muy buenos resultados en consultas de tipo `range`. En la Figura 10

podemos ver un ejemplo de árbol balanceado.

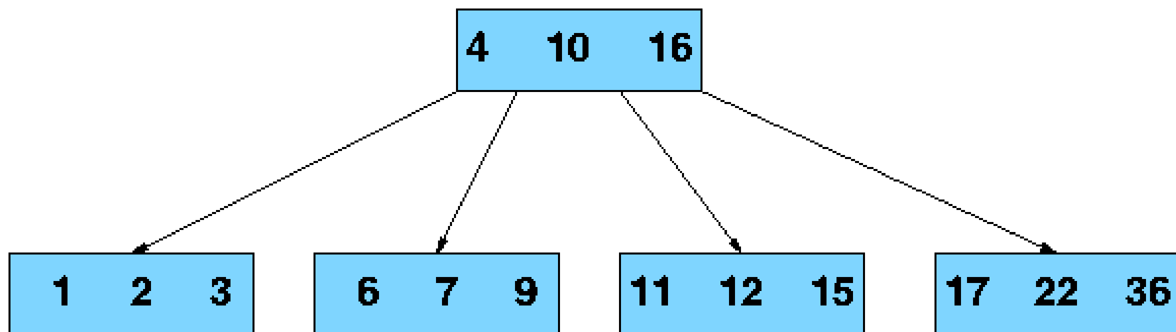


Figura 10: Ejemplo de árbol balanceado

Si queremos buscar un rango de valores, solo hay que localizar uno de los extremos, y entonces recorrer el árbol para obtener todos los valores ordenados.

Índices hash

La segunda estructura más usada son los índices hash. Como su nombre indica, esta estructura no es de tipo árbol sino que es una tabla hash. En lugar de hacer una búsqueda en el árbol, las tablas hash localizan los elementos aplicando una función hash para determinar en que porción de la tabla se encuentra el registro.

Una función hash muy común y muy usada es MD5. Esta función puede usarse directamente en MySQL:

```
mysql> SELECT MD5('Smith');
+----------------------------------+
| MD5('Smith')                     |
+----------------------------------+
| e95f770ac4fb91ac2e4873e4b2dfc0e6 |
+----------------------------------+
```

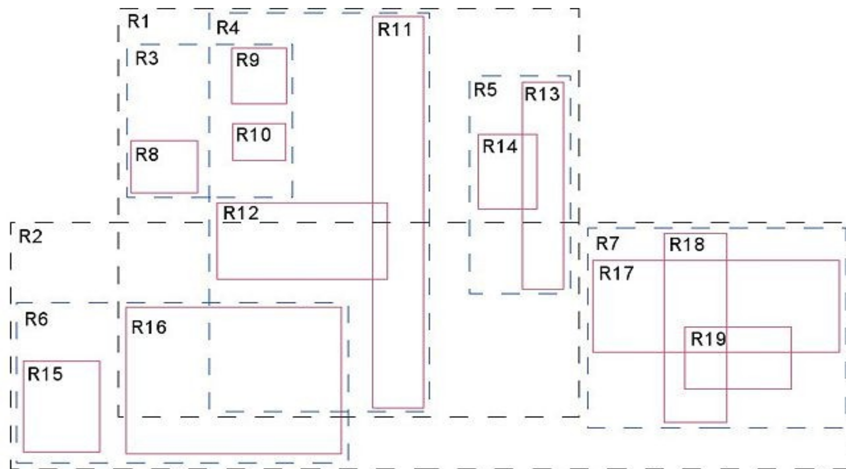
El problema es que retorna enteros de 128 bits, un rango demasiado grande para los sistemas de almacenamiento actuales.

Una solución común para reducir el rango de la función hash es crear un número grande de buckets (espacio donde se almacenan los datos correspondientes a un mismo resultado de la función hash), y dividir el resultado de la función por este número. En este caso, varios resultados de la función hash irán a parar al mismo bucket.

Este sistema promete un acceso O(1), pero la eficiencia real depende de lo buena que sea la función hash para distribuir los elementos de forma homogénea entre los buckets. A pesar de su mejor eficiencia en búsquedas de un registro, son peores cuando se busca un rango ordenado de valores ya que los registros no están almacenados en ningún orden.

Índices R-Tree

Son índices para guardar danos con N dimensiones. Fueron introducidos para poder tener índices en datos de tipo geográfico. Además de como índice, se usan para resolver consultas de tipo geométrico. Podemos ver un ejemplo de estructura de R-Tree en la Figura 11.



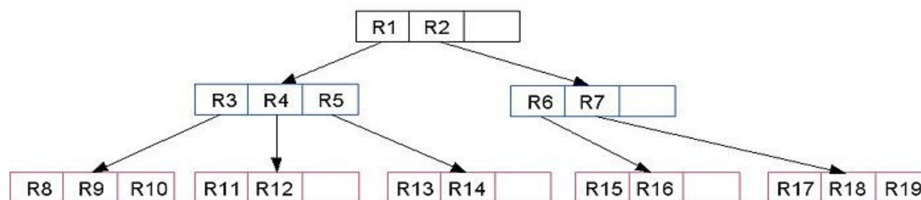


Figura 11: Ejemplo de R-Tree

7.3 Índices y tipos de tablas

7.3.1 Tablas MyISAM

Usa por defecto B-Trees y para los datos espaciales usa R-Trees. En el caso de los B-Trees, además de una buena implementación, se ofrecen dos características importantes:

- Compresión de prefijo. Para índices sobre cadenas de caracteres, MySQL puede reconocer prefijos comunes y comprimirlos para que los índices ocupen menos. El ejemplo típico es un índice sobre direcciones web donde todas comienzan por `http://` y `http://www.`
- Claves empaquetadas. Es un mecanismo parecido al anterior pero para enteros. En los índices los enteros se almacenan con el byte más significativo primero. Este suele variar muy poco conforme incrementamos los valores, de manera que se pueden agrupar rangos en los que no se almacenará la clave entera. Para activarlos añadiremos `PACKED_KEY = 1` al crear la tabla.

Escritura de clave retardada

Una de las optimizaciones de este tipo de tablas es la capacidad para retrasar la escritura del índice a disco. Normalmente esta escritura se hace inmediatamente después de que haya una modificación, pero este comportamiento puede cambiar si añadimos `DELAY_KEY_WRITE` al crear la tabla. En este caso, el comportamiento exacto dependerá

de la variable `delay_key_write`, que puede tener tres valores:

- ON: la escritura a disco se produce cuando la tabla se cierra, y eso ocurre bajo determinadas circunstancias que dependen del estado de la cache, con lo cual no es posible controlarlo.
- OFF: la opción `DELAY_KEY_WRITE` se ignora.
- ALL: activa el retraso en la escritura para todas las tablas MyISAM independientemente del valor `DELAY_KEY_WRITE` de cada una.

Obviamente, el problema de usar este mecanismo es que los índices pueden estar mal sincronizados con el contenido de sus respectivas tablas, y eso es especialmente peligroso si se produce una caida del sistema justo en ese momento.

7.3.2 Tablas HEAP

Se puede escoger entre usar B-Trees o hash. Este último es el valor por defecto, y para cambiarlo hay que hacer lo siguiente:

```
mysql> create table heap_test (
    -> name varchar(50) not null,
    -> index using btree (name)
    -> ) type = HEAP;
```

Al combinar el almacenamiento en memoria con los B-Trees, la eficiencia de estas tablas es muy alta, especialmente si hacemos consultas de rangos.

7.3.3 Tablas InnoDB

Solo implementan índices B-Tree, y no tiene las optimizaciones de las tablas MyISAM. Además, InnoDB requiere una clave primaria por tabla (con su correspondiente índice). Si no se especifica una, MySQL creará una por defecto con un entero de 64 bits.

InnoDB almacena el índice en el espacio de tablas común, y utiliza índices clusterizados. Esto hace que buscar un registro en este tipo de tablas sea muy rápido.

7.4 Índices Full-text

Es un tipo de índice especial para columnas de tipo texto que permite buscar palabras muy rápidamente. Solo están disponibles para tablas MyISAM. Se pueden construir para una o más columnas de tipo texto.

Se construye creando un árbol B-Tree con dos componentes, uno VARCHAR y otro FLOAT. El primero contiene una palabra indexada, y el segundo su peso en el registro. Eso significa que habrá un elemento por cada palabra en el campo del índice, lo que significa que el tamaño del índice puede ser muy grande. Así que este índice es un compromiso entre espacio y velocidad.

8. Optimización de consultas

Una de las tareas principales a la hora de hacer nuestras aplicaciones eficientes es la optimización de las consultas a la base de datos. En esta sección veremos como procesa las consultas MySQL y qué aspectos hay que tener en cuenta para acelerar consultas lentas.

8.1 Gestión interna de las consultas

MySQL realiza la ejecución de las consultas en varias etapas.

8.1.1 Cache de consultas

Este mecanismo se puede activar y desactivar a voluntad en el fichero de configuración del servidor a través de la variable:

```
query_cache_type = 1
```

Cuando está activada, MySQL intenta localizar la consulta en la cache antes de analizarla. Para ello dispone de una tabla hash interna en donde las claves son el texto exacto de las consultas. Eso quiere decir que es sensible a cualquier variación en la manera de escribir las consultas. Por ejemplo, estas dos consultas son diferentes desde el punto de vista de la cache:

```
SELECT * FROM nombre_tabla
select * FROM nombre_tabla
```

El simple hecho de usar minúsculas en lugar de mayúsculas hará que la cache identifique las dos consultas como diferentes. Y lo mismo pasa si se usa un número diferente de espacios en blanco.

Es importante recalcar que MySQL solo guarda consultas del tipo SELECT ya que son las únicas que tiene sentido guardar.

8.1.2 Análisis sintáctico y optimización

Cuando una consulta llega a MySQL y no se encuentra en la cache, lo primero que se hace es analizarla sintácticamente y extraer información de su estructura:

- Tipo: SELECT, INSERT, UPDATE o DELETE, o algún tipo de comando administrativo como GRANT o SET.
- Qué tablas aparecen, y qué alias se usan.
- Como es la cláusula WHERE.
- Qué otros atributos aparecen.

Una vez que la consulta se descompone en partes más básicas, empieza la parte complicada donde se decide como se ejecutará. Aquí es donde empieza el trabajo del optimizador, cuya tarea es ejecutar la consulta lo más eficientemente posible. La mayoría de las veces este trabajo consiste en reducir al máximo el número de registros a

consultar. Esto es debido a que la mayoría de veces el tiempo de entrada-salida a disco es el factor más importante en el tiempo total de ejecución.

MySQL tiene una serie de reglas y heurísticas que han ido evolucionando con las diferentes versiones. Y por su propia naturaleza, funcionan bien la mayoría de las veces, mientras que otras producen resultados subóptimos.

El optimizador trata de responder una serie de cuestiones:

- Hay algún índice que potencialmente pueda usarse para reducir el número de registros a consultar ?
- Cual es el mejor índice ? SI hay varias tablas en la consulta, cual es el mejor para cada tabla ?
- Qué tabla depende de otras en un JOIN ?
- Cual es el orden de tablas óptimo en un JOIN ?

Estas preguntas tienen que ser respondidas en muy poco tiempo y eso implica que no tiene tiempo para probar todas las opciones.

El trabajo principal se centra en los índices y en el orden de los JOIN.

8.2 EXPLAIN SELECT

La herramienta fundamental para analizar una consulta y determinar si el optimizador hace un buen trabajo, o como mejorarla, tenemos el la instrucción EXPLAIN que colocaremos delante de cada SELECT que queramos analizar.

Por ejemplo, consideremos la tabla `titulos` de la base de datos `biblioteca`:

```
mysql> DESCRIBE titulos;
+------------+--------------+------+-----+-------------------+-----------------------------+
| Field      | Type         | Null | Key | Default           | Extra                       |
+------------+--------------+------+-----+-------------------+-----------------------------+
| tituloID   | int(11)      | NO   | PRI | NULL              | auto_increment              |
| titulo     | varchar(100) | NO   | MUL |                   |                             |
| subtitulo  | varchar(100) | YES  |     | NULL              |                             |
| edition    | tinyint(4)   | YES  |     | NULL              |                             |
| editID     | int(11)      | YES  |     | NULL              |                             |
| catID      | int(11)      | YES  | MUL | NULL              |                             |
| idiomaID   | int(11)      | YES  | MUL | NULL              |                             |
| año        | int(11)      | YES  |     | NULL              |                             |
| isbn       | varchar(20)  | YES  |     | NULL              |                             |
| comentario | varchar(255) | YES  |     | NULL              |                             |
| ts         | timestamp    | NO   |     | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
| autores    | varchar(255) | YES  |     | NULL              |                             |
+------------+--------------+------+-----+-------------------+-----------------------------+
12 rows in set (0.00 sec)
```

Si queremos consultar un título por su identificador haremos esto:

```
mysql> SELECT titulo, subtitulo FROM titulos WHERE tituloID=11;
+-----------------------------+-----------+
| titulo                      | subtitulo |
+-----------------------------+-----------+
| A Guide to the SQL Standard | NULL      |
+-----------------------------+-----------+
1 row in set (0.00 sec)
```

En este caso es obvio que solo puede haber un resultado ya que estamos haciendo la consulta para un valor de la clave primaria. La estrategia para buscar el registro es muy simple: buscar en el índice el valor dado y acceder al registro directamente. Para

comprobarlo, usemos EXPLAIN:

```
mysql> EXPLAIN SELECT titulo, subtitulo FROM titulos WHERE tituloID=11 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: titulos
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
1 row in set (0.02 sec)
```

Como esperábamos, solo hay un registro que cumple la condición (`rows: 1`). SIn embargo, luego veremos que no siempre este valor es exacto. Pero EXPLAIN nos da mucha más información que el número de registros examinados. Veamos cual es:

- `id`: el identificador del SELECT. En la tabla que produce EXPLAIN solo hay un registro por cada tabla que aparece en la consulta.
- `select_type`: cual es el papel de esta tabla en toda la consulta ? Los valores posibles son `SIMPLE`, `PRIMARY`, `UNION`, `DEPENDENT UNION`, `SUBSELECT`, y `DERIVED`. Conforme veamos consultas más complejas veremos que quieren decir.
- `table`: el nombre de la tabla correspondiente.
- `type`: qué tipo de JOIN se va a usar ? En el ejemplo vemos `const` ya que hay un valor constante en la consulta. Otros valores posibles son `system`, `eq_ref`, `ref`, `range`, `index`, y `ALL`. Veremos que quieren decir en la sección sobre JOIN.
- `possible_keys`: lista de todos los índices que MySQL puede utilizar para buscar registros en la tabla correspondiente.
- `key`: el índice que MySQL escoge como el mejor y que debe estar listado en `possible_keys`. Hay expeciones.
- `ref`: indica la columna o columnas que se compararan con los valores del índice escogido en `key`.
- `rows`: el número de registros que MySQL piensa que ha de leer para satisfacer la consulta. Si se hacen muchas inserciones y borrados de registros, es conveniente ejecutar ANALIZE TABLES frecuentemente para mantener actualizadas las estadísticas que MySQL usa para esta estimación.
- `extra`: información adicional que MySQL ha usado para analizar la consulta.

Compliquemos ligeramente el ejemplo y busquemos un rango de valores:

```
mysql> EXPLAIN SELECT titulo FROM titulos WHERE tituloID BETWEEN 1 AND 11 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: titulos
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 7
```

```
        Extra: Using where
1 row in set (0.05 sec)
```

En este caso `type` ha cambiado de `const` a `range` para indicar que estamos buscando un rango de registros, no uno solo. También ha aparecido `Using where` en el campo `Extra`. Eso indica que MySQL advierte de que se aplican las restricciones de la cláusula WHERE a lo hora de seleccionar registros.

Ahora vamos a ver si hacemos una consulta por una columna sin índices en una tabla con 1 millón de registros:

```
mysql> select count(*) from titulos_big where titulo >= "The";
+----------+
| count(*) |
+----------+
|   285344 |
+----------+
1 row in set (4.62 sec)
```

Este es el tiempo de ejecución la primera vez que accedemos a la tabla después de inicializar el servidor. Si volvemos a ejecutar la misma consulta inmediatamente después obtenemos esto:

```
mysql> select count(*) from titulos_big where titulo >= "The";
+----------+
| count(*) |
+----------+
|   285344 |
+----------+
1 row in set (1.17 sec)
```

Como puede observarse el tiempo es mucho menor. Esto es debido a que la mayoría de los datos que hay que consultar están en la cache en la segunda ejecución. Este es un detalle importantísimo a la hora de realizar tests de velocidad. Siempre que hagamos una prueba debemos asegurarnos que la cache está vacía, ya que si no los resultados que obtengamos no tendrán en cuenta de forma correcta los accesos a disco necesarios. Hay que pensar que en un entorno de producción con muchas consultas concurrentes, es probable que la parte de la cache disponible para nuestra consulta sea muy pequeña, por lo que siempre tendremos que considerar el caso peor en que nada está en la cache antes de la consulta. Esta consideración es más importante si la tabla cabe entera en memoria ya que puede provocar que la consulta ni siquiera acceda a disco.

El comando EXPLAIN nos confirma que la consulta no ha usado ningún índice:

```
mysql> EXPLAIN select count(*) from titulos_big where titulo >= "The" \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: titulos_big
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1004142
        Extra: Using where
1 row in set (0.00 sec)
```

Añadamos un índice por la columna titulo para intentar mejorar la velocidad de la consulta:

```
mysql> ALTER TABLE titulos_big ADD INDEX `titulo_index`(`titulo`);
Query OK, 1000000 rows affected (11 min 19.12 sec)
Records: 1000000  Duplicates: 0  Warnings: 0
```

Del resultado de esta acción hay que destacar el tiempo de ejecución, que es muy elevado. Este tiempo es para una tabla de 1 millón de registro, pero hay que pensar que en bases de datos reales podemos tener 100 millones fácilmente. Eso quiere decir que si nos damos cuenta tarde de la necesidad de un índice, tendremos que parar el servidor un tiempo considerable para crear el índice.

El tiempo de ejecución con el índice disminuye:

```
mysql> select count(*) from titulos_big where titulo >= "The";
+----------+
| count(*) |
+----------+
|   285344 |
+----------+
1 row in set (2.51 sec)
```

Y en una segunda ejecución se nota más:

```
mysql> select count(*) from titulos_big where titulo >= "The";
+----------+
| count(*) |
+----------+
|   285344 |
+----------+
1 row in set (0.37 sec)
```

Ahora la información que da EXPLAIN es mejor:

```
mysql> EXPLAIN select count(*) from titulos_big where titulo >= "The" \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: titulos_big
         type: range
possible_keys: titulo_index
          key: titulo_index
      key_len: 102
          ref: NULL
         rows: 502071
        Extra: Using where; Using index
1 row in set (0.01 sec)
```

Ahora se está usando el índice `titulo_index` y la consulta es tipo `range`. El campo Extra nos indica con `Using index` que la consulta únicamente accede al índice y no lee ningún registro. Esto es debido a que lo único que hacemos es contar registros y no hacemos nada más con los datos.

Otro detalle importante es ver la discrepancia entre la estimación de registros que da `EXPLAIN` y la cantidad real de registros que realmente se acceden en la consulta (en esta caso casi la mitad).

Ahora veamos un ejemplo con subselects:

```
mysql> SELECT titulo FROM titulos_big WHERE tituloID IN (SELECT MAX(tituloID)
FROM titulos_big);
+----------------------------------------+
```

```
| titulo                                |
+---------------------------------------+
| A Guide to the SQL Standard 722530606 |
+---------------------------------------+
1 row in set (1.38 sec)

mysql> SELECT titulo FROM titulos_big WHERE tituloID = (SELECT MAX(tituloID)
FROM titulos_big);
+---------------------------------------+
| titulo                                |
+---------------------------------------+
| A Guide to the SQL Standard 722530606 |
+---------------------------------------+
1 row in set (0.00 sec)
```

Estas dos consultas son semánticamente iguales, y la una diferencia es que en la primera se usa el operador IN para seleccionar el registro que buscamos, mientras que en la segunda se usa el operador =. Sin embargo el tiempo de ejecución es radicalmente diferente. Esto es una muestra de que el optimizador de MySQL no siempre actúa como podríamos esperar. Veamos como usando EXPLAIN podemos ver qué ha ocurrido:

```
mysql> EXPLAIN SELECT titulo FROM titulos_big WHERE tituloID IN (SELECT
MAX(tituloID) FROM titulos_big)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: titulos_big
         type: index
possible_keys: NULL
          key: titulo_index
      key_len: 102
          ref: NULL
         rows: 1004142
        Extra: Using where; Using index
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: Select tables optimized away
2 rows in set (0.00 sec)
```

El campo `type` de la primera fila es `index` lo que quiere decir que MySQL ha recorrido todo el índice de tabla para ver que valores estaban `IN`. Fijarse que el campo `rows` tiene un valor de 1 millón aproximadamente. En la segunda tabla todo es NULL ya que según el campo `Extra: Select tables optimized away` quiere decir que el valor del select se ha obtenido de manera optima internamente.

Veamos que nos dice EXPLAIN de la segunda versión:

```
mysql> EXPLAIN SELECT titulo FROM titulos_big WHERE tituloID = (SELECT
MAX(tituloID) FROM titulos_big)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: titulos_big
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
```

```
         rows: 1
        Extra:
*************************** 2. row ***************************
           id: 2
  select_type: SUBQUERY
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: Select tables optimized away
2 rows in set (0.00 sec)
```

Ahora en la primera fila el campo `type` es `const`, lo que quiere decir que se ha accedido a un valor directamente a través del índice, por lo que el campo `rows` vale 1. Por eso es mucho más rápido.

Sin embargo, las dos consultas representan lo mismo, pero el optimizador de MySQL no ha sabido resolverlo en uno de los casos. Para eso hemos utilizado EXPLAIN y hemos visto cual era la razón.

Ahora veamos un ejemplo donde la consulta depende de los valores de dos columnas con índice. Por ejemplo:

```
mysql> SELECT COUNT(*) FROM titulos_big WHERE tituloID > 500000 AND editID <
5;
+----------+
| COUNT(*) |
+----------+
|   276810 |
+----------+
1 row in set (0.26 sec)

mysql> EXPLAIN SELECT COUNT(*) FROM titulos_big WHERE tituloID > 500000 AND
editID < 5 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: titulos_big
         type: range
possible_keys: PRIMARY,publIdIndex
          key: publIdIndex
      key_len: 5
          ref: NULL
         rows: 245798
        Extra: Using where; Using index
1 row in set (0.00 sec)
```

En este caso MySQL tiene dos posibles índices para realizar el SELECT. En general se escogerá el que más reduzca el número de registros a examinar. En este caso, como `editID < 5` representa menos de la mitad de los registros y `tituloID > 500000` representa justo la mitad, MySQL escoge el índice correspondiente a `editID`. Sin embargo, si cambiamos el valor que se compara con `editID`:

```
mysql> EXPLAIN SELECT COUNT(*) FROM titulos_big WHERE tituloID > 500000 AND
editID < 15 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: titulos_big
         type: range
possible_keys: PRIMARY,publIdIndex
          key: PRIMARY
```

```
      key_len: 4
          ref: NULL
         rows: 490916
        Extra: Using where
1 row in set (0.00 sec)
```

En este caso `editID < 15` representa más de la mitad de los registros, y MySQL cambia a `PRIMARY`.

En general, es importante recordar que MySQL toma sus decisiones en lo que es capaz de adivinar o estimar de los datos. Esto se hace en base a unas estadísticas que cambian con el tipo de tabla, por lo que los resultados obtenidos podrían ser diferentes con otro tipo de tablas (hemos usado InnoDB). Además, si ejecutamos ANALIZE TABLE actualizaremos las estadísticas, y por tanto afectaremos las decisiones del optimizador.

JOIN

Las cosas se complican cuando empezamos a usar más de una tabla en una consulta.

Por ejemplo, si queremos obtener una lista no ordenada de todos los libros con todos sus autores, utilizaremos la siguiente consulta:

```
mysql> EXPLAIN SELECT titulo, nombreAutor FROM titulos_big,
rel_titulo_autor_big, autores WHERE rel_titulo_autor_big.IDautor =
autores.IDautor AND rel_titulo_autor_big.tituloID = titulos_big.tituloID \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: autores
         type: index
possible_keys: PRIMARY
          key: nombreAutor
          ref: NULL
         rows: 26
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: rel_titulo_autor_big
         type: ref
possible_keys: PRIMARY,IDautor
          key: IDautor
          ref: biblioteca.autores.IDautor
         rows: 128309
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: titulos_big
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
          ref: biblioteca.rel_titulo_autor_big.tituloID
         rows: 1
        Extra:
3 rows in set (0.00 sec)
```

La primera fila se refiere a `autores`, por lo que SELECT comienza por acceder a esta tabla. Como `type` es `index` quiere decir que accede a todos los registros de la tabla a través del índice, sin leerlos. Sin embargo aquí podemos ver una excepción que hace que la clave escogida no esté entre las listadas en `possible_keys`. Esto ocurre cuando las columnas a seleccionar de la tabla están contenidas íntegramente en un índice, y de esa manera es más eficiente que acceder por el índice primario y leer los registros.

En un segundo paso se accede a `rel_titulo_autor`. El tipo de JOIN es `ref` lo que quiere decir es que para cada registro anterior (`autores`) se accederá a varios de `rel_titulo_autores` para los cuales coincida la clave `IDautor` (campo `key`) . Además nos dice que la clave se ha comparado con `autores.IDautor` de cada registro previo. En este momento, el campo rows da una media de los registros que se tienen que consultar de `rel_titulo_autores` por cada registro de `autores` del paso anterior.

En el último paso se ha accedido a la tabla `titulos`, solo se ha accedido a un registro por cada grupo de registros del paso anterior (`type` es `eq_ref`) , se ha usado la clave primaria para seleccionar el registro (`key` es `PRIMARY`) , y esta se ha comparado con la columna `rel_titulo_autor.tituloID` (campo `ref`) de cada registro el paso anterior.

Aquí tenemos otro ejemplo:

```
mysql> EXPLAIN SELECT titulo, nombreAutor FROM titulos, rel_titulo_autor,
autores
WHERE titulos.editID=2 AND titulos.tituloID = rel_titulo_autor.tituloID
AND autores.IDautor = rel_titulo_autor.IDautor
ORDER BY titulo \G
*************************** 1. row ***************************
  select_type: SIMPLE
        table: titulos_big
         type: ref
possible_keys: PRIMARY,publIdIndex
          key: publIdIndex
          ref: const
         rows: 42458
        Extra: Using where; Using filesort
*************************** 2. row ***************************
  select_type: SIMPLE
        table: rel_titulo_autor_big
         type: ref
possible_keys: PRIMARY,IDautor
          key: PRIMARY
          ref: biblioteca.titulos_big.tituloID
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
  select_type: SIMPLE
        table: autores
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
          ref: biblioteca.rel_titulo_autor_big.IDautor
         rows: 1
        Extra:
```

Este resultado nos dice que la consulta está bien optimizada ya que en cada paso se usa un índice para acceder a los registros (la columna `key` no tiene ningún NULL). La primera fila contiene un `Using filesort`, y eso quiere decir que se necesita un paso extra,es decir, primero se acede a la lista de claves primarias que pasan el test del WHERE (`Using where`), se ordenan, y después se accede a los registros. Pero como va precedido del `Using where` significa que este paso se aplica a pocos registros. En este primer paso se seleccionan los títulos que pertenecen a la editorial correspondiente (editID=2)

Las dos filas restantes tienen lo que esperamos. Acceso a través de clave primaria a muy pocos elementos ya que por cada título seleccionado accederemos a muy pocos autores.

Ahora podríamos hacer la prueba de eliminar el índice publIndex de la tabla titulos_big

para ver como reacciona MySQL.

```
mysql> EXPLAIN SELECT titulo, nombreAutor FROM titulos_big,
rel_titulo_autor_big, autores WHERE titulos_big.editID=2   AND
titulos_big.tituloID = rel_titulo_autor_big.tituloID   AND autores.IDautor =
rel_titulo_autor_big.IDautor ORDER BY titulo \G
*************************** 1. row ***************************
  select_type: SIMPLE
        table: titulos_big
         type: index
possible_keys: PRIMARY
          key: titulo_index
          ref: NULL
         rows: 998706
        Extra: Using where
*************************** 2. row ***************************
  select_type: SIMPLE
        table: rel_titulo_autor_big
         type: ref
possible_keys: PRIMARY,IDautor
          key: PRIMARY
          ref: biblioteca.titulos_big.tituloID
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
  select_type: SIMPLE
        table: autores
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
          ref: biblioteca.rel_titulo_autor_big.IDautor
         rows: 1
        Extra:
```

En este caso el orden de acceso a las tablas ha cambiado. El primer paso es ahora recorrer toda la tabla titulos_big usando el índice de la clave primaria. A partir de aquí se busca los autores de cada título accediendo primero a `rel_titulo_autor_big` y luego a `autor`.

Desafortunadamente, la estrategia de MySQL para determinar el orden óptimo de acceso a las tablas en un JOIN no es muy buena. De hecho, se comprueban todas las posibles combinaciones de tablas para determinar la mejor, y en el caso de tener muchas, eso supone un número muy alto de combinaciones que en ocasiones puede llegar a hacer que el tiempo dedicado por el optimizador sea mucho mayor que el de la consulta en sí ! Más adelante veremos como se puede forzar a MySQL a que siga un determinado orden.

8.3 Características del optimizador

Cuando se quiere comprobar si una consulta es eficiente, una fuente de problemas común es como MySQL trata los datos de prueba. SI no se sabe como funciona el optimizador, se puede perder mucho tiempo intentando solucionar un problema que no existe, o lo que es peor, que ha sido creado con los datos de prueba.

En general, MySQL usa un índice siempre que crea que usarlo dará mejor rendimiento que no hacerlo. Esto puede llevar a situaciones equivocadas durante la fase de pruebas. Esto es debido, en general, a dos problemas.

Poca diversidad

A pesar de generar miles o millones de registros de prueba, MySQL puede elegir no usar nuestros índices si los datos contienen poca diversidad. Por qué puede ocurrir ? Imaginemos una tabla que guarda datos históricos sobre el clima de muchas ciudades:

```
CREATE TABLE weather
(
  city       VARCHAR(100) NOT NULL,
  high_temp  TINYINT      NOT NULL,
  low_temp   TINYINT      NOT NULL,
  the_date   DATE         NOT NULL,
  INDEX (city),
  INDEX (the_date),
)
```

Supongamos que para hacer pruebas consideramos los datos de dos años, por ejemplo 1980 y 1981. Después de algunas consultas nos damos cuanta que MySQL no usa nunca el índice `the_date`. El problema es que usar ese índice solo puede descartar el 50% de los registros, y en ese caso el sistema considera que es más rápido leer la tabla directamente que usar el índice.

Más o menos el punto en el que MySQL decide usar un índice es cuando puede descartar más del 70% de los registros, aunque este número no viene de ninguna fórmula sino de la experiencia de los desarrolladores de MySQL. Además, diferentes motores de almacenamiento tienen umbrales diferentes.

La razón de esto es el tiempo de acceso al disco. Los índices se acceden en orden, y por lo tanto, no de manera secuencial según su disposición en el disco, por lo que las lecturas no son eficientes en este sentido. Su eficiencia viene de que se leen mucha menos cantidad de datos. Por otro lado, las tablas, a leerse sin orde, se hace siguiendo su disposición en disco, por lo que los tiempos de espera son muy eficientes, aunque como contrapartida se lee mucha más cantidad de datos.

Ordenación con índices

Uno de los puntos débiles de MySQL es la ordenación de los resultados de una consulta. Hay dos problemas con la ordenación. Primero, ordenar siempre añade coste de CPU, y eso no hay manera de resolverlo a parte de usar CPUs más rápidas. El otro problema importante es que hay que recordar que MySQL solo puede usar un índice por tabla en una consulta. De manera que si tenemos un índice que podría ser utilizado para ordenar, probablemente no lo será porque ya se habrá usado otro para acceder a los datos. Es el caso típico en que el optimizador accede a los datos a través de una clave primaria, y luego solicitamos ordenación por otra columna.

La única manera de solucionar el segundo problema es añadir la columna por la cual queremos ordenar al índice. Usando el ejemplo de los datos meteorológicos, si queremos que esta consulta sea eficiente:

```
SELECT * FROM weather WHERE city = 'Toledo' ORDER BY the_date DESC
```

Añadiremos este índice:

```
ALTER TABLE weather DROP INDEX city, ADD INDEX (city, the_date)
```

Aquí es importante recordar que el orden de las columnas en el índice es mu importante. Si queremos acceder a los datos a través de `city`, está columna debe aparecer lo más a la izquierda del índice.

En este punto podríamos estar tentados de suprimir el índice `the_date`, pero solo deberemos hacerlo cuando no haya ninguna consulta que use esta columna en una cláusula `WHERE`. Porque la parte `the_date` del índice compuesto no se puede usar para acceder a los datos ya que está en la parte derecha.

Consultas imposibles

Imaginemos una consulta como esta:

```
mysql> SELECT titulo FROM titulos_big WHERE tituloID < 100000 AND tituloID >
200000;
Empty set (0.16 sec)
```

Evidentemente no retorna ningún resultado ya que la cláusula WHERE is imposible que se cumpla. A pesar de ello, MySQL no nos avisa. Sin embargo, si usamos EXPLAIN:

```
mysql> EXPLAIN SELECT titulo FROM titulos_big WHERE tituloID < 100000 AND
tituloID > 200000 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: Impossible WHERE noticed after reading const tables
```

Vemos que MySQL sí que detecta que es una consulta imposible, y nos lo dice en el campo Extra.

8.4 Identificar consultas lentas

Saber porqué una consulta es lenta es difícil, pero para detectar cuales son esas consultas MySQL proporciona el mecanismo del *slow query log*. Este mecanismo se activa al poner el marcha el servidor con la opción:

```
 --log-slow-queries[=file_name]
```

Al activarlo, MySQL guardará información de todas las consultas que tarden más de `long_query_time` segundos y hayan leído como mínimo `min_examined_row_limit` `registros.`

Si lo activamos, MySQL guardará un registro de todas las consultas que tarden más de un tiempo especificado. Además de la consulta en sí, registra más datos.

```
# Time: 030303  0:51:27
# User@Host: user[user] @ client.example.com [192.168.50.12]
# Query_time: 25  Lock_time: 0  Rows_sent: 3949  Rows_examined: 378036
select ArticleHtmlFiles.SourceTag, ArticleHtmlFiles.AuxId from
ArticleHtmlFiles left
join Headlines on ArticleHtmlFiles.SourceTag = Headlines.SourceTag and
ArticleHtmlFiles.AuxId = Headlines.AuxId where Headlines.AuxId is NULL;
```

A pesar de que se nos da mucha información, hay que ser cuidadoso al interpretarla. El hecho de que una consulta haya tardado mucho tiempo en un determinado momento, no quiere decir que siempre vaya a serlo. Eso puede ser por varios motivos:

- La tabla puede haber estado bloqueada en el momento de hacer la consulta, y dentro del tiempo que ha tardado está incluido el tiempo de espera hasta que la tabla ha sido liberada. Par ello se nos proporciona el tiempo `Lock_time` que se ha esperado por bloqueos.
- No hay ningún parte de esa tabla en la cache. Por ejemplo, la primera vez que se ejecuta una consulta después de poner en marcha el servidor.

- Había un proceso haciendo un respaldo en ese mismo momento y eso reduce mucho el tiempo de acceso a disco.
- El servidor puede haber tenido un pico de trabajo justo en ese momento

En resumen, este registro nos avisa de errores potenciales, pero no es un prueba definitiva de que una consulta deba optimizarse. Ahora bien, si una misma consulta aparece muchas veces y en situaciones diferentes, hay muchas posibilidades de que haya que prestarle atención.

8.5 Modificar el comportamiento de MySQL

Muchos sistemas de bases de datos relaciones implementan una serie de atributos o *hints* que permiten dar pistas al optimizador para mejorar sus resultados. En esta sección veremos algunos de esos hints. Estos casi siempre aparecen justo después de SELECT:

```
SELECT SQL_CACHE * FROM mytable ...
```

Sin embargo,la mayoría de ellos no son SQL estándar, así que podemos usarlos de esta manera:

```
SELECT /*! SQL_CACHE */ * FROM mytable ...
```

De esta manera solo MySQL interpretará los hints, y el resto de servidores se los saltarán como si fuese un comentario. Incluso se puede especificar que versión de servidor puede usar el hint o no:

```
CREATE /*!32302 TEMPORARY */ TABLE t (a INT);
```

De esta manera el atributo TEMPORARY solo será tenido en cuenta por servidores MySQL con versión 3.23.02 o superiores.

Orden de los JOIN

MySQL no se preocupa del orden en que aparecen las tablas en un JOIN. EL optimizador probará todas las combinaciones y escogerá la que crea más eficiente. Esto puede provocar que algunas veces la elección no sea la más óptima. Para comprobarlo siempre podemos usar EXPLAIN. SI el resultado nos confirma que existe un orden mejor, se puede usar STRAIGHT_JOIN:

```
SELECT * FROM table1 STRAIGHT_JOIN table2 WHERE ...
```

En este caso el orden del JOIN será exactamente el indicado en el SELECT.

Uso de índices

El optimizador también decide qué índices usar en cada consulta. Y como en el caso de JOIN, también existen hints para forzar un comportamiento diferente al del optimizador. Por ejemplo, se puede forzar a MySQL a que escoja un índice de una lista que le demos, en lugar de considerarlos todos:

```
SELECT * FROM titulos USE INDEX (editIDindex, catIDindex) WHERE ...
```

O podemos hacer lo contrario y decirle a MySQL qué índices no queremos considerar, y que use los restantes:

```
SELECT * FROM titulos IGNORE INDEX (tituloIDindex) WHERE ...
```

Y si queremos forzar a usar un índice en concreto:

```
SELECT * FROM titulos FORCE INDEX (editIDindex) WHERE ...
```

Esta último hint puede ser desobedecido si MySQL detecta que es imposible resolver la consulta usándolo.

A partir de MySQL 5.1.17 ha habido algunos cambios a esta sintaxis. Primero, se puede usar el tributo USE INDEX vacío:

```
SELECT * FROM titulos USE INDEX () WHERE ...
```

Esto significa que queremos que MySQL no use ningún índice para esa tabla.

Otra novedad es que se puede especificar a qué parte de la consulta se aplica la restricción: `FOR ORDER BY, FOR GROUP BY, FOR JOIN`. Por ejemplo:

```
SELECT * FROM t1 USE INDEX (i1) IGNORE INDEX FOR ORDER BY (i2) ORDER BY a;
```

Con esto decimos al optimizador que no use el índice `i2` a la hora de realizar la ordenación.

Tamaño de los resultados

Hay unos hints que permiten decirle al servidor como tratar los resultados en función de su tamaño. Hay que usarlos solo cuando estemos realmente seguros de que los necesitamos para mejorar la eficiencia de una consulta. Porque si los usamos demasiado podemos hacer que el servidor vaya más lento en conjunto.

Si usamos SQL_BUFFER_RESULT forzamos a que MySQL guarde los resultados en una tabla temporal. Eso puede ser útil cuando el número de registros es muy alto y el cliente puede tardar mucho tiempo en procesarlos. El efecto es que si hemos hecho un bloqueo para realizar la consulta lo podremos liberar antes.

Otro hint que podemos usar cuando sabemos que el resultado de una consulta puede ser muy grande es SQL_BIG_RESULT. En es te caso MySQL puede ser mucho más agresivo a la hora de decidir usar tablas temporales en disco. Además puede decidir no crear índices en los resultados para ordenarlos.

Cache de consultas

Como hemos explicado anteriormente, MySQL tiene una cache para las consultas SELECT. Se puede controlar si una consulta ha de ser almacenada en la cache o no. El comportamiento de estos hints depende del valor de la variable `query_cache_type`:

- Si vale 0, entonces la cache está desconectada y no se puede forzar a que funcione.
- Si vale 1 la cache está activada y toda las consultas se guardan, excepto las que llevan SQL_NO_CACHE
- Si vale 2 la cache está desactivada excepto para aquellas consultas que llevan SQL_CACHE

9. Vistas

Las vistas (*views*) hacen posible el definir representaciones especiales de una tabla. Una vista se comporta de manera casi idéntica a una tabla. Se pueden consultar con SELECT y modificar con INSERT, UPDATE y DELETE. Las vistas se pueden usar en MySQL desde la versión 5.0. Hay dos razones básicas para usar vistas:

- **Seguridad**. Hay ocasiones en las que querríamos que un usuario pudiera acceder a una tabla, pero no a todos sus registros. Un ejemplo es una tabla con los datos de los empleados de una empresa. En genera, querremos que todos los empleados accedan a esa tabla, pero que no puedan acceder a todos las columnas de cada registro donde probablemente hay información confidencial.

  La solución es crear una vista de la tabla de empleados en la que solo aparezcan las columnas con información pública, y permitir el acceso a la vista y no a la tabla original.

- **Eficiencia**. En muchas ocasiones las aplicaciones ejecutan muchas veces la misma consulta para seleccionar un mismo conjunto de registros de una tabla. En lugar de dejar que cada usuario o programador repita una y otra vez la misma consulta, podemos crear una vista con dicha consulta y hacerla visible a los usuarios o programadores.

Las vistas son como tablas virtuales cuyo contenido es el resultado de un SELECT. Por ejemplo, si queremos que los algunos usuarios de la base de datos `biblioteca` tengan acceso a la tabla `titulos`, pero solo a las columnas `tituloID`, `titulo` y `subtitulo` ordenadas alfabéticamente haremos los siguiente:

```
CREATE VIEW v1 AS
  SELECT tituloID, titulo, subtitulo FROM titulos
  ORDER BY title, subtitle
```

Pero la definición puede ser tan compleja como nuestras necesidades. Otro ejemplo: si queremos determinados usuarios de biblioteca solo puedan acceder a la lista de libros en español, y de cada uno solo saber la editorial y la categoría crearemos la siguiente vista:

```
CREATE VIEW v2 AS
  SELECT titulo, nombreEditorial, nombreCat FROM titulos, editoriales,
categorias
  WHERE titulos.editID = editoriales.editID
  AND titulos.catID = categorias.catID
  AND langID = 5
```

Para modificar una vista con INSERT, UPDATE y DELETE la única restricción que se aplica es el comando SELECT que se ha usado para crearla. Para ello se aplican las siguientes reglas:

- El comando SELECT no debe contener GROUP BY, DISTINCT, LIMIT, UNION o HAVING.
- Las vistas que se forman con la consulta de dos o más tablas son casi siempre imposibles de modificar.
- Las vistas deberían contener todas las columnas para las que hay claves primarias o índices únicos o claves foráneas. Si no lo hacen, la opción `updatable_views_with_limit` de MySQL define qué cambios deben producir

un aviso (comportamiento por defecto) o dar un error.

Al crear una vista hay una serie de atributos que podemos usar. La sintaxis de CREATE VIEW es la siguiente:

```
CREATE [OR REPLACE] [ALGORITHM = UNDEFINED | MERGE | TEMPTABLE]
VIEW name [(columnlist)] AS select command
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

El significado de las opciones es:

- OR REPLACE significa que si existía una vista con el mismo nombre debe de ser reemplazada con la nueva definición sin dar ningún mensaje de error.
- ALGORITHM nos dice como se representa la vista internamente. Esta opción está sin documentar hasta el momento.
- WITH CHECK OPTION significa que está permitido modificar los registros solo si las condiciones WHERE del SELECT se continuaran cumpliendo después de la modificación. Esta opción solo tiene sentido si la vista que queremos crear va a ser modificada.
    - La variante LOCAL afecta a las vistas que se derivan a su vez de otras vistas. Esto significa que solo se consideran las condiciones WHERE de la primera vista, no las de las siguientes.
    - El efecto de la variante CASCADE es el contrario. Se consideran todas las condiciones WHERE.

10. Procedimientos Almacenados

Los procedimientos almacenados (*Stored Procedures, SP*) suponen una importante innovación introducida en MySQL en la versión 5.0. Son funciones SQL definidas por el usuario que se ejecutan directamente en el servidor. Con los SP es posible guardar parte de la lógica de la aplicación cliente-servidor en el servidor.

En este capítulo veremos porqué los SP son una buena idea (más velocidad, más seguridad para los datos, menos duplicación de código, ...).

En general, laz razones para usar SP son:

- **Velocidad**. En muchas ocasiones, determinadas operaciones de una aplicación pueden suponer que muchos datos sean enviados del servidor al cliente y viceversa. El cliente ejecuta un SELECT que produce muchos resultados, que son enviados desde el servidor al cliente. El cliente los procesa y envía una gran lista de UPDATEs de vuelta al servidor. Si todos estos pasos se pudieran ejecutar en el servidor nos evitaríamos una gran sobrecarga de la comunicación cliente/servidor.

  Además, MySQL puede preprocesar el código de los SP, evitando tener que compilar el código de un SP cada vez que se ejecuta. Sin embargo, MySQl no documenta que tipo de optimizaciones realiza cuando precompila los SP.

  De todas formas, usar SP no garantiza una mejora en la velocidad de ejecución de nuestras aplicaciones. Solo se consiguen mejoras si el código del SP es eficiente. Y como SQL es un lenguaje mucho más primitivo que otros, no siempre es posible escribir código eficiente.

  Otro aspecto a tener en cuenta es que los SP aumentan la carga del servidor, mientras que la reducen en la parte del cliente. El resultado de este balance dependerá mucho de como esté configurado nuestro sistema y de cuales sean los cuellos de botella de nuestra aplicación.

- **Reutilización de código**. Ocurre frecuentemente que diferentes aplicaciones, o diferentes usuarios de una misma aplicación realizan la misma operación una y otra vez. Si la lógica de estas operaciones puede ser trasladada al servidor en forma de SPs entonces se consigue reducir la redundancia de código, y además, y muy importante, el sistema es mucho más fácil de mantener.

- **Seguridad de los datos**. Hay muchas aplicaciones en la que la integridad de los datos es crítica, como las relacionadas con el sector financiero. En estos casos, es deseable que los usuarios no puedan acceder directamente a determinadas tablas. La manera de obtener los datos es a través de SPs que se encargan de realizar los SELECT, UPTATE o INSERT a través de los SPs que son visibles a los usuarios. Un beneficio adicional es que los administradores pueden monitorizar los accesos a la base de datos mucho más fácilmente.

La gran desventaja de los SP es que normalmente su portabilidad es muy baja, con los que las funciones escritas para un sistema concreto no se ejecutaran directamente en otro sistema. La razón es que cada sistema de bases de datos usa su propia sintaxis y sus propias extensiones para los SP, de forma que estos no están estandarizados.

Definir un SP

La mejor manera de ilustrar como funciona un SP es con un ejemplo. El primer detalle a tener en cuenta es que al definir un SP utilizaremos el carácter ';' que es el delimitador por defecto. Por esta razón deberemos cambiar el delimitador, y en este ejemplo usaremos la secuencia '$$'.:

```
mysql> delimiter $$
```

Ahora el ejemplo. Supongamos que tenemos una tabla en la que almacenamos en forma de texto comentarios de los usuarios, de un máximo de 255 caracteres de longitud. Además, tenemos una columna en la que guardamos un resumen de `n<255` caracteres del comentario. A la hora de guardar el resumen podemos hacer dos cosas: ignorar el límite de `n` y MySQL truncará la cadena y se quedará con el principio de comentario, o hacer algo más complicado como guardar el principio y el final del comentario, colocando la cadena '...' en medio. Para hacer esto definiremos un SP como este:

```
  mysql> CREATE FUNCTION acortar(s VARCHAR(255), n INT)
      ->   RETURNS VARCHAR(255)
      -> BEGIN
      ->   IF ISNULL(s) THEN
      ->     RETURN '';
      ->   ELSEIF n<15 THEN
      ->     RETURN LEFT(s, n);
      ->   ELSE
      ->     IF CHAR_LENGTH(s) <= n THEN
      ->       RETURN s;
      ->     ELSE
      ->       RETURN CONCAT(LEFT(s, n-10), ' ... ', RIGHT(s, 5));
      ->     END IF;
      ->   END IF;
      -> END$$
```

Este procedimiento lo que hace es recibir una cadena de caracteres s (tipo VARCHAR(255)) y un entero n (tipo INT) y retorna una cadena de caracteres (VARCHAR(255)). El entero n indica a qué longitud queremos acortar la cadena inicial s. Si n<15, consideramos que son muy pocos caracteres como para guardar el principio y el final y simplemente guardamos el principio. Si n>=15 y la cadena de entrada es mayor que n entonces generamos el resumen del comentario de la manera que hemos explicado anteriormente.

Para probar si funciona:

```
mysql> SELECT acortar("Este comentario no significa nada y solo sirve de
ejemplo", 15);
+---------------------------------------------------------------------------+
| acortar("Este comentario no significa nada y solo sirve de ejemplo", 15) |
+---------------------------------------------------------------------------+
| Este  ... emplo                                                           |
+---------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

O podemos utilizarlo para generar una lista compacta de los títulos de la base de datos `biblioteca`:

```
mysql> SELECT titulo, acortar(titulo, 20) FROM titulos LIMIT 10;
+-------------------------------------------+----------------------+
| titulo                                    | acortar(titulo, 20)  |
+-------------------------------------------+----------------------+
| Linux                                     | Linux                |
| The Definitive Guide to Excel VBA         | The Defini ... l VBA |
| Client/Server Survival Guide              | Client/Ser ... Guide |
| Web Application Development with PHP 4.0  | Web Applic ... P 4.0 |
```

```
| MySQL                                     | MySQL                 |
| MySQL & mSQL                              | MySQL & mSQL          |
| A Guide to the SQL Standard               | A Guide to ... ndard  |
| Visual Basic 6                            | Visual Basic 6        |
| Excel 2000 programmieren                  | Excel 2000 ... ieren  |
| PHP - Introducción                        | PHP - Introducción    |
+-------------------------------------------+-----------------------+
10 rows in set (0.00 sec)
```

Implementación de SP

Afortunadamente los SP de MySQL siguen el estándar SQL:2003. De esta manera, los SP de MySQL se pueden portar fácilmente al sistema DB/2 de IBM. Sin embargo, no se pueden portar a Oracle o Microsoft SQL Server ya que estos no siguen el estándar.

El almacenamiento interno de los SP se hace en la tabla `mysql.proc`. En las columnas de esta tabla están almacenados todos los datos relativos al SP. Si no tenemos acceso a la abse de datos `mysql`, podemos recuperar la misma información con `INFORMATION_SCHEMA.ROUTINES`.

Crear, editar y borrar SPs

La sintaxis para crear un SP es:

```
CREATE FUNCTION nombre ([parametro[,...]) RETURNS tipo_de_datos
[opciones] codigo_sql
CREATE PROCEDURE nombre ([parametro[,...])
[opciones] codigo_sql
```

La única diferencia entre los dos es que `FUNCTION` retorna un valor y `PROCEDURE` no. El SP creado se asocia a la base de datos actual. Puede haber una `FUNCTION` y un `PROCEDURE` con el mismo nombre. Los parámetros tienen la sintaxis:

```
parametro:
      [IN | OUT | INOUT] nombre_parametro
```

Los parámetros de `FUNCTION` son todos `IN` por defecto. Los parámetros que son `OUT` en un `PROCEDURE` son valores que se retornan y que se inicializan a `NULL`. Los `INOUT` son parámetros de `PROCEDURE` que son tanto de entrada como de salida.

Algunas de las opciones que se puede especificar son:

- LANGUAGE SQL: el único valor posible de momento. Se prevé que en el futuro se puedan usar otros lenguajes de programación.
- [NOT] DETERMINISTIC: un SP está considerado determinista cuando siempre retorna el mismo resultado con los mismos parámetros. Los SP que dependen de una tabla de la base de datos son no deterministas obviamente.

  Por defecto los SP son no deterministas. Sin embargo, los SP que sí lo son pueden ejecutarse de una manera mucho más eficiente ya que, por ejemplo, los resultados se pueden almacenar en una cache. De cualquier manera, actualmente esta opción es ignorada por MySQL.
- SQL SECURITY DEFINER o INVOKER: el modo SQL SECURITY especifica los privilegios de acceso al SP.
- COMMENT 'text': el comentario se almacena con el SP a modo de documentación.

Para borrar un SP:

```
DELETE FUNCTION [IF EXISTS] nombre
DELETE PROCEDURE [IF EXISTS] nombre
```

La opción IF EXISTS hace que si el SP no existe no se produzca ningún error.

Los SP pueden modificarse con ALTER. Se necesita el privilegio ALTER ROUTINE para hacerlo. La sintaxis es:

```
ALTER FUNCTION/PROCEDURE nombre
  [NAME nombre_nuevo]
  [SQL SECURITY DEFINER/INVOKER]
  [COMMENT 'nuevo comentario']
```

Con la versión actual de MySQL (5.1) no se puede modificar el código del SP.

10.1 Sintaxis

Los SP están compuestos principalmente de un conjunto de instrucciones SQL ordinarias. Sin embargo hay una serie de instrucciones adicionales que permiten control de flujo, alternativas, cursores, ... Primero veremos cuales son las diferencias entre FUNCTION y PROCEDURE, que están resumidas en Tabla 8.

	PROCEDURE	**FUNCTION**
Llamada	Solo con CALL	Posible en todas las instrucciones SQL (SELECT, UPDATE, ...)
Retorno	Puede retornar uno o más SELECT	Retorna un valor único de un tipo determinado.
Parámetros	Permite por valor y por referencia (IN, OUT, INOUT)	Solo parámetros por valor.
Instrucciones permitidas	Todas las SQL	No se puede acceder a tablas
Llamadas a otras funciones o procedimientos	Puede llamar a otros procedimientos y/o funciones	Solo puede llamar otras funciones

Tabla 8: Diferencias entre FUNCTION y PROCEDURE

Las reglas generales para escribir SP son:

- Punto y coma. Para separar las diferentes instrucciones que componen un SP se utiliza el carácter ';'.
- BEGIN-END. Las instrucciones que no se encuentran entre dos palabras reservadas (p.e. IF, THEN, ELSE) han de ponerse entre un BEGIN y un END.
- Salto de línea. Los saltos de línea son considerados como espacios en blanco.
- Variables. Las variables locales y los parámetros se acceden sin un carácter @ al principio. El resto de variables SQL deben comenzar con @.
- Mayúsculas/minúsculas. MySQL no las distingue al definir el nombre del SP.
- Caracteres especiales. Evitar el uso de caracteres fuera del código ASCII (acentos,

ñ, ...). A pesar de estar permitidos, pueden aparecer problemas al administrar la base de datos.

- Comentarios. Los comentarios se introducen con '--' y se extienden hasta el final de la línea.

10.2 Llamadas

Las funciones y los procedimientos tienen diferentes maneras de invocarse. Además, los SP están asociados a una base de datos con lo que si queremos ejecutar un SP de otra base de datos debemos hacerlo poniendo el nombre de la base de datos delante seguido por un punto (p.e. nombre_bd.nombre_sp())

Funciones

Las funciones, como las predefinidas por SQL, se pueden integrar en comandos SQL. Por ejemplo, la función acortar que hemos definido antes la hemos usado dentro de un SELECT. Por ejemplo:

```
SELECT acortar(titulo, 30), acortar(titulo, 20) FROM titulos
UPDATE titulos SET titulo = acortar(titulo, 70) WHERE CHAR_LENGHT(title)>70
```

Además también se puede almacenar el resultado de una función en una variable SQL:

```
mysql> SET @s = " una cadena de caracteres my larga ";
Query OK, 0 rows affected (0.00 sec)

mysql> SET @a = acortar(@s, 15);
Query OK, 0 rows affected (0.00 sec)

mysql> select acortar(@s, 10) INTO @b;
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @s, @a,@b;
+-------------------------------------+-----------------+------------+
| @s                                  | @a              | @b         |
+-------------------------------------+-----------------+------------+
|  una cadena de caracteres my larga  |  una  ... arga  |  una caden |
+-------------------------------------+-----------------+------------+
1 row in set (0.00 sec)
```

Procedimientos

Los procedimientos deben ser llamados con CALL. Se puede retornar como resultadp una tabla (igual que con un comando SELECT). No se pueden usar procedimientos dentro de sentencias SQL.

Por ejemplo, podemos usar un procedimiento para obtener el titulo, subtitulo y editorial de cada libro:

```
CREATE PROCEDURE imartin.obtener_titulo(IN id INT)
BEGIN
  SELECT titulo, subtitulo, nombreEdit
  FROM titulos, editoriales
  WHERE titulos.tituloID = id
  AND titulos.editID = editoriales.editID;
END
```

Y para usarlo:

```
mysql> CALL obtener_titulo(1);
+--------+-------------------------------------------+-------------+
| titulo | subtitulo                                 | nombreEdit  |
+--------+-------------------------------------------+-------------+
| Linux  | Instalacion, Configuracion, Administracion | Grupo Anaya |
+--------+-------------------------------------------+-------------+
```

En el caso de procedimientos con variables de salida, el resultado solo puede ser evaluado si se pasa una variable SQL:

```
mysql> delimiter ;
mysql> CREATE PROCEDURE mitad(IN a INT, OUT b INT) BEGIN SET b = a/2; END$$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> CALL mitad(10, @resultado);
Query OK, 0 rows affected (0.04 sec)

mysql> SELECT @resultado;
+------------+
| @resultado |
+------------+
|          5 |
+------------+
```

Recursividad

En el código de un procedimiento se pueden llamar a otros procedimientos o funciones, mientras que en el código de una función solo se pueden llamar a otras funciones. Además, funciones y procedimientos pueden llamarse a ellos mismos, con lo cual se pueden implementar algoritmos recursivos. Por ejemplo, podemos implementar el típo ejemplo de calcular el factorial con una función:

```
CREATE FUNCTION factorial(n BIGINT) RETURNS BIGINT
BEGIN
  IF n>=2 THEN
    RETURN n * factorial(n-1);
  ELSE
    RETURN n;
  END IF;
END
```

Al instalar el servidor de MySQL la recursión está desactivada por defecto. Para poder activarla hay que cambiar la variable `max_sp_recursion_depth` a un valor mayor que 0. El valor que coloquemos será el nivel máximo de recursión permitido. Hay que tener cuidado con esta variable ya que al activar la recursión, y sobre todo con un valor alto, podemos sobrecargar el servidor si nuestras funciones o procedimientos están mal diseñados.

10.3 Parámetros y valores de retorno

Hay una serie de detalles importantes acerca de como MySQL gestiona los parámetros que se pasan a funciones y procedimientos.

Procedimientos

Recordar que la sintaxis para crear un procedimiento es:

```
CREATE PROCEDURE nombre ([[IN OUT INOUT] nombre_parametro
tipo_parametro[,...])
```

Los atributos IN, OUT e INOUT determinan si el parámetro será usado solo como entrada, solo como salida, o como en las dos direcciones.

Se pueden usar todos los tipos de datos de MySQL. Sin embargo, no se pueden usar atributos para los tipos, como NULL, NOT NULL. Actualmente MySQL no hace ningún tipo de comprobación de tipos de los parámetros.

A pesar que los procedimientos no pueden retornar un solo valor, sí que se pueden usar instrucciones SELECT normales. Incluso se pueden ejecutar varios SELECT en secuencia. En ese caso, el procedimiento retorna el resultado en forma de varias tablas. Sin embargo, solo los lenguajes de programación que soportan el atributo MULTI_RESULT pueden acceder a esos resultados múltiples.

Funciones

La sintaxis para crear un función es:

```
CREATE FUNCTION nombre ([nombre_parametro tipo_datos[,...]) RETURNS
tipo_datos
```

En este caso todos los parámetros son de entrada (por valor).

Las funcionas retornan un solo valor del tipo especificado con la sentencia RETURN, que además termina la ejecución de la función.

10.4 Variables

Declaración

Se pueden declarar variables locales dentro de los procedimientos y funciones. La sitaxis es:

```
DECFLARE nombre_var1, nombre_var2, .... tipo_datos [DEFAULT valor];
```

Se debe definir el tipo de datos para todas las variables. Se inicializan con NULL, a menos que se les de un valor explícito.

Hay que tener cuidado de no dar a las variables los mismos nombres que columnas o tablas de la base de datos ya que, a pesar de estar permitido, puede dar lugar a errores muy difíciles de rastrear.

Asignación

Las asignaciones de tipo *x = x + 1* no están permitidas en SQL. Hay que usar SET o SELECT INTO. Este último es una variante de SELECT que acaba en INTO nombre_variable. Esta variante solo se puede usar cuando la instrucción SELECT retorna un solo registro. En las funciones solo se puede usar SET, y no SELECT INTO.

Ejemplos:

```
SET var1=valor1, var2=valor2, ...
SELECT var:=valor
SELECT 2*7 INTO var
```

```
SELECT COUNT(*) FROM tabla WHERE condicion INTO var
SELECT titulo, subtitulo FROM titulos WHERE tituloID=3
  INTO mititulo,misubtitulo
```

10.5 Encapsulación de instrucciones

Cada función o procedimiento consiste en una o más instrucciones SQL que comienzan con un BEGIN y terminan con un END. Una construcción BEGIN-END puede incluirse dentro de otra si tenemos que declarar determinadas variables que solo van a ser utilizadas en un determinado ámbito. Dentro de una construcción BEGIN-END hay que seguir un determinado orden:

```
BEGIN
  DECLARE variables;
  DECLARE cursores;
  DECLARE condiciones;
  DECLARE handlers;
  instrucciones SQL;
END;
```

Antes de BEGIN puede añadirse una etiqueta, que se puede usar en conjunción con la instrucción LEAVE que sirve para abandonar un determinado ámbito definido por un BEGIN-END. Sintaxis:

```
nombre_bloque:BEGIN
  instrucciones;
  IF condicion THEN LEAVE nombre_bloque; ENDE IF;
  instrucciones;
END nombre_bloque;
```

De esta manera, con el comando LEAVE podemos escoger qué ámbito queremos abandonar.

10.6 Control de flujo

Hay básicamente dos maneras de controlar el flujo en SPs.

IF-THE-ELSE

La sintaxis de esta construcción es:

```
IF condicion THEN
  instrucciones;
[ELSE IF condicion THEN
  instrucciones;]
[ELSE
  instrucciones;]
END IF;
```

Nos es necesario usar BEGIN-END dentro de estas estructuras de control. La condición puede formularse usando consultas con WHERE o HAVING.

CASE

Es una variante de IF que es útil cuando todas las condiciones dependen de un solo valor de una expresión. La sintaxis es:

```
CASE expresion
WHEN valor1 THEN
  instrucciones;
[WHEN valor2 THEN
  instrucciones;]
[ELSE
  instrucciones;]
END CASE;
```

10.7 Bucles

MySQL ofrece una serie de opciones para construir bucles. Curiosamente, la opción de FOR no existe.

REPEAT-UNTIL

Las instrucciones dentro de esta construcción se ejecutan hasta que se cumple una determinada condición. Como la condición se evalúa al final del bucle, este siempre se ejecuta al menos una vez.

El bucle puede llevar opcional mente una etiqueta, que puede ser usada para abandonar el bucle usando la instrucción LEAVE, o repetir una iteración con ITERATE (ver más adelante).

```
[nombre_loop:] REPEAT
  instrucciones;
UNTIL condicion1
END REPEAT [nombre_loop];
```

Por ejemplo, ahora mostraremos una función que retorna una cadena de caracteres compuesta por n caracteres '*':

```
CREATE FUNCTION asteriscos(n INT) RETURNS TEXT
BEGIN
  DECLARE i INT DEFAULT 0;
  DECLARE s TEXT DEFAULT '';
  bucle1: REPEAT
    SET i = i + 1;
    SET s = CONCAT(s, “*”);
  UNTIL i>=n END REPEAT;
  RETURN s;
END;
```

WHILE-DO

Las instrucciones entre WHILE y DO se ejecutan siempre que la condición correspondiente sea cierta. Como la condición se evalúa al principio del bucle, es posible que no se produzca ninguna iteración. Sintaxis:

```
[nombre_bucle:] WHILE condicion DO
  instrucciones;
END WHILE [nombre-.bucle];
```

LOOP

Con esta construcción se define un bucle que se ejecuta hasta que se sale de él mediante una instrucción LEAVE. Sintaxis:

```
nombre_bucle: LOOP
  instrucciones;
END LOOP nombre_loop;
```

LEAVE e ITERATE

LEAVE nombre_bucle aborta la ejecución de un bucle o de una construcción BEGIN-END.

ITERATE nombre_bucle sirve para ejecutar las instrucciones del bucle nombre_bucle una vez. ITERATE no puede usarse con BEGIN-END.

10.8 Gestión de errores (handlers)

Durante la ejecución de instrucciones SQl dentro de un SP pueden producirse errores. SQL define un mecanismo de handlers para gestionar esos errores. Un handler debe ser definido después de la declaración de variables, cursores y condiciones, pero antes del bloque BEGIN-END. Sintaxis:

```
DECLARE tipo HANDLER FOR condicion1, condicion2, ... instrucción;
```

Los elementos que intervienen en esta declaración son:

- `tipo`: puede ser CONTINUE o EXIT. El primero significa que la ejecución del programa continuará a pesar del error. EXIT significa salir del bloque BEGIN-END. Hay prevista una tercera opción que será UNDO y que servirá para deshacer los cambios producidos hasta ese momento por el SP.
- `condicion`: indica bajo qué condiciones debería activarse el handler. Hay varias opciones:
  - SQLSTATE 'codigo_error': especifica un error concreto mediante su código.
  - SQLWARNING: comprende todos los estados SQLSTATE 01nnn.
  - NOT FOUND: comprende el resto de errores.
  - codigo_error_mysql: se especifica el número de error MySQL.
  - nombre_condicion: se refiera a una condición que ha sido declarada entes con DECLARE CONDITION.
- `instrucción`: esta instrucción es ejecutada cuando se produce un error.

Las condiciones hacen posible dar a cierto grupo de errores un nombre claro. La sintaxis para declarar una condición es:

```
DECLARE nombre CONDITION FOR condicion;
```

La condición se puede formular con `SQLSTATE 'codigo_error'` o `num_error_mysql`. Por ejemplo, si queremos gestionar el error que se produce al insertar un registro con una clave duplicada, haremos esto:

```
DECLARE clavedup VARCHAR(100);
DECLARE clave_duplicada CONDITION FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR clave_duplicada SET mi_error='clavedup'
```

Desgraciadamente MySQL no permite que se pueda activar un error en un momento

dado. La única manera es ejecutando una sentencia que sabemos que producirá el error deseado.

10.9 Cursores

Un cursor funciona como un puntero a un registro. Con ellos se puede iterar sobre todos los registros de una tabla. Normalmente se usan para facilitar el diseño de funciones y procedimientos que modifican los datos de una tabla, y que si se tuvieran que hacer con UPDATE serían muy complicados.

De todas maneras, hay detractores del uso de cursores. La razón que esgrimen es que cualquier algoritmo que use cursores se puede reescribir de manera más elegante, y a veces más eficiente, sin usar cursores.

Sintaxis

El uso de cursores requiere varios pasos. Primero hay que declarar el cursor con:

```
DELCARE nombre_cursor CURSOR FOR SELECT ...;
```

Se puede usar cualquier comando SELECT. Después hay que activar el cursor con:

```
OPEN nombre_cursor;
```

A partir de ese momento se puede usar el comando FETCH:

```
FETCH nombre_cursor INTO var1, var2, ...;
```

De esta manera, la primera columna del registro al que apunta `nombre_cursor` se almacena en la variable `var1`, la segunda en `var2`, y así sucesivamente. Estas variables tienen que haber sido declaradas con anterioridad, y han de ser del tipo correcto.

Para saber cuando se han acabado de leer los registros correspondientes al SELECT del cursor, MySQL emite el error 1329 (*no data to fetch*) que corresponde a SQLSTATE 0200. Este error no se puede evitar, pero se puede capturar con un handler. De esta manera, siempre que usemos cursores tendremos que definir el handler correspondiente. Normalmente se usa NOT FOUND como condición, donde se engloban todos los SQLSTATE 0snnn.

El cursor se puede cerrar con:

```
CLOSE nombre_cursor;
```

De todas maneras, esto no se suele hacer ya que el cursor es automáticamente cerrado al final del bloque BEGIN-END.

Limitaciones

Hay tres limitaciones principales en el uso de cursores:

- Los cursores son solo de lectura, no se pueden modificar los datos a los que apunta.
- Los cursores solo pueden avanzar, de manera que los datos deben ser procesados en el orden en el que han sido proporcionados por el servidor.
- No se puede cambiar la estructura de las tablas mientras se están leyendo datos

con un cursor. Si se hace, MySQL no garantiza un comportamiento consistente.

Como ejemplo haremos un procedimiento que recorre todos los registros de la tabla titulo y suma el número de caracteres de titulo y subtitulo. Al final, la suma se divide por el número de registros lo que nos da el número medio de caracteres para titulo más subtitulo:

```
CREATE PROCEDURE biblioteca.testCursor(OUT longitud_media DOUBLE)
BEGIN
  DECLARE t, subt VARCHAR(100);
  DECLARE terminado INT DEFAULT 0;
  DECLARE n BIGINT DEFAULT 0;
  DECLARE cnt INT;
  DECLARE miCursor CURSOR FOR
    SELECT titulo, subtitulo FROM titulos;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET terminado=1;
    SELECT COUNT(*) FROM titulos INTO cnt;
  OPEN miCursor;
  miBucle: LOOP
    FETCH miCursor INTO t, subt;
    IF terminado=1 THEN LEAVE miBucle; END IF;
    SET n = n + CHAR_LENGTH(t);
    IF NOT ISNULL(subt) THEN
      SET n = n + CHAR_LENGTH(subt);
    END IF;
  END LOOP miBucle;
  SET longitud-media = n/cnt;
END
```

Y si lo ejecutamos obtendremos algo como:

```
mysql> CALL testcursor(@result);
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT @result;
+---------------+
| @result       |
+---------------+
| 31.629629629 |
+---------------+
```

Y para ilustrar como se podría haber hecho lo mismo sin un SP:

```
mysql> SELECT (SUM(CHAR_LENGTH(titulo)) +
SUM(CHAR_LENGTH(subtitulo)))/COUNT(*) AS longitud_media FROM titulos;
+-----------------+
| longitud_media |
+-----------------+
|        31.6296 |
+-----------------+
```

11. Triggers

Es un mecanismo muy ligado a los SP. Los triggers son conjuntos de sentencias SQL o SPs que son ejecutados automáticamente antes o después de una modificación de la base de datos (UPDATE, INSERT, DELETE). Se usan para mantener condiciones sobre los datos de la base de datos y para monitorizar las operaciones.

Hay que tener en cuenta que un trigger se ejecuta por cada modificación, así que si las operaciones a realizar automáticamente son muy complicadas, puede llevar a una ralentización considerable de nuestra aplicación. Esto se puede dar particularmente cuando se hacen modificaciones a muchos registros al mismo tiempo, ya que se ejecutará un trigger por cada registro modificado.

Lo que explicaremos a continuación sobre triggers se aplica solamente a versiones siguales o superiores a 5.1.6.

La sintaxis para la creación de triggers es:

```
CREATE
    [DEFINER = { usuario | CURRENT_USER }]
    TRIGGER nombre_trigger tiempo_trigger evento_trigger
    ON nombre_tabla FOR EACH ROW sentencia_trigge
```

Un trigger es un evento ligado a una base de datos y a la tabla `nombre_tabla`, que ha de ser de tipo permanente, es decir, no puede ser de tipo TEMPORARY ni VIEW. El trigger se ejecuta cuando se produce una determinada operación. Para crear un trigger se necesita el privilegio TRIGGER para la tabla asociada.

El atributo DEFINER determina el contexto de seguridad que se usará para determinar los privilegios de acceso en el momento de activación. El atributo nombre_trigger determina si la activación se debe de hacer antes (BEFORE) o después (AFTER) de modificar un registro.

El tipo de acción que activa el trigger viene determinado por evento_trigger. Los valores posibles son:

- INSERT. El trigger se activa cuando se inserta un registro. Esto incluye las instrucciones SQL INSERT, LOAD DATA y REPLACE.
- UPDATE. El trigger se activa cuando se modifica un registro. Esto corresponde a la instrucción SQL UPDATE.
- DELETE. El trigger se activa cuando se borra un registro. Por ejemplo cuando se usan instrucciones SQL como DELETE y REPLACE. Sin embargo, si se borran registro usando DROP TABLE o TRUCATE no se activan estos triggers ya que estas instrucciones no se transforman en DELETE.

Es importante recalcar que estos tres tipos de eventos no se corresponden con las instrucciones SQL que tienen el mismo nombre, sino que son tipos de eventos que pueden englobar uno o más tipos de instrucciones SQL.

Como ejemplo de posible confusión tenemos el siguiente comando:

```
INSERT INTO ... ON DUPLICATE KEY UPDATE ...
```

A pesar de se un comando INSERT, se pueden producir modificaciones si hay claves

duplicados, con lo que en este solo comando se pueden activar triggers de tipo INSERT y UPDATE.

No se pueden tener dos triggers definidos para la misma tabla, el mismo tiempo y el mismo evento. Por ejemplo, para una tabla solo puede haber un trigger del tipo BEFORE INSERT.

La acción SQL que se ejecuta si se activa el truigger es `sentencia_trigger`. Si se quieren ejecutar varias instrucciones SQL se puede usar BEGIN-END. Eso quiere decir que se puede usar la misma sintaxis que para los procedimientos almacenados. De todas maneras, hay una serie de restricciones para los triggers en relación a las instrucciones que están permitidas. Las más importantes son:

- `LOCK TABLES, UNLOCK TABLES.`
- `LOAD DATA, LOAD TABLE.`
- Las instrucciones `PREPARE, EXECUTE, DEALLOCATE PREPARE` no se pueden usar.
- No se pueden hacer commit ni rollback.

Más información en:

http://dev.mysql.com/doc/refman/5.1/en/routine-restrictions.html

Otra limitación en este momento es que los triggers no se activan cuando se ejecuta un CASCADE en una clave foránea. Esta restricción está prevista eliminarla pronto.

Veamos un ejemplo de como funcionan los triggers:

```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE test4(
  a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  b4 INT DEFAULT 0
);

DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
|

DELIMITER ;

INSERT INTO test3 (a3) VALUES
  (NULL), (NULL), (NULL), (NULL), (NULL),
  (NULL), (NULL), (NULL), (NULL), (NULL);

INSERT INTO test4 (a4) VALUES
  (0), (0), (0), (0), (0), (0), (0), (0), (0), (0);
```

Ahora realizamos un INSERT en la la tablas test1:

```
mysql> INSERT INTO test1 VALUES (1), (3), (1), (7), (1), (8), (4), (4);
Query OK, 8 rows affected (0.04 sec)
```

```
Records: 8  Duplicates: 0  Warnings: 0
```

En esta instrucción, al haberse producido INSERTs en la tabla test1, el trigger testref se ha activado para cada uno de los INSERTS. Veamos si se ha ejecutado correctamente:

```
mysql> SELECT * FROM test1;
+------+
| a1   |
+------+
|    1 |
|    3 |
|    1 |
|    7 |
|    1 |
|    8 |
|    4 |
|    4 |
+------+
8 rows in set (0.00 sec)
```

La tabla test1, evidentemente, contiene los valores que hemos insertado directamente. Veamos el resto de tablas afectadas. La tabla `test2` ha quedado así:

```
mysql> SELECT * FROM test2;
+------+
| a2   |
+------+
|    1 |
|    3 |
|    1 |
|    7 |
|    1 |
|    8 |
|    4 |
|    4 |
+------+
8 rows in set (0.00 sec)
```

Esto es debido a que el trigger lo único que hace es insertar un registro en `test2` que en la columna `a2` contiene los mismo que la columna `a1` del nuevo registro (`NEW.a1`) que se está a punto de insertar.

La tabla test3 ha quedado así:

```
mysql> SELECT * FROM test3;
+----+
| a3 |
+----+
|  2 |
|  5 |
|  6 |
|  9 |
| 10 |
+----+
5 rows in set (0.00 sec)
```

La tabla `test3` se había inicializado insertando registro con valor `NULL`, pero al ser su columna `a3` de tipo `NOT NULL PRIMARY KEY`, el sistema inicializó los valores de esta columna con 1,2,3, ... 10. Al ejecutarse el segundo `INSERT`, el que activa los triggers, cada vez que se inserta un registro en `test1`, se borran todos los registro de `test3` cuya columna `a3` es igual a la columna `a1` del registro que se está a punto de insertar.

Y finalmente veamos como ha quedado la tabla `test4`:

```
mysql> SELECT * FROM test4;
+----+------+
| a4 | b4   |
+----+------+
|  1 |    3 |
|  2 |    0 |
|  3 |    1 |
|  4 |    2 |
|  5 |    0 |
|  6 |    0 |
|  7 |    1 |
|  8 |    1 |
|  9 |    0 |
| 10 |    0 |
+----+------+
10 rows in set (0.00 sec)
```

Si nos fijamos en la instrucción del trigger que afecta a la tabla test4, veremos que lo único que hace es contar cuantos registros insertados contienen el valor 1 en a1, el valor 2, y así sucesivamente.

Para referirnos al registro que se va a insertar o modificar en un trigger BEFORE podemos usar NEW, y para referirnos al registro que se ha borrado o modificado en un trigger AFTER podemos usar OLD.

También podemos alterar un registro que se va a insertar o modificar. Por ejemplo, podemos hacer cosas como esta:

```
CREATE TRIGGER sdata_insert BEFORE INSERT ON `sometable`
FOR EACH ROW
BEGIN
SET NEW.guid = UUID();
END;
```

Sin embargo, esto solo está permitido cuando el trigger es de tipo BEFORE. Si lo usamos en uno AFTER obtendremos un error.

Un punto importante a tener en cuenta es el uso de triggers BEFORE con tablas InnoDB. Si hemos definido restricciones, puede ser que determinados INSERT fallen debido a dichas restricciones, pero los triggers sí que se activarán !! Por eso es recomendable usar triggers de tipo AFTER siempre que sea posible.

Borrar triggers

Para borrar un trigger existente ejecutaremos DROP TRIGGER. Es necesarios especificar la tabla a la que está asociado:

```
DROP TRIGGER nombre_tabla.nombre_trigger
```

12. Transacciones

Las transacciones son un mecanismo SQL para asegurar que un grupo de instrucciones se ejecutan de forma atómica. Eso quiere decir que el sistema nos asegura que o se ejecutan todas o ninguna, pero no puede ser que se ejecute una parte. Esto es especialmente importante para mantener la integridad de nuestros datos en situaciones de caídas de sistema o de acceso concurrente.

Imaginemos una base de datos de un banco en que existe una tabla `cuenta` en la que se almacenan los datos sobre el dinero de los usuario, y hay una columna `balance` que contiene el saldo de la cuenta. Imaginemos que queremos hacer una transferencia de 500 euros de una cuenta a otra. Simplificando, las instrucciones SQL que necesitaríamos serían:

```
UPDATE cuenta SET balance = balance + 500 WHERE cuenta.id = 1;
UPDATE cuenta SET balance = balance - 500 WHERE cuenta.id = 2;
```

En principio este código es correcto y hará lo que queremos. Pero qué pasa si después de la primera instrucción hay una caída del sistema ? La cuenta 1 tendrá 500 euros más, pero la segunda seguirá como antes. Como la transferencia no se ha terminado, el sistema, probablemente, volverá a intentar hacerla. Se vuelven a a ejecutar las dos instrucciones y esta vez todo sale bien. Qué ha ocurrido ? Que la cuenta 1 tiene 500 euros más que le llegaron en la primera transferencia que no terminó.

En este caso lo que queremos es que las dos instrucciones se ejecuten en una transacción y que el sistema nos asegure que o se ejecutan las do o ninguna. Para ello haremos lo siguiente;

```
START TRANSACTION;
UPDATE cuenta SET balance = balance + 500 WHERE cuenta.id = 1;
UPDATE cuenta SET balance = balance - 500 WHERE cuenta.id = 2;
COMMIT;
```

Esto nos asegura que o las dos cuentas se actualizan o ninguna. Es más, si otro usuario accede de forma concurrente al sistema y consulta los saldos de las cuentas 1 y 2, nunca obtendrá un resultado que corresponda a un estado intermedio de la transacción. Es decir, o verá las dos cuentas sin actualizar, o verá las dos actualizadas.

Una transacción ha de cumplir cuatro reglas:

- Atomicidad. O se ejecuta todo, o nada
- Consistencia. Durante la transacción se pueden romper restricciones de integridad, pero al acabar se deben restaurar.
- Aislamiento. Los datos que se actualizan en un transacción no están visibles para otras transacciones hasta que esta acaba.
- Durabilidad. Una vez la transacción ha sido cerrada no se puede deshacer.

Si durante una transacción comprobamos que no podemos seguir y que hay que anular todo lo que se ha hecho desde el principio de dicha transacción, usaremos la instrucción ROLLBACK.

AUTOCOMMIT

MySQL considera cada instrucción aislada como una transacción, de manera que al terminar sus efectos son permanentes. Este comportamiento se puede cambiar con:

```
SET AUTOCOMMIT = 0;
```

De esta manera las instrucciones SQL que enviemos al servidor no tendrán efecto hasta que hagamos COMMIT.

Más información sobre transacciones en:

http://dev.mysql.com/doc/refman/5.1/en/commit.html

13. Respaldos (backups)

Los respaldos (backups) son una de las tareas más importante dentro de la administración de una base de datos.

La manera más simple de hacer un respaldo con MySQL es usar el comando `mysqldump`. Este retorna un fichero con instrucciones SQL para generar las tablas de la base de datos y rellenarlas con ls información que contenía en el momento de ejecutar el `mysqldump`. Este sistema es lento, pero ofrece el máximo de compatibilidad a la hora de hacer una migración. Para reconstruir la base de datos se usa el comando `mysql`.

Si se usan tablas MyISAM hay un sistema más rápido y seguro que es copiar los directorios de la base de datos. Esto siempre hay que hacerlo con el servidor parado. Para hacer esto se puede usar el comando `mysqlhotcopy`. Para reconstruir la base de datos, simplemente hay que copiar los directorios donde corresponda.

Otra opción es crear ficheros de logs con los cambios que se van produciendo en la base de datos, y de esta manera tener un respaldo incremental. Sin embargo, si el volumen de cambios en la base de datos es muy alto, puede suponer un problema de espacio en el disco.

Generando respaldos

El comando `mysqldump` genera un fichero de instrucciones SQL que son capaces de, posteriormente, volver a generar la misma base de datos que teníamos. Normalmente, este fichero contiene un CREATE TABLE por cada tabla de la base de datos, instrucciones para crear índices, y numerosos INSERT para llenar las tablas con datos (uno por registro).

Por ejemplo, esto es lo que genera mysqldump para la base de datos biblioteca:

```
-- MySQL dump 10.13  Distrib 5.1.23-rc, for redhat-linux-gnu (i686)
--
-- Host: 127.0.0.1    Database: biblioteca
-- ------------------------------------------------------
-- Server version       5.1.23-rc

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `autores`
--

DROP TABLE IF EXISTS `autores`;
SET @saved_cs_client     = @@character_set_client;
SET character_set_client = utf8;
CREATE TABLE `autores` (
  `autorID` int(11) NOT NULL AUTO_INCREMENT,
```

```
  `nombreAutor` varchar(60) COLLATE utf8_spanish_ci NOT NULL DEFAULT '',
  `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`autorID`),
  KEY `nombreAutor` (`nombreAutor`)
) ENGINE=InnoDB AUTO_INCREMENT=90 DEFAULT CHARSET=utf8
COLLATE=utf8_spanish_ci;
SET character_set_client = @saved_cs_client;

--
-- Dumping data for table `autores`
--

LOCK TABLES `autores` WRITE;
/*!40000 ALTER TABLE `autores` DISABLE KEYS */;
INSERT INTO `autores` VALUES (1,'Perez Miquel','2008-04-03 16:48:25'),
(2,'Puig David','2008-04-03 16:48:25'),(3,'Vila Robert',
'2008-04-03 16:48:25'),(4,'Lopez Dan','2008-04-03 16:48:25'),(5,'Martin
Josep','2008-04-03 16:48:25'),(6,'Alvarez Tobias','200
8-04-03 16:48:25'),(7,'Costa Pau','2008-04-03 16:48:25'),(12,'Gil
Carles','2008-04-03 16:48:25'),(13,'Garcia Jordi','2008-04-0
3 16:48:25'),(14,'Reig Tomas','2008-04-03 16:48:25'),(15,'Ortega
Narcis','2008-04-03 16:48:25'),(16,'Molina Marc','2008-04-03
16:48:25'),(17,'Font Xavier','2008-04-03 16:48:25'),(20,'Abreu
Raul','2008-04-03 16:48:25'),(21,'Coma Miquel','2008-04-03 16:4

...

/*!40000 ALTER TABLE `titulos` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2008-04-19 15:08:57
```

Las líneas que comienzan por '--' se consideran comentarios en SQL. Además, al igual que en C, los caracteres '/*' inician un comentario. Pero, si van seguidas de un número n, entonces si la versión de MySQL que ejecute ese archivo, si su versión es mayor o igual que n entonces no será considerado comentario y se ejecutará su contenido. Eso es para prevenir errores de sintaxis cuando trabajamos con ficheros SQL en diferentes versiones de servidores.

Las instrucciones SET iniciales (que solo se ejecutarán en servidores con versions mayores o iguales a 4.1.1 y 4.0.14) guardan el juego de caracteres actual y lo actualizan a utf8. Además, se desactivan los test sobre claves foráneas y únicas para conseguir mayor eficiencia (se supone que la base de datos fue creada en un estado válido y esas comprobaciones no son necesarias al regenerarla).

A partir de entonces hay una serie de instrucciones por cada tabla. Lo primero es DROP TABLE que borra la tabla que estamos a punto de crear para que no se mezcle con datos ya existentes. Después se crea la tabla con CREATE TABLE y se rellena con INSERT. Al final se restaura el juego de caracteres que estaba activo al principio y se restauran otros estados.

La sintaxis de mysqldump es:

```
mysqldump [opcions] nombre_bd [tablas]  backup.sql
```

Si no se especifican tablas entonces se vuelcan todas. También se pueden volcar varias

bases de datos, o incluso todas:

```
mysqldump [opciones] –databases nombre_bd1 nombre_bd2 ...
mysqldump [opciones] –all-databases
```

Los detalles del respaldo se pueden controlar mediante un gran número de opciones. Para hacer un respaldo normal se pueden usar las opciones por defecto. Estas tienen los siguientes efectos:

- Durante el respaldo se ejecuta un bloqueo de lectura para cada tabla.
- El fichero resultante es el más pequeño posible.
- El fichero contiene un DROP TABLE por cada tabla para borrarla si ya existe cuando se restaura la base de datos.
- Todas las características de la base de datos se preservan, incluidos detalles específicos de MySQL que podrían dificultar la migración a otro sistema.
- El respaldo se crea usando utf8 y contiene las instrucciones necesarias para restaurar el juego de caracteres cuando la base de datos es restaurada.

Estas opciones no deben usarse si las tablas de la base de datos son InnoDB y las tablas pueden cambiar individualmente durante el respaldo, afectando a la integridad. Tampoco deberían usarse cuando se quiere el máximo de compatibilidad con otros sistemas.

Cuando necesitemos bloquear todas las tablas durante el respaldo para no perder la integridad se puede usar la opción `–lock-all-tables`.

Si la base de datos contiene tablas del tipo InnoDB hay que evitar el uso de las opciones por defecto usando `--skip-opt`, y en ese caso hay que especificar todas las opciones que necesitemos:

```
mysqldump -u root -p –skip-opt –single-transaction –ad-drop-table \
      --create-options –quick –extended-insert \
      --set-charset –disable-keys nombre_bd > backup.sql
```

Este comando también guardará las vistas correspondientes a las bases de datos. Si queremos que además también se vuelquen las funciones y los procedimientos almacenados necesitaremos usar la opción `–routines`.

Restaurar una base de datos

Si tenemos un fichero backup.sql que contiene una sola base de datos, la podemos restaurar así:

```
mysql -u root -p nombre_db < backup.sql
```

La base de datos `nombre_bd` debe existir. Si queremos restaurar varias bases de datos que están en un solo fichero, este contendrá los comandos CREATE DATABSE correspondientes. Simplemente hay que hacer:

```
mysql -u root -p < backup.sql
```

14. Administración de accesos y seguridad

Es normal que en una base de datos no toda la información sea accesible por todos sus usuarios. Por ello, MySQL tiene un sistema granular para controlar quień puede acceder a determinados datos, y qué tipo de acceso tendrá. MySQL tiene dos niveles de controlar el acceso a sus bases de datos. En un primer nivel, se determina si un usuario tiene permiso para establecer una conexión con el servidor. En un segundo nivel, se determina qué acciones tiene permitido realizar el usuario (SELECT, INSERT, DROP, ...) con cada base de datos, tabla o columna.

14.1 Tipos de conexiones

El primer punto de control es el momento en que el cliente solicita una conexión al servidor. Esto puede producirse a través de un script, usando el comando `mysql`, desde phpMyAdmin, ... etc. Hay varias maneras de que se produzca esta conexión.

Conexión remota

Cuando el cliente y el servidor se ejecutan en ordenadores diferentes la conexión se ha de realizar a través del protocolo TCP/IP. Tienen que darse dos condiciones:

- Los dos ordenadores deben ser accesibles vía TCP/IP
- EL puerto 3306, el que usa por defecto MySQL, no debe estar bloqueado por un cortafuegos.

Conexión local

Cuando el cliente y el servidor se ejecutan en el mismo ordenador hay varias posibilidades:

- TCP/IP: en este caso también se puede usar este protocolo.
- Socket (solo unix/linux): un socker permite la comunicación eficiente entre dos programas que se ejecutan en un sistema operativo unix/linux. Es el tipo de comunicación por defecto en unix/linux.
- Pipes (solo Windows 2000/XP): es el equivalente a sockets de unix/linux. Sin embargo, esta opción está desactivada por defecto. Además, el cliente debe soportar esta variante, lo cual es bastante raro, por lo que este sistema de comunicación se usa poco.
- Memoria compartida (solo Windows/XP): es otra alternativa a la variante de sockets de unix/linux. En este caso, cliente y servidor comparten una región de memoria para comunicación. Sin embargo, este método tampoco es muy usado ya que solo funciona si la opción `shared_memory` está activada en la configuración del servidor.

14.2 Administración de accesos

En una base de datos puede haber muchos y muy diferentes niveles de accesos para los diferentes tipos de usuarios. MySQL ofrece un sistema muy granular para controlar el tipo

de acceso.

Cambiar los privilegios

Hay varias maneras de cambiar los privilegios:

- La más simple es usar un programa de administración con interfície gráfica (p.e. MySQL Administrator o phpMyAdmin).
- Usando directamente el comando `mysql`.
- Usando las instrucciones SQL GRANT y REVOKE.
- Usando el script Perl `mysql_setpermission.pl`.

Nombre de usuario, password y nombre de host

La primara fase de acceso, la conexión del cliente al servidor, depende de tres elementos:

- Nombre de usuario. Es el nombre con el que el cliente se identifica delante del servidor. Los usuarios de MySQL no tienen ninguna conexión con los usuarios del sistema operativo, aunque normalmente coinciden.
- Password. Ocurre lo mismo que con los nombre de usuario, no tienen conexión con el password del sistema operativo, y son almacenados y gestionados independientemente por MySQL.
- Host. Al establecer la conexión hay que especificar el ordenador en el que se ejecuta el servidor. Se puede dar como una dirección IP o con un nombre. AL recibir una conexión, el servidor usa la dirección del ordenador cliente para determinar los derechos de acceso ya que un mismo usuario puede tener permiso para conectarse solo desde determinadas direcciones.

14.3 Creación de bases de datos, usuarios y sus privilegios

Para crear una base de datos y dar privilegio de acceso a ella a un determinado usuario de forma local haremos:

```
CREATE DATABASE nombre_bd
GRANT ALL ON nombre_bd.* TO nombre_usuario@localhost IDENTIFIED BY 'xxx'
```

En este caso se otorgan todos los privilegios (ALL) sobre toda la base de datos (nombre_bd.*) a un usuario que se conecta localmente (nombre_usuario@localhost) usando el password 'xxx'.

Sin embargo, podemos dar unos privilegios más delimitados con, por ejemplo:

```
GRANT Select, Insert, Update, Delete ON nombre_bd.*
```

Esto sería el acceso básico a una base de datos. SI queremos ampliarlo con permisos más específicos:

```
GRANT Lock Tables, CreateTemporary Tables, Execute ON nombre_bd.*
```

Con esto permitimos al usuario bloquear tablas, crear tablas temporales, y ejecutar

procedimientos almacenados.

Es importante remarcar que al identificar al usuario con `nombre_usuario@localhost` estamos restringiendo su acceso al modo local con sockets. Si queremos proporcionar acceso local por TCP/IP debemos usar `nombre_usuario@nombre_ordenador`.

Crear nuevos usuarios

La sintaxis para crear un nuevo usuario es:

```
CREATE USER nombre_usuario IDENTIFIED BY 'password'
```

Cambiar privilegios

Hasta ahora hemos visto algunos ejemplos de como otorgar privilegios, pero la sintaxis completa es:

```
GRANT privilegios
ON [nombre_bd.]nombre_tabla o nombre_bd.nombre_procedimiento
TO usuario@host [IDENTIFIED BY 'password']
[WITH GRANT OPTION]
```

Y para retirar privilegios:

```
REVOKE privilegios
ON [nombre_bd.]nombre_tabla o nombre_bd.nombre_procedimiento
FROM usuario@host
```

14.4 Base de datos 'mysql'

Toda la información sobre los privilegios en un servidor MySQL están almacenados en una base de datos especial llamada `mysql`. Contiene una gran cantidad de tablas con información sobre el sistema. De ellas, hay 6 que guardan la información sobre los privilegios. La Tabla 9 describe cuales son y qué información contienen.

Nombre	Significado
`user`	Controla qué usuarios pueden conectarse al servidor y desde qué ordenador. Esta tabla también contiene privilegios globales.
`db`	Especifica qué usuarios pueden acceder a qué bases de datos
`host`	Extiende la tabla `db` con información de los ordenadores que tienen acceso.
`tables_priv`	Especifica quien puede acceder a las tablas de una base de datos
`columns_priv`	Especifica quien puede acceder a las columnas de una tabla
`func`	Permite la gestión de funciones definidas por el usuario (*use-defined functions*). Está sin documentar.
`procs_priv`	Especifica quien puede ejecutar procedimientos almacenados individuales.

Tabla 9: Descripción de las tablas `mysql` *que almacenan la información sobre privilegio*