MySQL por ejemplos para principiantes

Lea " Cómo instalar MySQL y comenzar " sobre cómo instalar, personalizar y comenzar con MySQL.

1. Resumen de los comandos MySQL utilizados en este tutorial

Para obtener una sintaxis detallada, consulte el manual de MySQL "Sintaxis de declaración SQL" @ http://dev.mysql.com/doc/refman/5.5/en/sql-syntax.html .

```
-- Nivel de base de datos
DROP DATABASE nombre de base de datos                  -- Eliminar la base de datos (¡irrecuperable!)
DROP DATABASE IF EXISTS nombre de base de datos        -- Eliminar si existe
CREAR BASE DE DATOS nombre de base de datos -- Crear una nueva base de datos
CREAR BASE DE DATOS SI NO EXISTE nombre de base de datos  -- Crear solo si existe no existe
MOSTRAR BASES DE DATOS                                 - Mostrar todas las bases de datos en este servidor
USE nombre de base de datos                            - Establecer la base de datos predeterminada (actual)
SELECT DATABASE()                                - Mostrar la base de datos predeterminada
MOSTRAR CREAR BASE DE DATOS nombre de base de datos           - Mostrar la instrucción CREATE DATABASE

-
TABLA DROP a nivel de tabla [SI EXISTE] nombre de tabla , ...
CREAR TABLA [SI NO EXISTE] nombreTabla (
    nombreColumnaTipoColumnaAtributoColumna , ...
   CLAVE PRIMARIA ( nombre de columna ),
   CLAVE EXTRANJERA ( columnaNmae ) REFERENCIAS nombretabla ( columnaNmae )
)
SHOW TABLES                 - Muestra todas las tablas en la base de datos predeterminada
DESCRIBE|DESC tableName     - Describe los detalles de una tabla
ALTER TABLE tableName ...   - Modifica una tabla, por ejemplo, ADD COLUMN y DROP COLUMN
ALTER TABLE tableName ADD columnDefinition
ALTER TABLE Nombre de tabla DROP Nombre de columna
ALTER TABLE Nombre de tabla AGREGAR CLAVE EXTRANJERA ( Nmae columna ) REFERENCIAS Nombre de tabla ( Nmae columna )
ALTER TABLE Nombre de tabla DROP LLAVE EXTRANJERA Nombre de restricción
SHOW CREATE TABLE nombre de tabla         - Muestra la instrucción CREATE TABLE para este nombre de tabla

- Nivel de fila
INSERT INTO tableName
   VALUES ( columna1Value , column2Value ,...)               - Insertar en todas las columnas
INSERT INTO tableName
   VALUES ( columna1Value , column2Value ,...), ...          - Insertar varias filas
INSERT INTO tableName ( NombreColumna1 , ..., NombreColumnaN )
   VALORES ( columna1Valor , ..., columnaNValue )              - Insertar en las columnas seleccionadas
ELIMINAR DE nombreTabla DONDE criterios
ACTUALIZAR nombreTabla SET nombreColumna = expr , ... DONDE criterios
SELECCIONAR * | NombreColumna1 AS alias1 , ..., NombreColumna AS aliasN
   FROM NombreTabla
   DONDE criterios
   GROUP BY NombreColumna
   ORDENAR POR NombreColumna  ASC |DESC,...
   TENER restricciones de grupo
   LÍMITE de recuento | recuento de compensación
```

ÍNDICE DE CONTENIDOS (OCULTAR)

```
-- Otros
MUESTRAN ADVERTENCIAS;    -- Mostrar las advertencias de la declaración anterior.
```

2. Un ejemplo para principiantes (pero NO para principiantes)

A MySQL database server contains many databases (or schemas). Each database consists of one or more tables. A table is made up of columns (or fields) and rows (records).

The SQL keywords and commands are NOT case-sensitive. For clarity, they are shown in uppercase. The *names* or *identifiers* (database names, table names, column names, etc.) are case-sensitive in some systems, but not in other systems. Hence, it is best to treat *identifiers* as case-sensitive.

SHOW DATABASES

You can use `SHOW DATABASES` to list all the existing databases in the server.

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
........
```

The databases "`mysql`", "`information_schema`" and "`performance_schema`" are system databases used internally by MySQL. A "`test`" database is provided during installation for your testing.

Let us begin with a simple example - a *product sales database*. A product sales database typically consists of many tables, e.g., products, customers, suppliers, orders, payments, employees, among others. Let's call our database "`southwind`" (inspired from Microsoft's Northwind Trader sample database). We shall begin with the first table called "`products`" with the following columns (having data types as indicated) and rows:

```
Database: southwind
Table: products

productID   productCode      name       quantity      price
   INT        CHAR(3)     VARCHAR(30)     INT      DECIMAL(10,2)

1001         PEN            Pen Red        5000        1.23
1002         PEN            Pen Blue       8000        1.25
1003         PEN            Pen Black      2000        1.25
1004         PEC            Pencil 2B      10000       0.48
1005         PEC            Pencil 2H      8000        0.49
```

2.1 Creating and Deleting a Database - CREATE DATABASE and DROP DATABASE

You can create a new database using SQL command "`CREATE DATABASE` *`databaseName`*"; and delete a database using "`DROP DATABASE` *`databaseName`*". You could optionally apply condition "`IF EXISTS`" or "`IF NOT EXISTS`" to these commands. For example,

```
mysql> CREATE DATABASE southwind;
Query OK, 1 row affected (0.03 sec)

mysql> DROP DATABASE southwind;
Query OK, 0 rows affected (0.11 sec)

mysql> CREATE DATABASE IF NOT EXISTS southwind;
Query OK, 1 row affected (0.01 sec)

mysql> DROP DATABASE IF EXISTS southwind;
Query OK, 0 rows affected (0.00 sec)
```

IMPORTANT: Use SQL `DROP` (and `DELETE`) commands with extreme care, as the deleted entities are irrecoverable. **THERE IS NO UNDO!!!**

SHOW CREATE DATABASE

The CREATE DATABASE commands uses some defaults. You can issue a "SHOW CREATE DATABASE *databaseName*" to display the full command and check these default values. We use \G (instead of ';') to display the results vertically. (Try comparing the outputs produced by ';' and \G.)

```
mysql> CREATE DATABASE IF NOT EXISTS southwind;

mysql> SHOW CREATE DATABASE southwind \G
*************************** 1. row ***************************
       Database: southwind
Create Database: CREATE DATABASE `southwind` /*!40100 DEFAULT CHARACTER SET latin1 */
```

Back-Quoted Identifiers (`name`)

Unquoted names or identifiers (such as database name, table name and column name) cannot contain blank and special characters, or crash with MySQL keywords (such as ORDER and DESC). You can include blanks and special characters or use MySQL keyword as identifier by enclosing it with a pair of back-quote, in the form of `*name*`.

For robustness, the SHOW command back-quotes all the identifiers, as illustrated in the above example.

Comments and Version Comments

MySQL *multi-line comments* are enclosed within /* and */; *end-of-line comments* begins with -- (followed by a space) or #.

The /*!40100 ..... */ is known as *version comment*, which will only be run if the server is at or above this version number 4.01.00. To check the version of your MySQL server, issue query "SELECT version()".

2.2 Setting the Default Database - USE

The command "USE *databaseName*" sets a particular database as the default (or current) database. You can reference a table in the default database using *tableName* directly. But you need to use the fully-qualified *databaseName.tableName* to reference a table NOT in the default database.

In our example, we have a database named "southwind" with a table named "products". If we issue "USE southwind" to set southwind as the default database, we can simply call the table as "products". Otherwise, we need to reference the table as "southwind.products".

To display the current default database, issue command "SELECT DATABASE()".

2.3 Creating and Deleting a Table - CREATE TABLE and DROP TABLE

You can create a new table *in the default database* using command "CREATE TABLE *tableName*" and "DROP TABLE *tableName*". You can also apply condition "IF EXISTS" or "IF NOT EXISTS". To create a table, you need to define all its columns, by providing the columns' *name*, *type*, and *attributes*.

Let's create a table "products" in our database "southwind".

```
-- Remove the database "southwind", if it exists.
-- Beware that DROP (and DELETE) actions are irreversible and not recoverable!
mysql> DROP DATABASE IF EXISTS southwind;
Query OK, 1 rows affected (0.31 sec)

-- Create the database "southwind"
mysql> CREATE DATABASE southwind;
Query OK, 1 row affected (0.01 sec)

-- Show all the databases in the server
--   to confirm that "southwind" database has been created.
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| southwind          |
| ......             |

-- Set "southwind" as the default database so as to reference its table directly.
mysql> USE southwind;
Database changed

-- Show the current (default) database
mysql> SELECT DATABASE();
+------------+
```

```
| DATABASE() |
+------------+
| southwind  |
+------------+

-- Show all the tables in the current database.
-- "southwind" has no table (empty set).
mysql> SHOW TABLES;
Empty set (0.00 sec)

-- Create the table "products". Read "explanations" below for the column definitions
mysql> CREATE TABLE IF NOT EXISTS products (
         productID    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
         productCode  CHAR(3)       NOT NULL DEFAULT '',
         name         VARCHAR(30)   NOT NULL DEFAULT '',
         quantity     INT UNSIGNED  NOT NULL DEFAULT 0,
         price        DECIMAL(7,2)  NOT NULL DEFAULT 99999.99,
         PRIMARY KEY  (productID)
       );
Query OK, 0 rows affected (0.08 sec)

-- Show all the tables to confirm that the "products" table has been created
mysql> SHOW TABLES;
+---------------------+
| Tables_in_southwind |
+---------------------+
| products            |
+---------------------+

-- Describe the fields (columns) of the "products" table
mysql> DESCRIBE products;
+-------------+------------------+------+-----+------------+----------------+
| Field       | Type             | Null | Key | Default    | Extra          |
+-------------+------------------+------+-----+------------+----------------+
| productID   | int(10) unsigned | NO   | PRI | NULL       | auto_increment |
| productCode | char(3)          | NO   |     |            |                |
| name        | varchar(30)      | NO   |     |            |                |
| quantity    | int(10) unsigned | NO   |     | 0          |                |
| price       | decimal(7,2)     | NO   |     | 99999.99   |                |
+-------------+------------------+------+-----+------------+----------------+

-- Show the complete CREATE TABLE statement used by MySQL to create this table
mysql> SHOW CREATE TABLE products \G
*************************** 1. row ***************************
       Table: products
Create Table:
CREATE TABLE `products` (
  `productID`    int(10) unsigned  NOT NULL AUTO_INCREMENT,
  `productCode`  char(3)           NOT NULL DEFAULT '',
  `name`         varchar(30)       NOT NULL DEFAULT '',
  `quantity`     int(10) unsigned  NOT NULL DEFAULT '0',
  `price`        decimal(7,2)      NOT NULL DEFAULT '99999.99',
  PRIMARY KEY (`productID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Explanations

We define 5 columns in the table `products`: `productID`, `productCode`, `name`, `quantity` and `price`. The types are:

- `productID` is `INT UNSIGNED` - non-negative integers.
- `productCode` is `CHAR(3)` - a fixed-length alphanumeric string of 3 characters.
- `name` is `VARCHAR(30)` - a variable-length string of up to 30 characters.
  We use fixed-length string for `productCode`, as we assume that the `productCode` contains *exactly* 3 characters. On the other hand, we use variable-length string for `name`, as its length varies - `VARCHAR` is more efficient than `CHAR`.
- `quantity` is also `INT UNSIGNED` (non-negative integers).
- `price` is `DECIMAL(10,2)` - a decimal number with 2 decimal places.
  `DECIMAL` is *precise* (represented as integer with a fix decimal point). On the other hand, `FLOAT` and `DOUBLE` (real numbers) are not precise and are approximated. `DECIMAL` type is recommended for currency.

The attribute "`NOT NULL`" specifies that the column cannot contain the `NULL` value. `NULL` is a special value indicating "no value", "unknown value" or "missing value". In our case, these columns shall have a proper value. We also set the default value of the columns. The column will take on its default value, if no value is specified during the record creation.

We set the column `productID` as the so-called *primary key*. Values of the primary-key column must be unique. Every table shall contain a primary key. This ensures that every row can be distinguished from other rows. You can specify a single column or a set of columns (e.g., `firstName` and `lastName`) as the primary key. An *index* is build automatically on the primary-key column to facilitate fast search. Primary key is also used as reference by other tables.

We set the column `productID` to `AUTO_INCREMENT`. with default starting value of 1. When you insert a row with `NULL` (recommended) (or 0, or a missing value) for the `AUTO_INCREMENT` column, the maximum value of that column plus 1 would be inserted. You can also insert a valid value to an `AUTO_INCREMENT` column, bypassing the auto-increment.

2.4 Inserting Rows - `INSERT INTO`

Let's fill up our "`products`" table with rows. We set the `productID` of the first record to 1001, and use `AUTO_INCREMENT` for the rest of records by inserting a `NULL`, or with a missing column value. Take note that strings must be enclosed with a pair of single quotes (or double quotes).

```
-- Insert a row with all the column values
mysql> INSERT INTO products VALUES (1001, 'PEN', 'Pen Red', 5000, 1.23);
Query OK, 1 row affected (0.04 sec)

-- Insert multiple rows in one command
-- Inserting NULL to the auto_increment column results in max_value + 1
mysql> INSERT INTO products VALUES
         (NULL, 'PEN', 'Pen Blue',  8000, 1.25),
         (NULL, 'PEN', 'Pen Black', 2000, 1.25);
Query OK, 2 rows affected (0.03 sec)
Records: 2  Duplicates: 0  Warnings: 0

-- Insert value to selected columns
-- Missing value for the auto_increment column also results in max_value + 1
mysql> INSERT INTO products (productCode, name, quantity, price) VALUES
         ('PEC', 'Pencil 2B', 10000, 0.48),
         ('PEC', 'Pencil 2H', 8000, 0.49);
Query OK, 2 row affected (0.03 sec)

-- Missing columns get their default values
mysql> INSERT INTO products (productCode, name) VALUES ('PEC', 'Pencil HB');
Query OK, 1 row affected (0.04 sec)

-- 2nd column (productCode) is defined to be NOT NULL
mysql> INSERT INTO products values (NULL, NULL, NULL, NULL, NULL);
ERROR 1048 (23000): Column 'productCode' cannot be null

-- Query the table
mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+------------+
| productID | productCode | name      | quantity | price      |
+-----------+-------------+-----------+----------+------------+
|      1001 | PEN         | Pen Red   |     5000 |       1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |       1.25 |
|      1003 | PEN         | Pen Black |     2000 |       1.25 |
|      1004 | PEC         | Pencil 2B |    10000 |       0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |       0.49 |
|      1006 | PEC         | Pencil HB |        0 | 9999999.99 |
+-----------+-------------+-----------+----------+------------+
6 rows in set (0.02 sec)

-- Remove the last row
mysql> DELETE FROM products WHERE productID = 1006;
```

`INSERT INTO` Syntax

We can use the `INSERT INTO` statement to insert a new row with all the column values, using the following syntax:

```
INSERT INTO tableName VALUES (firstColumnValue, ..., lastColumnValue)  -- All columns
```

You need to list the values in the same order in which the columns are defined in the `CREATE TABLE`, separated by commas. For columns of string data type (`CHAR`, `VARCHAR`), enclosed the value with a pair of single quotes (or double quotes). For columns of numeric data type (`INT`, `DECIMAL`, `FLOAT`, `DOUBLE`), simply place the number.

You can also insert multiple rows in one `INSERT INTO` statement:

```
INSERT INTO tableName VALUES
   (row1FirstColumnValue, ..., row1lastColumnValue),
   (row2FirstColumnValue, ..., row2lastColumnValue),
   ...
```

To insert a row with values on selected columns only, use:

```
-- Insert single record with selected columns
INSERT INTO tableName (column1Name, ..., columnNName) VALUES (column1Value, ..., columnNValue)
-- Alternately, use SET to set the values
INSERT INTO tableName SET column1=value1, column2=value2, ...

-- Insert multiple records
INSERT INTO tableName
   (column1Name, ..., columnNName)
VALUES
   (row1column1Value, ..., row2ColumnNValue),
   (row2column1Value, ..., row2ColumnNValue),
   ...
```

The remaining columns will receive their default value, such as `AUTO_INCREMENT`, default, or `NULL`.

2.5 Querying the Database - SELECT

The most common, important and complex task is to query a database for a subset of data that meets your needs - with the `SELECT` command. The `SELECT` command has the following syntax:

```
-- List all the rows of the specified columns
SELECT column1Name, column2Name, ... FROM tableName

-- List all the rows of ALL columns, * is a wildcard denoting all columns
SELECT * FROM tableName

-- List rows that meet the specified criteria in WHERE clause
SELECT column1Name, column2Name,... FROM tableName WHERE criteria
SELECT * FROM tableName WHERE criteria
```

For examples,

```
-- List all rows for the specified columns
mysql> SELECT name, price FROM products;
+-----------+-------+
| name      | price |
+-----------+-------+
| Pen Red   |  1.23 |
| Pen Blue  |  1.25 |
| Pen Black |  1.25 |
| Pencil 2B |  0.48 |
| Pencil 2H |  0.49 |
+-----------+-------+
5 rows in set (0.00 sec)

-- List all rows of ALL the columns. The wildcard * denotes ALL columns
mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
+-----------+-------------+-----------+----------+-------+
5 rows in set (0.00 sec)
```

SELECT without Table

You can also issue SELECT without a table. For example, you can SELECT an expression or evaluate a built-in function.

```
mysql> SELECT 1+1;
+-----+
| 1+1 |
+-----+
|   2 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2012-10-24 22:13:29 |
+---------------------+
1 row in set (0.00 sec)

// Multiple columns
mysql> SELECT 1+1, NOW();
+-----+---------------------+
| 1+1 | NOW()               |
+-----+---------------------+
|   2 | 2012-10-24 22:16:34 |
+-----+---------------------+
1 row in set (0.00 sec)
```

Comparison Operators

For numbers (INT, DECIMAL, FLOAT), you could use comparison operators: '=' (equal to), '<>' or '!=' (not equal to), '>' (greater than), '<' (less than), '>=' (greater than or equal to), '<=' (less than or equal to), to compare two numbers. For example, price > 1.0, quantity <= 500.

```
mysql> SELECT name, price FROM products WHERE price < 1.0;
+-----------+-------+
| name      | price |
+-----------+-------+
| Pencil 2B |  0.48 |
| Pencil 2H |  0.49 |
+-----------+-------+
2 rows in set (0.00 sec)

mysql> SELECT name, quantity FROM products WHERE quantity <= 2000;
+-----------+----------+
| name      | quantity |
+-----------+----------+
| Pen Black |     2000 |
+-----------+----------+
1 row in set (0.00 sec)
```

CAUTION: Do not compare FLOATs (real numbers) for equality ('=' or '<>'), as they are not precise. On the other hand, DECIMAL are precise.

For strings, you could also use '=', '<>', '>', '<', '>=', '<=' to compare two strings (e.g., productCode = 'PEC'). The ordering of string depends on the so-called *collation* chosen. For example,

```
mysql> SELECT name, price FROM products WHERE productCode = 'PEN';
                                       -- String values are quoted
+-----------+-------+
| name      | price |
+-----------+-------+
| Pen Red   |  1.23 |
| Pen Blue  |  1.25 |
| Pen Black |  1.25 |
+-----------+-------+
3 rows in set (0.00 sec)
```

String Pattern Matching - LIKE and NOT LIKE

For strings, in addition to full matching using operators like `'='` and `'<>'`, we can perform *pattern matching* using operator `LIKE` (or `NOT LIKE`) with wildcard characters. The wildcard `'_'` matches any single character; `'%'` matches any number of characters (including zero). For example,

- `'abc%'` matches strings beginning with `'abc'`;
- `'%xyz'` matches strings ending with `'xyz'`;
- `'%aaa%'` matches strings containing `'aaa'`;
- `'___'` matches strings containing exactly three characters; and
- `'a_b%'` matches strings beginning with `'a'`, followed by any single character, followed by `'b'`, followed by zero or more characters.

```
-- "name" begins with 'PENCIL'
mysql> SELECT name, price FROM products WHERE name LIKE 'PENCIL%';
+-----------+-------+
| name      | price |
+-----------+-------+
| Pencil 2B |  0.48 |
| Pencil 2H |  0.49 |
+-----------+-------+

-- "name" begins with 'P', followed by any two characters,
--   followed by space, followed by zero or more characters
mysql> SELECT name, price FROM products WHERE name LIKE 'P__ %';
+-----------+-------+
| name      | price |
+-----------+-------+
| Pen Red   |  1.23 |
| Pen Blue  |  1.25 |
| Pen Black |  1.25 |
+-----------+-------+
```

MySQL also support regular expression matching via the `REGEX` operator.

Arithmetic Operators

You can perform arithmetic operations on numeric fields using arithmetic operators, as tabulated below:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
DIV	Integer Division
%	Modulus (Remainder)

Logical Operators - AND, OR, NOT, XOR

You can combine multiple conditions with boolean operators `AND`, `OR`, `XOR`. You can also invert a condition using operator `NOT`. For examples,

```
mysql> SELECT * FROM products WHERE quantity >= 5000 AND name LIKE 'Pen %';
+-----------+-------------+----------+----------+-------+
| productID | productCode | name     | quantity | price |
+-----------+-------------+----------+----------+-------+
|      1001 | PEN         | Pen Red  |     5000 |  1.23 |
|      1002 | PEN         | Pen Blue |     8000 |  1.25 |
+-----------+-------------+----------+----------+-------+

mysql> SELECT * FROM products WHERE quantity >= 5000 AND price < 1.24 AND name LIKE 'Pen %';
+-----------+-------------+---------+----------+-------+
| productID | productCode | name    | quantity | price |
+-----------+-------------+---------+----------+-------+
|      1001 | PEN         | Pen Red |     5000 |  1.23 |
+-----------+-------------+---------+----------+-------+

mysql> SELECT * FROM products WHERE NOT (quantity >= 5000 AND name LIKE 'Pen %');
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
```

```
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
+-----------+-------------+-----------+----------+-------+
```

IN, NOT IN

You can select from members of a set with `IN` (or `NOT IN`) operator. This is easier and clearer than the equivalent `AND` - `OR` expression.

```
mysql> SELECT * FROM products WHERE name IN ('Pen Red', 'Pen Black');
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
+-----------+-------------+-----------+----------+-------+
```

BETWEEN, NOT BETWEEN

To check if the value is within a range, you could use `BETWEEN ... AND ...` operator. Again, this is easier and clearer than the equivalent `AND` - `OR` expression.

```
mysql> SELECT * FROM products
       WHERE (price BETWEEN 1.0 AND 2.0) AND (quantity BETWEEN 1000 AND 2000);
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
+-----------+-------------+-----------+----------+-------+
```

IS NULL, IS NOT NULL

`NULL` is a special value, which represent "no value", "missing value" or "unknown value". You can checking if a column contains `NULL` by `IS NULL` or `IS NOT NULL`. For example,

```
mysql> SELECT * FROM products WHERE productCode IS NULL;
Empty set (0.00 sec)
```

Using comparison operator (such as = or <>) to check for `NULL` is a *mistake* - a very *common mistake*. For example,

```
SELECT * FROM products WHERE productCode = NULL;
-- This is a common mistake. NULL cannot be compared.
```

ORDER BY Clause

You can order the rows selected using `ORDER BY` clause, with the following syntax:

```
SELECT ... FROM tableName
WHERE criteria
ORDER BY columnA ASC|DESC, columnB ASC|DESC, ...
```

The selected row will be ordered according to the values in *`columnA`*, in either ascending (`ASC`) (default) or descending (`DESC`) order. If several rows have the same value in *`columnA`*, it will be ordered according to *`columnB`*, and so on. For strings, the ordering could be case-sensitive or case-insensitive, depending on the so-called character collating sequence used. For examples,

```
-- Order the results by price in descending order
mysql> SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
+-----------+-------------+-----------+----------+-------+

-- Order by price in descending order, followed by quantity in ascending (default) order
mysql> SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC, quantity;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
```

```
+-----------+-------------+-----------+----------+-------+
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
+-----------+-------------+-----------+----------+-------+
```

You can randomize the returned records via function `RAND()`, e.g.,

```
mysql> SELECT * FROM products ORDER BY RAND();
```

LIMIT Clause

A `SELECT` query on a large database may produce many rows. You could use the `LIMIT` clause to limit the number of rows displayed, e.g.,

```
-- Display the first two rows
mysql> SELECT * FROM products ORDER BY price LIMIT 2;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
+-----------+-------------+-----------+----------+-------+
```

To continue to the following records , you could specify the number of rows to be skipped, followed by the number of rows to be displayed in the `LIMIT` clause, as follows:

```
-- Skip the first two rows and display the next 1 row
mysql> SELECT * FROM products ORDER BY price LIMIT 2, 1;
+-----------+-------------+---------+----------+-------+
| productID | productCode | name    | quantity | price |
+-----------+-------------+---------+----------+-------+
|      1001 | PEN         | Pen Red |     5000 |  1.23 |
+-----------+-------------+---------+----------+-------+
```

AS - Alias

You could use the keyword `AS` to define an *alias* for an identifier (such as column name, table name). The alias will be used in displaying the name. It can also be used as reference. For example,

```
mysql> SELECT productID AS ID, productCode AS Code,
              name AS Description, price AS `Unit Price`  -- Define aliases to be used as display names
       FROM products
       ORDER BY ID;  -- Use alias ID as reference
+------+------+-------------+------------+
| ID   | Code | Description | Unit Price |
+------+------+-------------+------------+
| 1001 | PEN  | Pen Red     |       1.23 |
| 1002 | PEN  | Pen Blue    |       1.25 |
| 1003 | PEN  | Pen Black   |       1.25 |
| 1004 | PEC  | Pencil 2B   |       0.48 |
| 1005 | PEC  | Pencil 2H   |       0.49 |
+------+------+-------------+------------+
```

Take note that the identifier "`Unit Price`" contains a blank and must be *back-quoted*.

Function CONCAT()

You can also concatenate a few columns as one (e.g., joining the last name and first name) using function `CONCAT()`. For example,

```
mysql> SELECT CONCAT(productCode, ' - ', name) AS `Product Description`, price FROM products;
+---------------------+-------+
| Product Description | price |
+---------------------+-------+
| PEN - Pen Red       |  1.23 |
| PEN - Pen Blue      |  1.25 |
| PEN - Pen Black     |  1.25 |
| PEC - Pencil 2B     |  0.48 |
| PEC - Pencil 2H     |  0.49 |
+---------------------+-------+
```

2.6 Producing Summary Reports

To produce a summary report, we often need to *aggregate related rows*.

DISTINCT

A column may have duplicate values, we could use keyword `DISTINCT` to select only distinct values. We can also apply `DISTINCT` to several columns to select distinct combinations of these columns. For examples,

```
-- Without DISTINCT
mysql> SELECT price FROM products;
+-------+
| price |
+-------+
|  1.23 |
|  1.25 |
|  1.25 |
|  0.48 |
|  0.49 |
+-------+

-- With DISTINCT on price
mysql> SELECT DISTINCT price AS `Distinct Price` FROM products;
+----------------+
| Distinct Price |
+----------------+
|           1.23 |
|           1.25 |
|           0.48 |
|           0.49 |
+----------------+

-- DISTINCT combination of price and name
mysql> SELECT DISTINCT price, name FROM products;
+-------+-----------+
| price | name      |
+-------+-----------+
|  1.23 | Pen Red   |
|  1.25 | Pen Blue  |
|  1.25 | Pen Black |
|  0.48 | Pencil 2B |
|  0.49 | Pencil 2H |
+-------+-----------+
```

GROUP BY Clause

The `GROUP BY` clause allows you to *collapse* multiple records with a common value into groups. For example,

```
mysql> SELECT * FROM products ORDER BY productCode, productID;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
+-----------+-------------+-----------+----------+-------+

mysql> SELECT * FROM products GROUP BY productCode;
       -- Only first record in each group is shown
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
+-----------+-------------+-----------+----------+-------+
```

`GROUP BY` by itself is not meaningful. It is used together with `GROUP BY` aggregate functions (such as `COUNT()`, `AVG()`, `SUM()`) to produce group summary.

GROUP BY Aggregate Functions: COUNT, MAX, MIN, AVG, SUM, STD, GROUP_CONCAT

We can apply `GROUP BY` Aggregate functions to each group to produce group summary report.

The function `COUNT(*)` returns the rows selected; `COUNT(columnName)` counts only the non-`NULL` values of the given column. For example,

```
-- Function COUNT(*) returns the number of rows selected
mysql> SELECT COUNT(*) AS `Count` FROM products;
       -- All rows without GROUP BY clause
+-------+
| Count |
+-------+
|     5 |
+-------+

mysql> SELECT productCode, COUNT(*) FROM products GROUP BY productCode;
+-------------+----------+
| productCode | COUNT(*) |
+-------------+----------+
| PEC         |        2 |
| PEN         |        3 |
+-------------+----------+

-- Order by COUNT - need to define an alias to be used as reference
mysql> SELECT productCode, COUNT(*) AS count
       FROM products
       GROUP BY productCode
       ORDER BY count DESC;
+-------------+-------+
| productCode | count |
+-------------+-------+
| PEN         |     3 |
| PEC         |     2 |
+-------------+-------+
```

Besides `COUNT()`, there are many other `GROUP BY` aggregate functions such as `AVG()`, `MAX()`, `MIN()` and `SUM()`. For example,

```
mysql> SELECT MAX(price), MIN(price), AVG(price), STD(price), SUM(quantity)
       FROM products;
       -- Without GROUP BY - All rows
+------------+------------+------------+------------+---------------+
| MAX(price) | MIN(price) | AVG(price) | STD(price) | SUM(quantity) |
+------------+------------+------------+------------+---------------+
|       1.25 |       0.48 |   0.940000 |   0.371591 |         33000 |
+------------+------------+------------+------------+---------------+

mysql> SELECT productCode, MAX(price) AS `Highest Price`, MIN(price) AS `Lowest Price`
       FROM products
       GROUP BY productCode;
+-------------+---------------+--------------+
| productCode | Highest Price | Lowest Price |
+-------------+---------------+--------------+
| PEC         |          0.49 |         0.48 |
| PEN         |          1.25 |         1.23 |
+-------------+---------------+--------------+

mysql> SELECT productCode, MAX(price), MIN(price),
              CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`,
              CAST(STD(price) AS DECIMAL(7,2)) AS `Std Dev`,
              SUM(quantity)
       FROM products
       GROUP BY productCode;
       -- Use CAST(... AS ...) function to format floating-point numbers
+-------------+------------+------------+---------+---------+---------------+
| productCode | MAX(price) | MIN(price) | Average | Std Dev | SUM(quantity) |
+-------------+------------+------------+---------+---------+---------------+
| PEC         |       0.49 |       0.48 |    0.49 |    0.01 |         18000 |
```

```
| PEN         |       1.25 |       1.23 |    1.24 |    0.01 |         15000 |
+-------------+------------+------------+---------+---------+---------------+
```

HAVING clause

HAVING is similar to WHERE, but it can operate on the GROUP BY aggregate functions; whereas WHERE operates only on columns.

```
mysql> SELECT
          productCode AS `Product Code`,
          COUNT(*) AS `Count`,
          CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`
       FROM products
       GROUP BY productCode
       HAVING Count >=3;
          -- CANNOT use WHERE count >= 3
+--------------+-------+---------+
| Product Code | Count | Average |
+--------------+-------+---------+
| PEN          |     3 |    1.24 |
+--------------+-------+---------+
```

WITH ROLLUP

The WITH ROLLUP clause shows the *summary of group summary*, e.g.,

```
mysql> SELECT
          productCode,
          MAX(price),
          MIN(price),
          CAST(AVG(price) AS DECIMAL(7,2)) AS `Average`,
          SUM(quantity)
       FROM products
       GROUP BY productCode
       WITH ROLLUP;        -- Apply aggregate functions to all groups
+-------------+------------+------------+---------+---------------+
| productCode | MAX(price) | MIN(price) | Average | SUM(quantity) |
+-------------+------------+------------+---------+---------------+
| PEC         |       0.49 |       0.48 |    0.49 |         18000 |
| PEN         |       1.25 |       1.23 |    1.24 |         15000 |
| NULL        |       1.25 |       0.48 |    0.94 |         33000 |
+-------------+------------+------------+---------+---------------+
```

2.7 Modifying Data - UPDATE

To modify existing data, use UPDATE ... SET command, with the following syntax:

```
UPDATE tableName SET columnName = {value|NULL|DEFAULT}, ... WHERE criteria
```

For example,

```
-- Increase the price by 10% for all products
mysql> UPDATE products SET price = price * 1.1;

mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     5000 |  1.35 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.38 |
|      1003 | PEN         | Pen Black |     2000 |  1.38 |
|      1004 | PEC         | Pencil 2B |    10000 |  0.53 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.54 |
+-----------+-------------+-----------+----------+-------+

-- Modify selected rows
mysql> UPDATE products SET quantity = quantity - 100 WHERE name = 'Pen Red';

mysql> SELECT * FROM products WHERE name = 'Pen Red';
+-----------+-------------+---------+----------+-------+
```

```
| productID | productCode | name    | quantity | price |
+-----------+-------------+---------+----------+-------+
|      1001 | PEN         | Pen Red |     4900 |  1.35 |
+-----------+-------------+---------+----------+-------+

-- You can modify more than one values
mysql> UPDATE products SET quantity = quantity + 50, price = 1.23 WHERE name = 'Pen Red';

mysql> SELECT * FROM products WHERE name = 'Pen Red';
+-----------+-------------+---------+----------+-------+
| productID | productCode | name    | quantity | price |
+-----------+-------------+---------+----------+-------+
|      1001 | PEN         | Pen Red |     4950 |  1.23 |
+-----------+-------------+---------+----------+-------+
```

CAUTION: If the WHERE clause is omitted in the UPDATE command, ALL ROWS will be updated. Hence, it is a good practice to issue a SELECT query, using the same criteria, to check the result set before issuing the UPDATE. This also applies to the DELETE statement in the following section.

2.8 Deleting Rows - DELETE FROM

Use the DELETE FROM command to delete row(s) from a table, with the following syntax:

```
-- Delete all rows from the table. Use with extreme care! Records are NOT recoverable!!!
DELETE FROM tableName
-- Delete only row(s) that meets the criteria
DELETE FROM tableName WHERE criteria
```

For example,

```
mysql> DELETE FROM products WHERE name LIKE 'Pencil%';
Query OK, 2 row affected (0.00 sec)

mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     4950 |  1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.38 |
|      1003 | PEN         | Pen Black |     2000 |  1.38 |
+-----------+-------------+-----------+----------+-------+

-- Use this with extreme care, as the deleted records are irrecoverable!
mysql> DELETE FROM products;
Query OK, 3 rows affected (0.00 sec)

mysql> SELECT * FROM products;
Empty set (0.00 sec)
```

Beware that "DELETE FROM *tableName*" without a WHERE clause deletes ALL records from the table. Even with a WHERE clause, you might have deleted some records unintentionally. It is always advisable to issue a SELECT command with the same WHERE clause to check the result set before issuing the DELETE (and UPDATE).

2.9 Loading/Exporting Data from/to a Text File

There are several ways to add data into the database: (a) manually issue the INSERT commands; (b) run the INSERT commands from a script; or (c) load raw data from a file using LOAD DATA or via `mysqlimport` utility.

LOAD DATA LOCAL INFILE ... INTO TABLE ...

Besides using INSERT commands to insert rows, you could keep your *raw data* in a text file, and load them into the table via the LOAD DATA command. For example, use a text editor to CREATE a NEW FILE called "`products_in.csv`", under "`d:\myProject`" (for Windows) or "`Documents`" (for Mac), containing the following records, where the values are separated by '`,`'. The file extension of "`.csv`" stands for *Comma-Separated Values* text file.

```
\N,PEC,Pencil 3B,500,0.52
\N,PEC,Pencil 4B,200,0.62
\N,PEC,Pencil 5B,100,0.73
\N,PEC,Pencil 6B,500,0.47
```

You can load the raw data into the `products` table as follows:

```
(For Windows)
-- Need to use forward-slash (instead of back-slash) as directory separator
mysql> LOAD DATA LOCAL INFILE 'd:/myProject/products_in.csv' INTO TABLE products
         COLUMNS TERMINATED BY ','
         LINES TERMINATED BY '\r\n';

(For Macs)
mysql> LOAD DATA LOCAL INFILE '~/Documents/products_in.csv' INTO TABLE products
         COLUMNS TERMINATED BY ',';
```

```
mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1007 | PEC         | Pencil 3B |      500 |  0.52 |
|      1008 | PEC         | Pencil 4B |      200 |  0.62 |
|      1009 | PEC         | Pencil 5B |      100 |  0.73 |
|      1010 | PEC         | Pencil 6B |      500 |  0.47 |
+-----------+-------------+-----------+----------+-------+
```

Notes:

- You need to provide the path (absolute or relative) and the filename. Use Unix-style forward-slash `'/'` as the directory separator, instead of Windows-style back-slash `'\'`.
- The default line delimiter (or end-of-line) is `'\n'` (Unix-style). If the text file is prepared in Windows, you need to include `LINES TERMINATED BY '\r\n'`.
- The default column delimiter is "tab" (in a so-called TSV file - Tab-Separated Values). If you use another delimiter, e.g. `','`, include `COLUMNS TERMINATED BY ','`.
- You need to use `\N` for `NULL`.

`mysqlimport` Utility Program

You can also use the `mysqlimport` utility program to load data from a text file.

```
-- SYNTAX
> mysqlimport -u username -p --local databaseName tableName.tsv
   -- The raw data must be kept in a TSV (Tab-Separated Values) file with filename the same as tablename

-- EXAMPLES
-- Create a new file called "products.tsv" containing the following record,
--  and saved under "d:\myProject" (for Windows) or "Documents" (for Mac)
-- The values are separated by tab (not spaces).
\N  PEC  Pencil 3B  500  0.52
\N  PEC  Pencil 4B  200  0.62
\N  PEC  Pencil 5B  100  0.73
\N  PEC  Pencil 6B  500  0.47

(For Windows)
> cd path-to-mysql-bin
> mysqlimport -u root -p --local southwind d:/myProject/products.tsv

(For Macs)
$ cd /usr/local/mysql/bin
$ ./mysqlimport -u root -p --local southwind ~/Documents/products.tsv
```

`SELECT ... INTO OUTFILE ...`

Complimenting `LOAD DATA` command, you can use `SELECT ... INTO OUTFILE fileName FROM tableName` to export data from a table to a text file. For example,

```
(For Windows)
mysql> SELECT * FROM products INTO OUTFILE 'd:/myProject/products_out.csv'
         COLUMNS TERMINATED BY ','
         LINES TERMINATED BY '\r\n';

(For Macs)
```

```
mysql> SELECT * FROM products INTO OUTFILE '~/Documents/products_out.csv'
         COLUMNS TERMINATED BY ',';
```

2.10 Running a SQL Script

Instead of manually entering each of the SQL statements, you can keep many SQL statements in a text file, called SQL script, and run the script. For example, use a programming text editor to prepare the following script and save as "`load_products.sql`" under "`d:\myProject`" (for Windows) or "`Documents`" (for Mac).

```
DELETE FROM products;
INSERT INTO products VALUES (2001, 'PEC', 'Pencil 3B', 500, 0.52),
                            (NULL, 'PEC', 'Pencil 4B', 200, 0.62),
                            (NULL, 'PEC', 'Pencil 5B', 100, 0.73),
                            (NULL, 'PEC', 'Pencil 6B', 500, 0.47);
SELECT * FROM products;
```

You can run the script either:

1. via the "`source`" command in a MySQL client. For example, to restore the `southwind` backup earlier:

   ```
   (For Windows)
   mysql> source d:/myProject/load_products.sql
      -- Use Unix-style forward slash (/) as directory separator

   (For Macs)
   mysql> source ~/Documents/load_products.sql
   ```

2. via the "batch mode" of the `mysql` client program, by re-directing the input from the script:

   ```
   (For Windows)
   > cd path-to-mysql-bin
   > mysql -u root -p southwind < d:\myProject\load_products.sql

   (For Macs)
   $ cd /usr/local/mysql/bin
   $ ./mysql -u root -p southwind < ~\Documents\load_products.sql
   ```

3. More Than One Tables

Our example so far involves only one table "`products`". A practical database contains many *related* tables.

Products have suppliers. If each product has one supplier, and each supplier supplies only one product (known as *one-to-one relationship*), we can simply add the supplier's data (name, address, phone number) into the `products` table. Suppose that each product has one supplier, and a supplier may supply zero or more products (known as *one-to-many* relationship). Putting the supplier's data into the `products` table results in duplication of data. This is because one supplier may supply many products, hence, the same supplier's data appear in many rows. This not only wastes the storage but also easily leads to inconsistency (as all duplicate data must be updated simultaneously). The situation is even more complicated if one product has many suppliers, and each supplier can supply many products, in a *many-to-many* relationship.

3.1 One-To-Many Relationship

Suppose that each product has one supplier, and each supplier supplies one or more products. We could create a table called `suppliers` to store suppliers' data (e.g., name, address and phone number). We create a column with unique value called `supplierID` to identify every suppliers. We set `supplierID` as the *primary key* for the table `suppliers` (to ensure uniqueness and facilitate fast search).

To relate the `suppliers` table to the `products` table, we add a new column into the `products` table - the `supplierID`. We then set the `supplierID` column of the `products` table as a foreign key references the `supplierID` column of the `suppliers` table to ensure the so-called *referential integrity*.

```
Database: southwind
Table: suppliers

supplierID      name        phone
   INT       VARCHAR(3)   CHAR(8)

501           ABC Traders    88881111
```

502	XYZ Company	88882222
503	QQ Corp	88883333

Database: southwind
Table: products

productID INT	productCode CHAR(3)	name VARCHAR(30)	quantity INT	price DECIMAL(10,2)	supplierID INT (Foreign Key)
2001	PEC	Pencil 3B	500	0.52	**501**
2002	PEC	Pencil 4B	200	0.62	**501**
2003	PEC	Pencil 5B	100	0.73	**501**
2004	PEC	Pencil 6B	500	0.47	**502**

We need to first create the `suppliers` table, because the `products` table references the `suppliers` table. The `suppliers` table is known as the *parent* table; while the `products` table is known as the *child* table in this relationship.

```
mysql> USE southwind;

mysql> DROP TABLE IF EXISTS suppliers;

mysql> CREATE TABLE suppliers (
         supplierID  INT UNSIGNED  NOT NULL AUTO_INCREMENT,
         name        VARCHAR(30)   NOT NULL DEFAULT '',
         phone       CHAR(8)       NOT NULL DEFAULT '',
         PRIMARY KEY (supplierID)
       );

mysql> DESCRIBE suppliers;
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| supplierID | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| name       | varchar(30)      | NO   |     |         |                |
| phone      | char(8)          | NO   |     |         |                |
+------------+------------------+------+-----+---------+----------------+

mysql> INSERT INTO suppliers VALUE
          (501, 'ABC Traders', '88881111'),
          (502, 'XYZ Company', '88882222'),
          (503, 'QQ Corp', '88883333');

mysql> SELECT * FROM suppliers;
+------------+-------------+----------+
| supplierID | name        | phone    |
+------------+-------------+----------+
|        501 | ABC Traders | 88881111 |
|        502 | XYZ Company | 88882222 |
|        503 | QQ Corp     | 88883333 |
+------------+-------------+----------+
```

ALTER TABLE

Instead of deleting and re-creating the `products` table, we shall use "`ALTER TABLE`" to add a new column `supplierID` into the `products` table.

```
mysql> ALTER TABLE products
       ADD COLUMN supplierID INT UNSIGNED NOT NULL;
Query OK, 4 rows affected (0.13 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> DESCRIBE products;
+-------------+------------------+------+-----+------------+----------------+
| Field       | Type             | Null | Key | Default    | Extra          |
+-------------+------------------+------+-----+------------+----------------+
| productID   | int(10) unsigned | NO   | PRI | NULL       | auto_increment |
```

```
| productCode | char(3)          | NO   |     |            |                |
| name        | varchar(30)      | NO   |     |            |                |
| quantity    | int(10) unsigned | NO   |     | 0          |                |
| price       | decimal(10,2)    | NO   |     | 9999999.99 |                |
| supplierID  | int(10) unsigned | NO   |     | NULL       |                |
+-------------+------------------+------+-----+------------+----------------+
```

Next, we shall add a *foreign key constraint* on the `supplierID` columns of the `products` child table to the `suppliers` parent table, to ensure that every `supplierID` in the `products` table always refers to a *valid* `supplierID` in the `suppliers` table - this is called *referential integrity*.

Before we can add the foreign key, we need to set the `supplierID` of the existing records in the `products` table to a valid `supplierID` in the `suppliers` table (say `supplierID=501`).

```
-- Set the supplierID of the existing records in "products" table to a VALID supplierID
--   of "suppliers" table
mysql> UPDATE products SET supplierID = 501;

-- Add a foreign key constrain
mysql> ALTER TABLE products
       ADD FOREIGN KEY (supplierID) REFERENCES suppliers (supplierID);

mysql> DESCRIBE products;
+-------------+------------------+------+-----+------------+----------------+
| Field       | Type             | Null | Key | Default    | Extra          |
+-------------+------------------+------+-----+------------+----------------+
  ......
| supplierID  | int(10) unsigned | NO   | MUL |            |                |
+-------------+------------------+------+-----+------------+----------------+

mysql> UPDATE products SET supplierID = 502 WHERE productID  = 2004;
  -- Choose a valid productID

mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+------------+
| productID | productCode | name      | quantity | price | supplierID |
+-----------+-------------+-----------+----------+-------+------------+
|      2001 | PEC         | Pencil 3B |      500 |  0.52 |        501 |
|      2002 | PEC         | Pencil 4B |      200 |  0.62 |        501 |
|      2003 | PEC         | Pencil 5B |      100 |  0.73 |        501 |
|      2004 | PEC         | Pencil 6B |      500 |  0.47 |        502 |
+-----------+-------------+-----------+----------+-------+------------+
```

SELECT with JOIN

`SELECT` command can be used to query and join data from two related tables. For example, to list the product's `name` (in `products` table) and supplier's `name` (in `suppliers` table), we could join the two table via the two common `supplierID` columns:

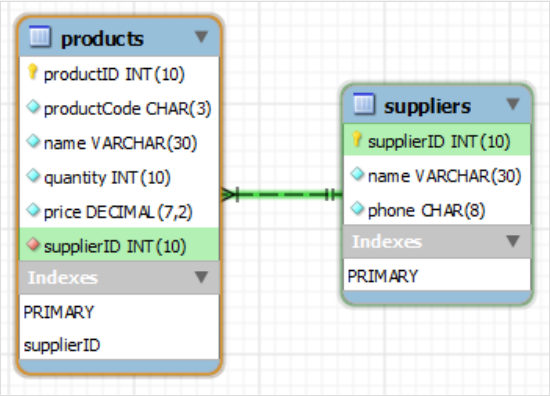
```
-- ANSI style: JOIN ... ON ...
mysql> SELECT products.name, price, suppliers.name
       FROM products
          JOIN suppliers ON products.supplierID = suppliers.supplierID
       WHERE price < 0.6;
+-----------+-------+-------------+
| name      | price | name        |
+-----------+-------+-------------+
| Pencil 3B |  0.52 | ABC Traders |
| Pencil 6B |  0.47 | XYZ Company |
+-----------+-------+-------------+
    -- Need to use products.name and suppliers.name to differentiate the two "names"

-- Join via WHERE clause (legacy and not recommended)
mysql> SELECT products.name, price, suppliers.name
       FROM products, suppliers
       WHERE products.supplierID = suppliers.supplierID
          AND price < 0.6;
+-----------+-------+-------------+
| name      | price | name        |
+-----------+-------+-------------+
| Pencil 3B |  0.52 | ABC Traders |
```

```
| Pencil 6B |  0.47 | XYZ Company |
+-----------+-------+-------------+
```

In the above query result, two of the columns have the same heading "name". We could create *aliases* for headings.

```
-- Use aliases for column names for display
mysql> SELECT products.name AS `Product Name`, price, suppliers.name AS `Supplier Name`
       FROM products
          JOIN suppliers ON products.supplierID = suppliers.supplierID
       WHERE price < 0.6;
+--------------+-------+---------------+
| Product Name | price | Supplier Name |
+--------------+-------+---------------+
| Pencil 3B    |  0.52 | ABC Traders   |
| Pencil 6B    |  0.47 | XYZ Company   |
+--------------+-------+---------------+

-- Use aliases for table names too
mysql> SELECT p.name AS `Product Name`, p.price, s.name AS `Supplier Name`
       FROM products AS p
          JOIN suppliers AS s ON p.supplierID = s.supplierID
       WHERE p.price < 0.6;
```

The database diagram is as illustrated. The link indicates a one-to-many relationship between products and suppliers.



3.2 Many-To-Many Relationship

Suppose that a product has many suppliers; and a supplier supplies many products in a so-called many-to-many relationship. The above solution breaks. You cannot include the supplierID in the products table, as you cannot determine the number of suppliers, and hence, the number of columns needed for the supplierIDs. Similarly, you cannot include the productID in the suppliers table, as you cannot determine the number of products.

To resolve this problem, you need to create a new table, known as a *junction table* (or *joint table*), to provide the linkage. Let's call the junction table products_suppliers, as illustrated.

```
Database: southwind
Table: products_suppliers

   productID        supplierID
      INT              INT
 (Foreign Key)    (Foreign Key)
2001              501
2002              501
2003              501
2004              502
2001              503


Database: southwind
Table: suppliers
```

supplierID	name	phone
INT	VARCHAR(30)	CHAR(8)
501	ABC Traders	88881111
502	XYZ Company	88882222
503	QQ Corp	88883333

Database: southwind
Table: products

productID	productCode	name	quantity	price
INT	CHAR(3)	VARCHAR(30)	INT	DECIMAL(10,2)
2001	PEC	Pencil 3B	500	0.52
2002	PEC	Pencil 4B	200	0.62
2003	PEC	Pencil 5B	100	0.73
2004	PEC	Pencil 6B	500	0.47

Let's create the `products_suppliers` table. The primary key of the table consists of two columns: `productID` and `supplierID`, as their combination uniquely identifies each rows. This primary key is defined to ensure uniqueness. Two foreign keys are defined to set the constraint to the two parent tables.

```
mysql> CREATE TABLE products_suppliers (
         productID   INT UNSIGNED  NOT NULL,
         supplierID  INT UNSIGNED  NOT NULL,
                     -- Same data types as the parent tables
         PRIMARY KEY (productID, supplierID),
                     -- uniqueness
         FOREIGN KEY (productID)  REFERENCES products  (productID),
         FOREIGN KEY (supplierID) REFERENCES suppliers (supplierID)
       );

mysql> DESCRIBE products_suppliers;
+------------+------------------+------+-----+---------+-------+
| Field      | Type             | Null | Key | Default | Extra |
+------------+------------------+------+-----+---------+-------+
| productID  | int(10) unsigned | NO   | PRI | NULL    |       |
| supplierID | int(10) unsigned | NO   | PRI | NULL    |       |
+------------+------------------+------+-----+---------+-------+

mysql> INSERT INTO products_suppliers VALUES (2001, 501), (2002, 501),
       (2003, 501), (2004, 502), (2001, 503);
-- Values in the foreign-key columns (of the child table) must match
--   valid values in the columns they reference (of the parent table)

mysql> SELECT * FROM products_suppliers;
+-----------+------------+
| productID | supplierID |
+-----------+------------+
|      2001 |        501 |
|      2002 |        501 |
|      2003 |        501 |
|      2004 |        502 |
|      2001 |        503 |
+-----------+------------+
```

Next, remove the `supplierID` column from the `products` table. (This column was added to establish the one-to-many relationship. It is no longer needed in the many-to-many relationship.)

Before this column can be removed, you need to remove the foreign key that builds on this column. To remove a key in MySQL, you need to know its constraint name, which was generated by the system. To find the constraint name, issue a "`SHOW CREATE TABLE products`" and take note of the foreign key's constraint name in the clause "`CONSTRAINT constraint_name FOREIGN KEY ....`". You can then drop the foreign key using "`ALTER TABLE products DROP FOREIGN KEY` *`constraint_name`*"

```
mysql> SHOW CREATE TABLE products \G
Create Table: CREATE TABLE `products` (
```

```
  `productID`   int(10) unsigned  NOT NULL AUTO_INCREMENT,
  `productCode` char(3)           NOT NULL DEFAULT '',
  `name`        varchar(30)       NOT NULL DEFAULT '',
  `quantity`    int(10) unsigned  NOT NULL DEFAULT '0',
  `price`       decimal(7,2)      NOT NULL DEFAULT '99999.99',
  `supplierID`  int(10) unsigned   NOT NULL DEFAULT '501',
  PRIMARY KEY (`productID`),
  KEY `supplierID` (`supplierID`),
  CONSTRAINT `products_ibfk_1` FOREIGN KEY (`supplierID`)
     REFERENCES `suppliers` (`supplierID`)
) ENGINE=InnoDB AUTO_INCREMENT=1006 DEFAULT CHARSET=latin1

mysql> ALTER TABLE products DROP FOREIGN KEY products_ibfk_1;

mysql> SHOW CREATE TABLE products \G
```

Now, we can remove the column redundant `supplierID` column.

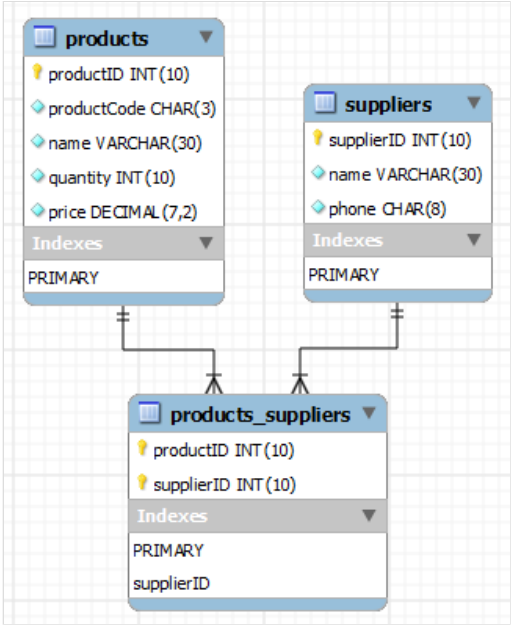
```
mysql> ALTER TABLE products DROP supplierID;

mysql> DESC products;
```

Querying

Similarly, we can use `SELECT` with `JOIN` to query data from the 3 tables, for examples,

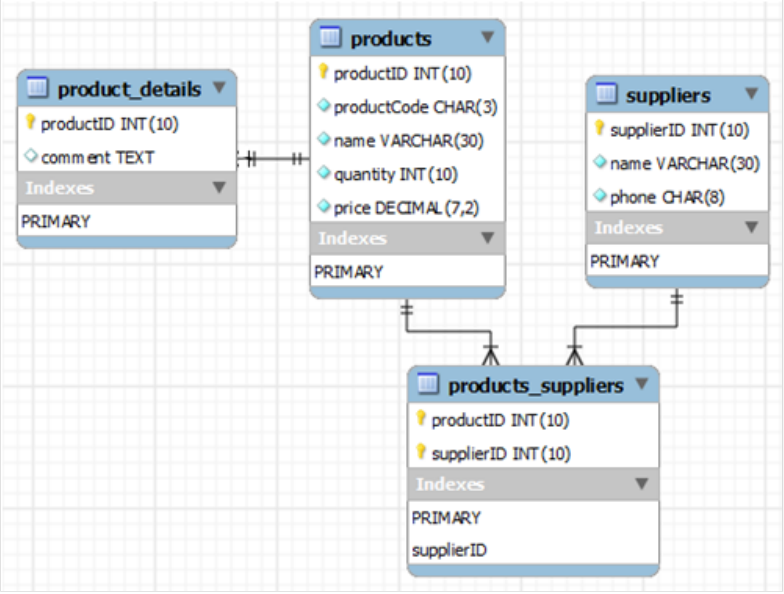
```
mysql> SELECT products.name AS `Product Name`, price, suppliers.name AS `Supplier Name`
       FROM products_suppliers
          JOIN products  ON products_suppliers.productID = products.productID
          JOIN suppliers ON products_suppliers.supplierID = suppliers.supplierID
       WHERE price < 0.6;
+--------------+-------+---------------+
| Product Name | price | Supplier Name |
+--------------+-------+---------------+
| Pencil 3B    |  0.52 | ABC Traders   |
| Pencil 3B    |  0.52 | QQ Corp       |
| Pencil 6B    |  0.47 | XYZ Company   |
+--------------+-------+---------------+

-- Define aliases for tablenames too
mysql> SELECT p.name AS `Product Name`, s.name AS `Supplier Name`
       FROM products_suppliers AS ps
          JOIN products AS p ON ps.productID = p.productID
          JOIN suppliers AS s ON ps.supplierID = s.supplierID
       WHERE p.name = 'Pencil 3B';
+--------------+---------------+
| Product Name | Supplier Name |
+--------------+---------------+
| Pencil 3B    | ABC Traders   |
| Pencil 3B    | QQ Corp       |
+--------------+---------------+

-- Using WHERE clause to join (legacy and not recommended)
mysql> SELECT p.name AS `Product Name`, s.name AS `Supplier Name`
       FROM products AS p, products_suppliers AS ps, suppliers AS s
       WHERE p.productID = ps.productID
          AND ps.supplierID = s.supplierID
          AND s.name = 'ABC Traders';
+--------------+---------------+
| Product Name | Supplier Name |
+--------------+---------------+
| Pencil 3B    | ABC Traders   |
| Pencil 4B    | ABC Traders   |
| Pencil 5B    | ABC Traders   |
+--------------+---------------+
```

The database diagram is as follows. Both `products` and `suppliers` tables exhibit a one-to-many relationship to the junction table. The many-to-many relationship is supported via the junction table.



3.3 One-to-one Relationship

Suppose that some products have *optional* data (e.g., photo, comment). Instead of keeping these optional data in the `products` table, it is more efficient to create another table called `product_details`, and link it to `products` with a *one-to-one relationship*, as illustrated.




```
mysql> CREATE TABLE product_details (
          productID  INT UNSIGNED   NOT NULL,
                     -- same data type as the parent table
          comment    TEXT  NULL,
                     -- up to 64KB
          PRIMARY KEY (productID),
          FOREIGN KEY (productID) REFERENCES products (productID)
       );

mysql> DESCRIBE product_details;
+-----------+------------------+------+-----+---------+-------+
| Field     | Type             | Null | Key | Default | Extra |
+-----------+------------------+------+-----+---------+-------+
| productID | int(10) unsigned | NO   | PRI | NULL    |       |
| comment   | text             | YES  |     | NULL    |       |
+-----------+------------------+------+-----+---------+-------+

mysql> SHOW CREATE TABLE product_details \G
*************************** 1. row ***************************
```

```
       Table: product_details
Create Table: CREATE TABLE `product_details` (
  `productID`  int(10) unsigned  NOT NULL,
  `comment`    text,
  PRIMARY KEY (`productID`),
  CONSTRAINT `product_details_ibfk_1` FOREIGN KEY (`productID`) REFERENCES `products` (`productID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

3.4 Backup and Restore

Backup: Before we conclude this example, let's run the `mysqldump` utility program to dump out (backup) the entire `southwind` database.

```
(For Windows)
-- Start a NEW "cmd"
> cd path-to-mysql-bin
> mysqldump -u root -p --databases southwind > "d:\myProject\backup_southwind.sql"

(For Macs)
-- Start a NEW "terminal"
$ cd /usr/local/mysql/bin
$ ./mysqldump -u root -p --databases southwind > ~/Documents/backup_southwind.sql
```

Study the output file, which contains `CREATE DATABASE`, `CREATE TABLE` and `INSERT` statements to re-create the tables dumped.

The SYNTAX for the `mysqldump` utility program is as follows:

```
-- Dump selected databases with --databases option
> mysqldump -u username -p --databases database1Name [database2Name ...] > backupFile.sql
-- Dump all databases in the server with --all-databases option, except mysql.user table (for security)
> mysqldump -u root -p --all-databases --ignore-table=mysql.user > backupServer.sql

-- Dump all the tables of a particular database
> mysqldump -u username -p databaseName > backupFile.sql
-- Dump selected tables of a particular database
> mysqldump -u username -p databaseName table1Name [table2Name ...] > backupFile.sql
```

Restore: The utility `mysqldump` produces a SQL script (consisting of `CREATE TABLE` and `INSERT` commands to re-create the tables and loading their data). You can restore from the backup by running the script either:

1. via the "`source`" command in an interactive client. For example, to restore the `southwind` backup earlier:

   ```
   (For Windows)
   -- Start a MySQL client
   mysql> source d:/myProject/backup_southwind.sql
      -- Provide absolute or relative filename of the script
      -- Use Unix-style forward slash (/) as path separator

   (For Macs)
   -- Start a MySQL client
   mysql> source ~/Documents/backup_southwind.sql
   ```

2. via the "batch mode" of the `mysql` client program by re-directing the input from the script:

   ```
   (For Windows)
   -- Start a NEW "cmd"
   > cd path-to-mysql-bin
   > mysql -u root -p southwind < d:\myProject\backup_southwind.sql

   (For Macs)
   -- Start a NEW "terminal"
   $ cd /usr/local/mysql/bin
   $ ./mysql -u root -p southwind < ~/Documents/backup_southwind.sql
   ```

4. More on Primary Key, Foreign Key and Index

4.1 Primary Key

In the relational model, a table shall not contain duplicate rows, because that would create ambiguity in retrieval. To ensure uniqueness, each table should have a column (or a set of columns), called *primary key*, that uniquely identifies every record of the table. For example, an unique number `customerID` can be used as the primary key for the `customers` table; `productCode` for `products` table; `isbn` for `books` table. A primary key is called a *simple key* if it is a single column; it is called a *composite key* if it is made up of several columns. Most RDBMSs build an index on the primary key to facilitate fast search. The primary key is often used to relate to other tables.

4.2 Foreign Key

A *foreign key* of a child table is used to reference the parent table. *Foreign key constraint* can be imposed to ensure so-called *referential integrity* - values in the child table must be valid values in the parent table.

We define the foreign key when defining the child table, which references a parent table, as follows:

```
-- Child table definition
CREATE TABLE tableName (
   ......
   ......
   CONSTRAINT constraintName FOREIGN KEY (columnName) REFERENCES parentTableName (columnName)
     [ON DELETE RESTRICT | CASCADE | SET NULL | NO ACTION]   -- On DELETE reference action
     [ON UPDATE RESTRICT | CASCADE | SET NULL | NO ACTION]   -- On UPDATE reference action
)
```

You can specify the *reference action* for `UPDATE` and `DELETE` via the optional `ON UPDATE` and `ON DELETE` clauses:

1. `RESTRICT` (default): disallow `DELETE` or `UPDATE` of the parent's row, if there are matching rows in child table.
2. `CASCADE`: cascade the `DELETE` or `UPDATE` action to the matching rows in the child table.
3. `SET NULL`: set the foreign key value in the child table to `NULL` (if `NULL` is allowed).
4. `NO ACTION`: a SQL term which means no action on the parent's row. Same as `RESTRICT` in MySQL, which disallows `DELETE` or `UPDATE` (do nothing).

Try deleting a record in the `suppliers` (parent) table that is referenced by `products_suppliers` (child) table, e.g.,

```
mysql> SELECT * FROM products_suppliers;
+-----------+------------+
| productID | supplierID |
+-----------+------------+
|      2001 |        501 |
|      2002 |        501 |
|      2003 |        501 |
|      2004 |        502 |
|      2001 |        503 |
+-----------+------------+

-- Try deleting a row from parent table with matching rows in the child table
mysql> DELETE FROM suppliers WHERE supplierID = 501;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails
(`southwind`.`products_suppliers`, CONSTRAINT `products_suppliers_ibfk_2`
FOREIGN KEY (`supplierID`) REFERENCES `suppliers` (`supplierID`))
```

The record cannot be deleted as the default "`ON DELETE RESTRICT`" constraint was imposed.

4.3 Indexes (or Keys)

Indexes (or Keys) can be created on selected column(s) to facilitate *fast search*. Without index, a "`SELECT * FROM products WHERE productID=x`" needs to match with the `productID` column of all the records in the `products` table. If `productID` column is indexed (e.g., using a binary tree), the matching can be greatly improved (via the binary tree search).

You should index columns which are frequently used in the `WHERE` clause; and as `JOIN` columns.

The drawback about indexing is cost and space. Building and maintaining indexes require computations and memory spaces. Indexes facilitate fast search but deplete the performance on modifying the table (`INSERT/UPDATE/DELETE`), and need to be justified. Nevertheless, relational databases are typically optimized for queries and retrievals, but NOT for updates.

In MySQL, the keyword `KEY` is synonym to `INDEX`.

In MySQL, indexes can be built on:

1. a single column (column-index)

2. a set of columns (concatenated-index)
3. on unique-value column (`UNIQUE INDEX` or `UNIQUE KEY`)
4. on a prefix of a column for strings (`VARCHAR` or `CHAR`), e.g., first 5 characters.

There can be more than one indexes in a table. Index are automatically built on the primary-key column(s).

You can build index via `CREATE TABLE`, `CREATE INDEX` or `ALTER TABLE`.

```
CREATE TABLE tableName (
   ......
   [UNIQUE] INDEX|KEY indexName (columnName, ...),
      -- The optional keyword UNIQUE ensures that all values in this column are distinct
      -- KEY is synonym to INDEX
   ......
   PRIMARY KEY (columnName, ...)  -- Index automatically built on PRIMARY KEY column
);

CREATE [UNIQUE] INDEX indexName ON tableName(columnName, ...);

ALTER TABLE tableName ADD UNIQUE|INDEX|PRIMARY KEY indexName (columnName, ...)

SHOW INDEX FROM tableName;
```

Example

```
mysql> CREATE TABLE employees (
          emp_no      INT UNSIGNED   NOT NULL AUTO_INCREMENT,
          name        VARCHAR(50)    NOT NULL,
          gender      ENUM ('M','F') NOT NULL,
          birth_date  DATE           NOT NULL,
          hire_date   DATE           NOT NULL,
          PRIMARY KEY (emp_no)  -- Index built automatically on primary-key column
       );

mysql> DESCRIBE employees;
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| emp_no     | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| name       | varchar(50)      | NO   |     | NULL    |                |
| gender     | enum('M','F')    | NO   |     | NULL    |                |
| birth_date | date             | NO   |     | NULL    |                |
| hire_date  | date             | NO   |     | NULL    |                |
+------------+------------------+------+-----+---------+----------------+

mysql> SHOW INDEX FROM employees \G
*************************** 1. row ***************************
        Table: employees
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: emp_no
     .......

mysql> CREATE TABLE departments (
          dept_no    CHAR(4)      NOT NULL,
          dept_name  VARCHAR(40)  NOT NULL,
          PRIMARY KEY  (dept_no),   -- Index built automatically on primary-key column
          UNIQUE INDEX (dept_name)  -- Build INDEX on this unique-value column
       );

mysql> DESCRIBE departments;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| dept_no   | char(4)     | NO   | PRI | NULL    |       |
| dept_name | varchar(40) | NO   | UNI | NULL    |       |
+-----------+-------------+------+-----+---------+-------+

mysql> SHOW INDEX FROM departments \G
```

```
*************************** 1. row ***************************
        Table: departments
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: dept_no
      .......
*************************** 2. row ***************************
        Table: departments
   Non_unique: 0
     Key_name: dept_name
 Seq_in_index: 1
  Column_name: dept_name
      .......

-- Many-to-many junction table between employees and departments
mysql> CREATE TABLE dept_emp (
         emp_no     INT UNSIGNED  NOT NULL,
         dept_no    CHAR(4)       NOT NULL,
         from_date  DATE          NOT NULL,
         to_date    DATE          NOT NULL,
         INDEX       (emp_no),          -- Build INDEX on this non-unique-value column
         INDEX       (dept_no),         -- Build INDEX on this non-unique-value column
         FOREIGN KEY (emp_no)  REFERENCES employees (emp_no)
            ON DELETE CASCADE ON UPDATE CASCADE,
         FOREIGN KEY (dept_no) REFERENCES departments (dept_no)
            ON DELETE CASCADE ON UPDATE CASCADE,
         PRIMARY KEY (emp_no, dept_no)  -- Index built automatically
     );

mysql> DESCRIBE dept_emp;
+-----------+------------------+------+-----+---------+-------+
| Field     | Type             | Null | Key | Default | Extra |
+-----------+------------------+------+-----+---------+-------+
| emp_no    | int(10) unsigned | NO   | PRI | NULL    |       |
| dept_no   | char(4)          | NO   | PRI | NULL    |       |
| from_date | date             | NO   |     | NULL    |       |
| to_date   | date             | NO   |     | NULL    |       |
+-----------+------------------+------+-----+---------+-------+

mysql> SHOW INDEX FROM dept_emp \G
*************************** 1. row ***************************
        Table: dept_emp
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: emp_no
      ........
*************************** 2. row ***************************
        Table: dept_emp
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 2
  Column_name: dept_no
      ........
*************************** 3. row ***************************
        Table: dept_emp
   Non_unique: 1
     Key_name: emp_no
 Seq_in_index: 1
  Column_name: emp_no
      ........
*************************** 4. row ***************************
        Table: dept_emp
   Non_unique: 1
     Key_name: dept_no
 Seq_in_index: 1
  Column_name: dept_no
      ........
```

5. More SQL

5.1 Sub-Query

Results of one query can be used in another SQL statement. Subquery is useful if more than one tables are involved.

SELECT with Subquery

In the previous many-to-many product sales example, how to find the suppliers that do not supply any product? You can query for the suppliers that supply at least one product in the `products_suppliers` table, and then query the `suppliers` table for those that are not in the previous result set.

```
mysql> SELECT suppliers.name from suppliers
       WHERE suppliers.supplierID
          NOT IN (SELECT DISTINCT supplierID from products_suppliers);
```

Can you do this without sub-query?

A subquery may return a scalar, a single column, a single row, or a table. You can use comparison operator (e.g., `'='`, `'>'`) on scalar, `IN` or `NOT IN` for single row or column, `EXISTS` or `NOT EXIST` to test for empty set.

INSERT|UPDATE|DELETE with Subquery

You can also use a subquery with other SQL statements such as `INSERT`, `DELETE`, or `UPDATE`. For example,

```
-- Supplier 'QQ Corp' now supplies 'Pencil 6B'
-- You need to put the SELECT sub-queries in parentheses
mysql> INSERT INTO products_suppliers VALUES (
          (SELECT productID  FROM products  WHERE name = 'Pencil 6B'),
          (SELECT supplierID FROM suppliers WHERE name = 'QQ Corp'));

-- Supplier 'QQ Copr' no longer supplies any item
mysql> DELETE FROM products_suppliers
       WHERE supplierID = (SELECT supplierID FROM suppliers WHERE name = 'QQ Corp');
```

5.2 Working with Date and Time

Date and time are of particular interest for database applications. This is because business records often carry date/time information (e.g., `orderDate`, `deliveryDate`, `paymentDate`, `dateOfBirth`), as well as the need to time-stamp the creation and last-update of the records for auditing and security.

With date/time data types, you can sort the results by date, search for a particular date or a range of dates, calculate the difference between dates, compute a new date by adding/subtracting an interval from a given date.

Date By Example

Let's begin with Date (without Time) with the following example. Take note that date value must be written as a string in the format of `'yyyy-mm-dd'`, e.g., `'2012-01-31'`.

```
-- Create a table 'patients' of a clinic
mysql> CREATE TABLE patients (
          patientID      INT UNSIGNED  NOT NULL AUTO_INCREMENT,
          name           VARCHAR(30)   NOT NULL DEFAULT '',
          dateOfBirth    DATE          NOT NULL,
          lastVisitDate  DATE          NOT NULL,
          nextVisitDate  DATE          NULL,
                         -- The 'Date' type contains a date value in 'yyyy-mm-dd'
          PRIMARY KEY (patientID)
       );

mysql> INSERT INTO patients VALUES
          (1001, 'Ah Teck', '1991-12-31', '2012-01-20', NULL),
          (NULL, 'Kumar', '2011-10-29', '2012-09-20', NULL),
          (NULL, 'Ali', '2011-01-30', CURDATE(), NULL);
-- Date must be written as 'yyyy-mm-dd'
-- Function CURDATE() returns today's date
```

```
mysql> SELECT * FROM patients;
+-----------+---------+-------------+---------------+---------------+
| patientID | name    | dateOfBirth | lastVisitDate | nextVisitDate |
+-----------+---------+-------------+---------------+---------------+
|      1001 | Ah Teck | 1991-12-31  | 2012-01-20    | NULL          |
|      1002 | Kumar   | 2011-10-29  | 2012-09-20    | NULL          |
|      1003 | Ali     | 2011-01-30  | 2012-10-21    | NULL          |
+-----------+---------+-------------+---------------+---------------+

-- Select patients who last visited on a particular range of date
mysql> SELECT * FROM patients
       WHERE lastVisitDate BETWEEN '2012-09-15' AND CURDATE()
       ORDER BY lastVisitDate;
+-----------+-------+-------------+---------------+---------------+
| patientID | name  | dateOfBirth | lastVisitDate | nextVisitDate |
+-----------+-------+-------------+---------------+---------------+
|      1002 | Kumar | 2011-10-29  | 2012-09-20    | NULL          |
|      1003 | Ali   | 2011-01-30  | 2012-10-21    | NULL          |
+-----------+-------+-------------+---------------+---------------+

-- Select patients who were born in a particular year and sort by birth-month
-- Function YEAR(date), MONTH(date), DAY(date) returns
--   the year, month, day part of the given date
mysql> SELECT * FROM patients
       WHERE YEAR(dateOfBirth) = 2011
       ORDER BY MONTH(dateOfBirth), DAY(dateOfBirth);
+-----------+-------+-------------+---------------+---------------+
| patientID | name  | dateOfBirth | lastVisitDate | nextVisitDate |
+-----------+-------+-------------+---------------+---------------+
|      1003 | Ali   | 2011-01-30  | 2012-10-21    | NULL          |
|      1002 | Kumar | 2011-10-29  | 2012-09-20    | NULL          |
+-----------+-------+-------------+---------------+---------------+

-- Select patients whose birthday is today
mysql> SELECT * FROM patients
       WHERE MONTH(dateOfBirth) = MONTH(CURDATE())
          AND DAY(dateOfBirth) = DAY(CURDATE());

-- List the age of patients
-- Function TIMESTAMPDIFF(unit, start, end) returns the difference in the unit specified
mysql> SELECT name, dateOfBirth, TIMESTAMPDIFF(YEAR, dateOfBirth, CURDATE()) AS age
       FROM patients
       ORDER BY age, dateOfBirth;
+---------+-------------+------+
| name    | dateOfBirth | age  |
+---------+-------------+------+
| Kumar   | 2011-10-29  |    0 |
| Ali     | 2011-01-30  |    1 |
| Ah Teck | 1991-12-31  |   20 |
+---------+-------------+------+

-- List patients whose last visited more than 60 days ago
mysql> SELECT name, lastVisitDate FROM patients
       WHERE TIMESTAMPDIFF(DAY, lastVisitDate, CURDATE()) > 60;
-- Functions TO_DAYS(date) converts the date to days
mysql> SELECT name, lastVisitDate FROM patients
       WHERE TO_DAYS(CURDATE()) - TO_DAYS(lastVisitDate) > 60;

-- Select patients 18 years old or younger
-- Function DATE_SUB(date, INTERVAL x unit) returns the date
--   by subtracting the given date by x unit.
mysql> SELECT * FROM patients
       WHERE dateOfBirth > DATE_SUB(CURDATE(), INTERVAL 18 YEAR);

-- Schedule Ali's next visit to be 6 months from now
-- Function DATE_ADD(date, INTERVAL x unit) returns the date
--   by adding the given date by x unit
mysql> UPDATE patients
       SET nextVisitDate = DATE_ADD(CURDATE(), INTERVAL 6 MONTH)
       WHERE name = 'Ali';
```

Date/Time Functions

MySQL provides these built-in functions for getting the *current* date, time and datetime:

- `NOW()`: returns the current date and time in the format of `'YYYY-MM-DD HH:MM:SS'`.
- `CURDATE()` (or `CURRENT_DATE()`, or `CURRENT_DATE`): returns the current date in the format of `'YYYY-MM-DD'`.
- `CURTIME()` (or `CURRENT_TIME()`, or `CURRENT_TIME`): returns the current time in the format of `'HH:MM:SS'`.

For examples,

```
mysql> select now(), curdate(), curtime();
+---------------------+------------+-----------+
| now()               | curdate()  | curtime() |
+---------------------+------------+-----------+
| 2012-10-19 19:53:20 | 2012-10-19 | 19:53:20  |
+---------------------+------------+-----------+
```

SQL Date/Time Types

MySQL provides these date/time data types:

- **`DATETIME`**: stores both date and time in the format of `'YYYY-MM-DD HH:MM:SS'`. The valid range is `'1000-01-01 00:00:00'` to `'9999-12-31 23:59:59'`. You can set a value using the valid format (e.g., `'2011-08-15 00:00:00'`). You could also apply functions `NOW()` or `CURDATE()` (time will be set to `'00:00:00'`), but not `CURTIME()`.
- **`DATE`**: stores date only in the format of `'YYYY-MM-DD'`. The range is `'1000-01-01'` to `'9999-12-31'`. You could apply `CURDATE()` or `NOW()` (the time discarded) on this field.
- **`TIME`**: stores time only in the format of `'HH:MM:SS'`. You could apply `CURTIME()` or `NOW()` (the date discarded) for this field.
- **`YEAR(4|2)`**: in `'YYYY'` or `'YY'`. The range of years is 1901 to 2155. Use `DATE` type for year outside this range. You could apply `CURDATE()` to this field (month and day discarded).
- **`TIMESTAMP`**: similar to `DATETIME` but stored the number of seconds since January 1, 1970 UTC (Unix-style). The range is `'1970-01-01 00:00:00'` to `'2037-12-31 23:59:59'`.
  The differences between `DATETIME` and `TIMESTAMP` are:
  - the range,
  - support for time zone,
  - `TIMESTAMP` column could be declared with `DEFAULT CURRENT_TIMESTAMP` to set the default value to the current date/time. (All other data types' default, including `DATETIME`, must be a constant and not a function return value). You can also declare a `TIMESTAMP` column with "`ON UPDATE CURRENT_TIMESTAMP`" to capture the timestamp of the last update.

The date/time value can be entered manually as a string literal (e.g., `'2010-12-31 23:59:59'` for `DATAETIME`). MySQL will issue a warning and insert all zeros (e.g., `'0000-00-00 00:00:00'` for `DATAETIME`), if the value of date/time to be inserted is invalid or out-of-range. `'0000-00-00'` is called a "dummy" date.

More Date/Time Functions

Reference: MySQL's "Date and Time Functions" @ http://dev.mysql.com/doc/refman/5.5/en/date-and-time-functions.html.

There are many date/time functions:

- Extracting part of a date/time: `YEAR()`, `MONTH()`, `DAY()`, `HOUR()`, `MINUTE()`, `SECOND()`, e.g.,

  ```
  mysql> SELECT YEAR(NOW()), MONTH(NOW()), DAY(NOW()), HOUR(NOW()), MINUTE(NOW()), SECOND(NOW());
  +-------------+--------------+------------+-------------+---------------+---------------+
  | YEAR(NOW()) | MONTH(NOW()) | DAY(NOW()) | HOUR(NOW()) | MINUTE(NOW()) | SECOND(NOW()) |
  +-------------+--------------+------------+-------------+---------------+---------------+
  |        2012 |           10 |         24 |          11 |            54 |     45        |
  +-------------+--------------+------------+-------------+---------------+---------------+
  ```

- Extracting information: `DAYNAME()` (e.g., `'Monday'`), `MONTHNAME()` (e.g., `'March'`), `DAYOFWEEK()` (1=Sunday, ..., 7=Saturday), `DAYOFYEAR()` (1-366), ...

  ```
  mysql> SELECT DAYNAME(NOW()), MONTHNAME(NOW()), DAYOFWEEK(NOW()), DAYOFYEAR(NOW());
  +---------------+-----------------+-----------------+-----------------+
  | DAYNAME(NOW()) | MONTHNAME(NOW()) | DAYOFWEEK(NOW()) | DAYOFYEAR(NOW()) |
  +---------------+-----------------+-----------------+-----------------+
  ```

```
| Wednesday      | October          |                4 |              298 |
+----------------+------------------+------------------+------------------+
```

- Computing another date/time: `DATE_SUB(date, INTERVAL expr unit)`, `DATE_ADD(date, INTERVAL expr unit)`, `TIMESTAMPADD(unit, interval, timestamp)`, e.g.,

```
mysql> SELECT DATE_ADD('2012-01-31', INTERVAL 5 DAY);
2012-02-05

mysql> SELECT DATE_SUB('2012-01-31', INTERVAL 2 MONTH);
2011-11-30
```

- Computing interval: `DATEDIFF(end_date, start_date)`, `TIMEDIFF(end_time, start_time)`, `TIMESTAMPDIFF(unit, start_timestamp, end_timestamp)`, e.g.,

```
mysql> SELECT DATEDIFF('2012-02-01', '2012-01-28');
4

mysql> SELECT TIMESTAMPDIFF(DAY, '2012-02-01', '2012-01-28');
-4
```

- Representation: `TO_DAYS(date)` (days since year 0), `FROM_DAYS(day_number)`, e.g.,

```
mysql> SELECT TO_DAYS('2012-01-31');
734898

mysql> SELECT FROM_DAYS(734899);
2012-02-01
```

- Formatting: `DATE_FORMAT(date, formatSpecifier)`, e.g.,

```
mysql> SELECT DATE_FORMAT('2012-01-01', '%W %D %M %Y');
Sunday 1st January 2012
       -- %W: Weekday name
       -- %D: Day with suffix
       -- %M: Month name
       -- %Y: 4-digit year
       -- The format specifiers are case-sensitive

mysql> SELECT DATE_FORMAT('2011-12-31 23:59:30', '%W %D %M %Y %r');
Saturday 31st December 2011 11:59:30 PM
       -- %r: Time in 12-hour format with suffix AM/PM
```

Example

1. Create a table with various date/time columns. Only the `TIMESTAMP` column can have the `DEFAULT CURRENT_TIMESTAMP` and `ON UPDATE CURRENT_TIMESTAMP`.

```
mysql> CREATE TABLE IF NOT EXISTS `datetime_arena` (
          `description`  VARCHAR(50)  DEFAULT NULL,
          `cDateTime`    DATETIME     DEFAULT '1000-01-01  00:00:00',
          `cDate`        DATE         DEFAULT '1000-01-01 ',
          `cTime`        TIME         DEFAULT '00:00:00',
          `cYear`        YEAR         DEFAULT '0000',
          `cYear2`       YEAR(2)      DEFAULT '0000',
          `cTimeStamp`   TIMESTAMP    DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
       );

mysql> DESCRIBE `datetime_arena`;
+-------------+-------------+------+-----+---------------------+-----------------------------+
| Field       | Type        | Null | Key | Default             | Extra                       |
+-------------+-------------+------+-----+---------------------+-----------------------------+
| description | varchar(50) | YES  |     | NULL                |                             |
| cDateTime   | datetime    | YES  |     | 1000-01-01 00:00:00 |                             |
| cDate       | date        | YES  |     | 1000-01-01          |                             |
| cTime       | time        | YES  |     | 00:00:00            |                             |
| cYear       | year(4)     | YES  |     | 0000                |                             |
| cYear2      | year(4)     | YES  |     | 0000                |                             |
| cTimeStamp  | timestamp   | NO   |     | CURRENT_TIMESTAMP   | on update CURRENT_TIMESTAMP |
+-------------+-------------+------+-----+---------------------+-----------------------------+
```

Notes:

- Don't use `year(2)` anymore.
- From MySQL 5.7, the supported range for datetime is `'1000-01-01 00:00:00'` to `'9999-12-31 23:59:59'`.

2. Insert values manually using string literals.

```
mysql> INSERT INTO `datetime_arena`
          (`description`, `cDateTime`, `cDate`, `cTime`, `cYear`, `cYear2`)
       VALUES
          ('Manual Entry', '2001-01-01 23:59:59', '2002-02-02', '12:30:30', '2004', '05');

mysql> SELECT * FROM `datetime_arena` WHERE description='Manual Entry';
+--------------+---------------------+------------+----------+-------+--------+---------------------+
| description  | cDateTime           | cDate      | cTime    | cYear | cYear2 | cTimeStamp          |
+--------------+---------------------+------------+----------+-------+--------+---------------------+
| Manual Entry | 2001-01-01 23:59:59 | 2002-02-02 | 12:30:30 |  2004 |     05 | 2010-04-08 14:44:37 |
+--------------+---------------------+------------+----------+-------+--------+---------------------+
```

3. Checking the on-update for `TIMSTAMP`.

```
mysql> UPDATE `datetime_arena` SET `cYear2`='99' WHERE description='Manual Entry';

mysql> SELECT * FROM `datetime_arena` WHERE description='Manual Entry';
+--------------+---------------------+------------+----------+-------+--------+---------------------+
| description  | cDateTime           | cDate      | cTime    | cYear | cYear2 | cTimeStamp          |
+--------------+---------------------+------------+----------+-------+--------+---------------------+
| Manual Entry | 2001-01-01 23:59:59 | 2002-02-02 | 12:30:30 |  2004 |     99 | 2010-04-08 14:44:48 |
+--------------+---------------------+------------+----------+-------+--------+---------------------+
```

4. Insert values using MySQL built-in functions `now()`, `curdate()`, `curtime()`.

```
mysql> INSERT INTO `datetime_arena`
          (`description`, `cDateTime`, `cDate`, `cTime`, `cYear`, `cYear2`)
       VALUES
          ('Built-in Functions', now(), curdate(), curtime(), now(), now());

mysql> SELECT * FROM `datetime_arena` WHERE description='Built-in Functions';
+--------------------+---------------------+------------+----------+-------+--------+---------------------+
| description        | cDateTime           | cDate      | cTime    | cYear | cYear2 | cTimeStamp          |
+--------------------+---------------------+------------+----------+-------+--------+---------------------+
| Built-in Functions | 2010-04-08 14:45:48 | 2010-04-08 | 14:45:48 |  2010 |     10 | 2010-04-08 14:45:48 |
+--------------------+---------------------+------------+----------+-------+--------+---------------------+
```

5. Insert invalid or out-of-range values. MySQL replaces with all zeros.

```
mysql> INSERT INTO `datetime_arena`
          (`description`, `cDateTime`, `cDate`, `cTime`, `cYear`, `cYear2`)
       VALUES
          ('Error Input', '2001-13-31 23:59:59', '2002-13-31', '12:61:61', '99999', '999');

mysql> SELECT * FROM `datetime_arena` WHERE description='Error Input';
+-------------+---------------------+------------+----------+-------+--------+---------------------+
| description | cDateTime           | cDate      | cTime    | cYear | cYear2 | cTimeStamp          |
+-------------+---------------------+------------+----------+-------+--------+---------------------+
| Error Input | 0000-00-00 00:00:00 | 0000-00-00 | 00:00:00 |  0000 |     00 | 2010-04-08 14:46:10 |
+-------------+---------------------+------------+----------+-------+--------+---------------------+
```

Note: Might not work in MySQL 5.7?!

6. An useful built-in function `INTERVAL` can be used to compute a future date, e.g.,

```
mysql> SELECT `cDate`, `cDate` + INTERVAL 30 DAY, `cDate` + INTERVAL 1 MONTH FROM `datetime_arena`;
+------------+--------------------------+---------------------------+
| cDate      | `cDate` + INTERVAL 30 DAY | `cDate` + INTERVAL 1 MONTH |
+------------+--------------------------+---------------------------+
| 2002-02-02 | 2002-03-04               | 2002-03-02                |
| 2010-04-08 | 2010-05-08               | 2010-05-08                |
| 0000-00-00 | NULL                     | NULL                      |
+------------+--------------------------+---------------------------+
```

5.3 View

A view is a *virtual table* that contains no physical data. It provide an alternative way to look at the data.

Example

```
-- Define a VIEW called supplier_view from products, suppliers and products_suppliers tables
mysql> CREATE VIEW supplier_view
       AS
       SELECT suppliers.name as `Supplier Name`, products.name as `Product Name`
       FROM products
          JOIN suppliers ON products.productID = products_suppliers.productID
          JOIN products_suppliers ON suppliers.supplierID = products_suppliers.supplierID;

-- You can treat the VIEW defined like a normal table
mysql> SELECT * FROM supplier_view;
+---------------+--------------+
| Supplier Name | Product Name |
+---------------+--------------+
| ABC Traders   | Pencil 3B    |
| ABC Traders   | Pencil 4B    |
| ABC Traders   | Pencil 5B    |
| XYZ Company   | Pencil 6B    |
+---------------+--------------+

mysql> SELECT * FROM supplier_view WHERE `Supplier Name` LIKE 'ABC%';
+---------------+--------------+
| Supplier Name | Product Name |
+---------------+--------------+
| ABC Traders   | Pencil 3B    |
| ABC Traders   | Pencil 4B    |
| ABC Traders   | Pencil 5B    |
+---------------+--------------+
```

Example

```
mysql> DROP VIEW IF EXISTS patient_view;

mysql> CREATE VIEW patient_view
       AS
       SELECT
          patientID AS ID,
          name AS Name,
          dateOfBirth AS DOB,
          TIMESTAMPDIFF(YEAR, dateOfBirth, NOW()) AS Age
       FROM patients
       ORDER BY Age, DOB;

mysql> SELECT * FROM patient_view WHERE Name LIKE 'A%';
+------+---------+------------+------+
| ID   | Name    | DOB        | Age  |
+------+---------+------------+------+
| 1003 | Ali     | 2011-01-30 |    1 |
| 1001 | Ah Teck | 1991-12-31 |   20 |
+------+---------+------------+------+

mysql> SELECT * FROM patient_view WHERE age >= 18;
+------+---------+------------+------+
| ID   | Name    | DOB        | Age  |
+------+---------+------------+------+
| 1001 | Ah Teck | 1991-12-31 |   20 |
+------+---------+------------+------+
```

5.4 Transactions

A *atomic transaction* is a set of SQL statements that either ALL succeed or ALL fail. Transaction is important to ensure that there is no *partial* update to the database, given an atomic of SQL statements. Transactions are carried out via `COMMIT` and `ROLLBACK`.

Example

```
mysql> CREATE TABLE accounts (
          name     VARCHAR(30),
          balance  DECIMAL(10,2)
       );

mysql> INSERT INTO accounts VALUES ('Paul', 1000), ('Peter', 2000);
mysql> SELECT * FROM accounts;
+-------+---------+
| name  | balance |
+-------+---------+
| Paul  | 1000.00 |
| Peter | 2000.00 |
+-------+---------+

-- Transfer money from one account to another account
mysql> START TRANSACTION;
mysql> UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';
mysql> UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';
mysql> COMMIT;     -- Commit the transaction and end transaction
mysql> SELECT * FROM accounts;
+-------+---------+
| name  | balance |
+-------+---------+
| Paul  |  900.00 |
| Peter | 2100.00 |
+-------+---------+

mysql> START TRANSACTION;
mysql> UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';
mysql> UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';
mysql> ROLLBACK;    -- Discard all changes of this transaction and end Transaction
mysql> SELECT * FROM accounts;
+-------+---------+
| name  | balance |
+-------+---------+
| Paul  |  900.00 |
| Peter | 2100.00 |
+-------+---------+
```

If you start another `mysql` client and do a SELECT during the transaction (before the commit or rollback), you will not see the changes.

Alternatively, you can also disable the so-called `autocommit` mode, which is set by default and commit every single SQL statement.

```
-- Disable autocommit by setting it to false (0)
mysql> SET autocommit = 0;
mysql> UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';
mysql> UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';
mysql> COMMIT;
mysql> SELECT * FROM accounts;
+-------+---------+
| name  | balance |
+-------+---------+
| Paul  |  800.00 |
| Peter | 2200.00 |
+-------+---------+

mysql> UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';
mysql> UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';
mysql> ROLLBACK;
mysql> SELECT * FROM accounts;
+-------+---------+
| name  | balance |
+-------+---------+
| Paul  |  800.00 |
| Peter | 2200.00 |
+-------+---------+

mysql> SET autocommit = 1;   -- Enable autocommit
```

A *transaction* groups a set of operations into a unit that meets the ACID test:

1. Atomicity: If all the operations succeed, changes are *committed* to the database. If any of the operations fails, the entire transaction is *rolled back*, and no change is made to the database. In other words, there is no partial update.
2. Consistency: A transaction transform the database from one consistent state to another consistent state.
3. Isolation: Changes to a transaction are not visible to another transaction until they are committed.
4. Durability: Committed changes are durable and never lost.

5.5 User Variables

In MySQL, you can define user variables via:

1. `@varname :=value` in a `SELECT` command, or
2. `SET @varname := value` or `SET @varname = value` command.

For examples,

```
mysql> SELECT @ali_dob := dateOfBirth FROM patients WHERE name = 'Ali';
mysql> SELECT name WHERE dateOfBirth < @ali_dob;

mysql> SET @today := CURDATE();
mysql> SELECT name FROM patients WHERE nextVisitDate = @today;
```

6. More on JOIN

6.1 INNER JOIN

In an inner join of two tables, each row of the first table is combined (joined) with every row of second table. Suppose that there are *n1* rows in the first table and *n2* rows in the second table, `INNER JOIN` produces all combinations of *n1*×*n2* rows - it is known as *Cartesian Product* or *Cross Product*.

Example

```
mysql> DROP TABLE IF EXISTS t1, t2;

mysql> CREATE TABLE t1 (
          id      INT PRIMARY KEY,
          `desc`  VARCHAR(30)
       );
-- `desc` is a reserved word - must be back-quoted

mysql> CREATE TABLE t2 (
          id      INT PRIMARY KEY,
          `desc`  VARCHAR(30)
       );

mysql> INSERT INTO t1 VALUES
         (1, 'ID 1 in t1'),
         (2, 'ID 2 in t1'),
         (3, 'ID 3 in t1');

mysql> INSERT INTO t2 VALUES
         (2, 'ID 2 in t2'),
         (3, 'ID 3 in t2'),
         (4, 'ID 4 in t2');

mysql> SELECT * FROM t1;
+----+------------+
| id | desc       |
+----+------------+
|  1 | ID 1 in t1 |
|  2 | ID 2 in t1 |
|  3 | ID 3 in t1 |
+----+------------+

mysql> SELECT * FROM t2;
+----+------------+
```

```
| id | desc       |
+----+------------+
|  2 | ID 2 in t2 |
|  3 | ID 3 in t2 |
|  4 | ID 4 in t2 |
+----+------------+

mysql> SELECT *
       FROM t1 INNER JOIN t2;
+----+------------+----+------------+
| id | desc       | id | desc       |
+----+------------+----+------------+
|  1 | ID 1 in t1 |  2 | ID 2 in t2 |
|  2 | ID 2 in t1 |  2 | ID 2 in t2 |
|  3 | ID 3 in t1 |  2 | ID 2 in t2 |
|  1 | ID 1 in t1 |  3 | ID 3 in t2 |
|  2 | ID 2 in t1 |  3 | ID 3 in t2 |
|  3 | ID 3 in t1 |  3 | ID 3 in t2 |
|  1 | ID 1 in t1 |  4 | ID 4 in t2 |
|  2 | ID 2 in t1 |  4 | ID 4 in t2 |
|  3 | ID 3 in t1 |  4 | ID 4 in t2 |
+----+------------+----+------------+
-- SELECT all columns in t1 and t2 (*)
-- INNER JOIN produces ALL combinations of rows in t1 and t2
```

You can impose constrain by using the `ON` clause, for example,

```
mysql> SELECT *
       FROM t1 INNER JOIN t2 ON t1.id = t2.id;
+----+------------+----+------------+
| id | desc       | id | desc       |
+----+------------+----+------------+
|  2 | ID 2 in t1 |  2 | ID 2 in t2 |
|  3 | ID 3 in t1 |  3 | ID 3 in t2 |
+----+------------+----+------------+
```

Take note that the following are equivalent:

```
mysql> SELECT *
       FROM t1 INNER JOIN t2 ON t1.id = t2.id;
mysql> SELECT *
       FROM t1 JOIN t2 ON t1.id = t2.id;        -- default JOIN is INNER JOIN
mysql> SELECT *
       FROM t1 CROSS JOIN t2 ON t1.id = t2.id;  -- Also called CROSS JOIN

-- You can use USING clause if the join-columns have the same name
mysql> SELECT *
       FROM t1 INNER JOIN t2 USING (id);
+----+------------+------------+
| id | desc       | desc       |
+----+------------+------------+
|  2 | ID 2 in t1 | ID 2 in t2 |
|  3 | ID 3 in t1 | ID 3 in t2 |
+----+------------+------------+
   -- Only 3 columns in the result set, instead of 4 columns with ON clause

mysql> SELECT *
       FROM t1 INNER JOIN t2 WHERE t1.id = t2.id;  -- Use WHERE instead of ON
mysql> SELECT *
       FROM t1, t2 WHERE t1.id = t2.id;            -- Use "commas" operator to join
```

6.2 OUTER JOIN - LEFT JOIN and RIGHT JOIN

`INNER JOIN` with constrain (`ON` or `USING`) produces rows that are found in both tables. On the other hand, `OUTER JOIN` can produce rows that are in one table, but not in another table. There are two kinds of `OUTER JOIN`s: `LEFT JOIN` produces rows that are in the left table, but may not in the right table; whereas `RIGHT JOIN` produces rows that are in the right table but may not in the left table.

In a `LEFT JOIN`, when a row in the left table does not match with the right table, it is still selected but by combining with a "fake" record of all `NULL`s for the right table.

```
mysql> SELECT *
       FROM t1 LEFT JOIN t2 ON t1.id = t2.id;
+----+------------+------+------------+
| id | desc       | id   | desc       |
+----+------------+------+------------+
|  1 | ID 1 in t1 | NULL | NULL       |
|  2 | ID 2 in t1 |    2 | ID 2 in t2 |
|  3 | ID 3 in t1 |    3 | ID 3 in t2 |
+----+------------+------+------------+

mysql> SELECT *
       FROM t1 LEFT JOIN t2 USING (id);
+----+------------+------------+
| id | desc       | desc       |
+----+------------+------------+
|  1 | ID 1 in t1 | NULL       |
|  2 | ID 2 in t1 | ID 2 in t2 |
|  3 | ID 3 in t1 | ID 3 in t2 |
+----+------------+------------+

mysql> SELECT *
       FROM t1 RIGHT JOIN t2 ON t1.id = t2.id;
+------+------------+----+------------+
| id   | desc       | id | desc       |
+------+------------+----+------------+
|    2 | ID 2 in t1 |  2 | ID 2 in t2 |
|    3 | ID 3 in t1 |  3 | ID 3 in t2 |
| NULL | NULL       |  4 | ID 4 in t2 |
+------+------------+----+------------+

mysql> SELECT *
       FROM t1 RIGHT JOIN t2 USING (id);
+----+------------+------------+
| id | desc       | desc       |
+----+------------+------------+
|  2 | ID 2 in t2 | ID 2 in t1 |
|  3 | ID 3 in t2 | ID 3 in t1 |
|  4 | ID 4 in t2 | NULL       |
+----+------------+------------+
```

As the result, `LEFT JOIN` ensures that the result set contains every row on the left table. This is important, as in some queries, you are interested to have result on every row on the left table, with no match in the right table, e.g., searching for items without supplier. For example,

```
mysql> SELECT t1.id, t1.desc
       FROM t1 LEFT JOIN t2 USING (id)
       WHERE t2.id IS NULL;
+----+------------+
| id | desc       |
+----+------------+
|  1 | ID 1 in t1 |
+----+------------+
```

Take note that the followings are equivalent:

```
mysql> SELECT *
       FROM t1 LEFT JOIN t2 ON t1.id = t2.id;
mysql> SELECT *
       FROM t1 LEFT OUTER JOIN t2 ON t1.id = t2.id;

mysql> SELECT *
       FROM t1 LEFT JOIN t2 USING (id);  -- join-columns have same name
+----+------------+------------+
| id | desc       | desc       |
+----+------------+------------+
|  1 | ID 1 in t1 | NULL       |
|  2 | ID 2 in t1 | ID 2 in t2 |
|  3 | ID 3 in t1 | ID 3 in t2 |
+----+------------+------------+

-- WHERE clause CANNOT be used on OUTER JOIN
mysql> SELECT *
```

```
       FROM t1 LEFT JOIN t2 WHERE t1.id = t2.id;
ERROR 1064 (42000): You have an error in your SQL syntax;
```

7. Exercises

7.1 Rental System

Peter runs a small car rental company with 10 cars and 5 trucks. He engages you to design a web portal to put his operation online.

For the initial phase, the web portal shall provide these basic functions:

1. Maintaining the records of the vehicles and customers.
2. Inquiring about the availability of vehicle, and
3. Reserving a vehicle for rental.

A customer record contains his/her name, address and phone number.

A vehicle, identified by the vehicle registration number, can be rented on a daily basis. The rental rate is different for different vehicles. There is a discount of 20% for rental of 7 days or more.

A customer can rental a vehicle from a start date to an end date. A special customer discount, ranging from 0-50%, can be given to preferred customers.

Database

The initial database contains 3 tables: `vehicles`, `customers`, and `rental_records`. The `rental_records` is a *junction table* supporting many-to-many relationship between `vehicles` and `customers`.

```
DROP DATABASE IF EXISTS `rental_db`;
CREATE DATABASE `rental_db`;
USE `rental_db`;

-- Create `vehicles` table
DROP TABLE IF EXISTS `vehicles`;
CREATE TABLE `vehicles` (
   `veh_reg_no`  VARCHAR(8)    NOT NULL,
   `category`    ENUM('car', 'truck')  NOT NULL DEFAULT 'car',
                 -- Enumeration of one of the items in the list
   `brand`       VARCHAR(30)   NOT NULL DEFAULT '',
   `desc`        VARCHAR(256)  NOT NULL DEFAULT '',
                 -- desc is a keyword (for descending) and must be back-quoted
   `photo`       BLOB          NULL,   -- binary large object of up to 64KB
                 -- to be implemented later
   `daily_rate`  DECIMAL(6,2)  NOT NULL DEFAULT 9999.99,
                 -- set default to max value
   PRIMARY KEY (`veh_reg_no`),
   INDEX (`category`)  -- Build index on this column for fast search
) ENGINE=InnoDB;
   -- MySQL provides a few ENGINEs.
   -- The InnoDB Engine supports foreign keys and transactions
DESC `vehicles`;
SHOW CREATE TABLE `vehicles` \G
SHOW INDEX FROM `vehicles` \G

-- Create `customers` table
DROP TABLE IF EXISTS `customers`;
CREATE TABLE `customers` (
   `customer_id`  INT UNSIGNED  NOT NULL AUTO_INCREMENT,
                  -- Always use INT for AUTO_INCREMENT column to avoid run-over
   `name`         VARCHAR(30)   NOT NULL DEFAULT '',
   `address`      VARCHAR(80)   NOT NULL DEFAULT '',
   `phone`        VARCHAR(15)   NOT NULL DEFAULT '',
   `discount`     DOUBLE        NOT NULL DEFAULT 0.0,
   PRIMARY KEY (`customer_id`),
   UNIQUE INDEX (`phone`),  -- Build index on this unique-value column
   INDEX (`name`)           -- Build index on this column
) ENGINE=InnoDB;
DESC `customers`;
SHOW CREATE TABLE `customers` \G
```

```
SHOW INDEX FROM `customers` \G

-- Create `rental_records` table
DROP TABLE IF EXISTS `rental_records`;
CREATE TABLE `rental_records` (
   `rental_id`    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
   `veh_reg_no`   VARCHAR(8)    NOT NULL,
   `customer_id`  INT UNSIGNED  NOT NULL,
   `start_date`   DATE          NOT NULL DEFAULT '0000-00-00',
   `end_date`     DATE          NOT NULL DEFAULT '0000-00-00',
   `lastUpdated`  TIMESTAMP     NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
      -- Keep the created and last updated timestamp for auditing and security
   PRIMARY KEY (`rental_id`),
   FOREIGN KEY (`customer_id`) REFERENCES `customers` (`customer_id`)
      ON DELETE RESTRICT ON UPDATE CASCADE,
      -- Disallow deletion of parent record if there are matching records here
      -- If parent record (customer_id) changes, update the matching records here
   FOREIGN KEY (`veh_reg_no`) REFERENCES `vehicles` (`veh_reg_no`)
      ON DELETE RESTRICT ON UPDATE CASCADE
) ENGINE=InnoDB;
DESC `rental_records`;
SHOW CREATE TABLE `rental_records` \G
SHOW INDEX FROM `rental_records` \G

-- Inserting test records
INSERT INTO `vehicles` VALUES
   ('SBA1111A', 'car', 'NISSAN SUNNY 1.6L', '4 Door Saloon, Automatic', NULL, 99.99),
   ('SBB2222B', 'car', 'TOYOTA ALTIS 1.6L', '4 Door Saloon, Automatic', NULL, 99.99),
   ('SBC3333C', 'car', 'HONDA CIVIC 1.8L',  '4 Door Saloon, Automatic', NULL, 119.99),
   ('GA5555E', 'truck', 'NISSAN CABSTAR 3.0L',  'Lorry, Manual ', NULL, 89.99),
   ('GA6666F', 'truck', 'OPEL COMBO 1.6L',  'Van, Manual', NULL, 69.99);
   -- No photo yet, set to NULL
SELECT * FROM `vehicles`;

INSERT INTO `customers` VALUES
   (1001, 'Tan Ah Teck', '8 Happy Ave', '88888888', 0.1),
   (NULL, 'Mohammed Ali', '1 Kg Java', '99999999', 0.15),
   (NULL, 'Kumar', '5 Prince Road', '55555555', 0),
   (NULL, 'Kevin Jones', '2 Sunset boulevard', '22222222', 0.2);
SELECT * FROM `customers`;

INSERT INTO `rental_records` VALUES
  (NULL, 'SBA1111A', 1001, '2012-01-01', '2012-01-21', NULL),
  (NULL, 'SBA1111A', 1001, '2012-02-01', '2012-02-05', NULL),
  (NULL, 'GA5555E',  1003, '2012-01-05', '2012-01-31', NULL),
  (NULL, 'GA6666F',  1004, '2012-01-20', '2012-02-20', NULL);
SELECT * FROM `rental_records`;
```

Exercises

1. Customer `'Tan Ah Teck'` has rented `'SBA1111A'` from today for 10 days. (Hint: You need to insert a rental record. Use a `SELECT` subquery to get the `customer_id`. Use `CURDATE()` (or `NOW()`) for today; and `DATE_ADD(CURDATE(), INTERVAL x unit)` to compute a future date.)

   ```
   INSERT INTO rental_records VALUES
      (NULL,
       'SBA1111A',
       (SELECT customer_id FROM customers WHERE name='Tan Ah Teck'),
       CURDATE(),
       DATE_ADD(CURDATE(), INTERVAL 10 DAY),
       NULL);
   ```

2. Customer `'Kumar'` has rented `'GA5555E'` from tomorrow for 3 months.
3. List all rental records (start date, end date) with vehicle's registration number, brand, and customer name, sorted by vehicle's categories followed by start date.

   ```
   SELECT
      r.start_date  AS `Start Date`,
      r.end_date    AS `End Date`,
      r.veh_reg_no  AS `Vehicle No`,
   ```

```
   v.brand       AS `Vehicle Brand`,
   c.name        AS `Customer Name`
FROM rental_records AS r
   INNER JOIN vehicles  AS v USING (veh_reg_no)
   INNER JOIN customers AS c USING (customer_id)
ORDER BY v.category, start_date;
```

4. List all the expired rental records (end_date before `CURDATE()`).
5. List the vehicles rented out on `'2012-01-10'` (not available for rental), in columns of vehicle registration no, customer name, start date and end date. (Hint: the given date is in between the `start_date` and `end_date`.)
6. List all vehicles rented out today, in columns registration number, customer name, start date, end date.
7. Similarly, list the vehicles rented out (not available for rental) for the period from `'2012-01-03'` to `'2012-01-18'`. (Hint: `start_date` is inside the range; or `end_date` is inside the range; or `start_date` is before the range and `end_date` is beyond the range.)
8. List the vehicles (registration number, brand and description) available for rental (not rented out) on `'2012-01-10'` (Hint: You could use a subquery based on a earlier query).
9. Similarly, list the vehicles available for rental for the period from `'2012-01-03'` to `'2012-01-18'`.
10. Similarly, list the vehicles available for rental from today for 10 days.
11. Foreign Key Test:
    a. Try deleting a parent row with matching row(s) in child table(s), e.g., delete `'GA6666F'` from `vehicles` table (`ON DELETE RESTRICT`).
    b. Try updating a parent row with matching row(s) in child table(s), e.g., rename `'GA6666F'` to `'GA9999F'` in `vehicles` table. Check the effects on the child table `rental_records` (`ON UPDATE CASCADE`).
    c. Remove `'GA6666F'` from the database (Hints: Remove it from child table `rental_records`; then parent table `vehicles`.)
12. Payments: A rental could be paid over a number of payments (e.g., deposit, installments, full payment). Each payment is for one rental. Create a new table called `payments`. Need to create columns to facilitate proper audit check (such as `create_date`, `create_by`, `last_update_date`, `last_update_by`, etc.)

```
DROP TABLE IF EXISTS `payments`;
CREATE TABLE payments (
   `payment_id`   INT UNSIGNED  NOT NULL AUTO_INCREMENT,
   `rental_id`    INT UNSIGNED  NOT NULL,
   `amount`       DECIMAL(8,2)  NOT NULL DEFAULT 0,
   `mode`         ENUM('cash', 'credit card', 'check'),
   `type`         ENUM('deposit', 'partial', 'full') NOT NULL DEFAULT 'full',
   `remark`       VARCHAR(255),
   `created_date`       DATETIME      NOT NULL,
   `created_by`         INT UNSIGNED  NOT NULL,  -- staff_id
                        -- Use a trigger to update create_date and create_by automatically
   `last_updated_date`  TIMESTAMP     DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
                        -- Updated by the system automatically
   `last_updated_by`    INT UNSIGNED  NOT NULL,
                        -- Use a trigger to update created_by
   PRIMARY KEY (`payment_id`),
   INDEX       (`rental_id`),
   FOREIGN KEY (`rental_id`) REFERENCES rental_records (`rental_id`)
) ENGINE=InnoDB;
DESC `payments`;
SHOW CREATE TABLE `payments` \G
SHOW INDEX FROM `payments` \G
```

13. Staff: Keeping track of staff serving the customers. Create a new `staff` table. Assume that each transaction is handled by one staff, we can add a new column called `staff_id` in the `rental_records` table,

```
DROP TABLE IF EXISTS `staff`;
CREATE TABLE `staff` (
   `staff_id`   INT UNSIGNED  NOT NULL AUTO_INCREMENT,
                -- Always use INT for AUTO_INCREMENT column to prevent run-over
   `name`       VARCHAR(30)   NOT NULL DEFAULT '',
   `title`      VARCHAR(30)   NOT NULL DEFAULT '',
   `address`    VARCHAR(80)   NOT NULL DEFAULT '',
   `phone`      VARCHAR(15)   NOT NULL DEFAULT '',
   `report_to`  INT UNSIGNED  NOT NULL,
                -- Reports to manager staff_id. Boss reports to himself
   PRIMARY KEY  (`staff_id`),
   UNIQUE INDEX (`phone`),  -- Build index on this unique-value column
```

```
   INDEX        (`name`),   -- Build index on this column
   FOREIGN KEY  (`report_to`) REFERENCES `staff` (`staff_id`)
      -- Reference itself
) ENGINE=InnoDB;
DESC `staff`;
SHOW INDEX FROM `staff` \G

INSERT INTO staff VALUE (8001, 'Peter Johns', 'Managing Director', '1 Happy Ave', '12345678', 8001);
SELECT * FROM staff;

-- Add a new column to rental_records table
ALTER TABLE `rental_records` ADD COLUMN `staff_id`  INT UNSIGNED  NOT NULL;
-- Need to set to a valid value, before adding the foreign key
UPDATE `rental_records` SET `staff_id` = 8001;
ALTER TABLE `rental_records` ADD FOREIGN KEY (`staff_id`) REFERENCES staff (`staff_id`)
   ON DELETE RESTRICT ON UPDATE CASCADE;

SHOW CREATE TABLE `rental_records` \G
SHOW INDEX FROM `rental_records` \G

-- Also Add a new column to payments table
ALTER TABLE `payments` ADD COLUMN `staff_id`  INT UNSIGNED  NOT NULL;
-- Need to set to a valid value, before adding the foreign key
UPDATE `payments` SET `staff_id` = 8001;
ALTER TABLE `payments` ADD FOREIGN KEY (`staff_id`) REFERENCES staff (`staff_id`)
   ON DELETE RESTRICT ON UPDATE CASCADE;

SHOW CREATE TABLE `payments` \G
SHOW INDEX FROM `payments` \G
```

Advanced Exercises

1. Adding Photo: We could store photo in MySQL using data type of `BLOB` (Binary Large Object) (up to 64KB), `MEDIUMBLOB` (up to 16MBytes), `LONGBOLB` (up to 4GBytes). For example,

   ```
   -- Use function LOAD_FILE to load a picture file into a BLOB field
   UPDATE vehicles SET photo=LOAD_FILE('d:/temp/car.jpg') WHERE veh_reg_no = 'SBA1111A';
   SELECT * FROM vehicles WHERE veh_reg_no = 'SBA1111A' \G
   ```

   You can conveniently load and view the photo via graphical tools such as MySQL Workbench. To load a image in MySQL Workbench ⇒ right-click on the cell ⇒ Load Value From File ⇒ Select the image file. To view the image ⇒ right-click on the BLOB cell ⇒ Open Value in Editor ⇒ choose "Image" pane.

   I also include a Java program for reading and writing image BLOB from/to the database, based on this example: "`TestImageBLOB.java`".

2. `VIEW`: Create a `VIEW` called `rental_prices` on the `rental_records` with an additional column called `price`. Show all the records of the `VIEW`.

   ```
   DROP VIEW IF EXISTS rental_prices;
   CREATE VIEW rental_prices
   AS
   SELECT
      v.veh_reg_no    AS `Vehicle No`,
      v.daily_rate    AS `Daily Rate`,
      c.name          AS `Customer Name`,
      c.discount*100  AS `Customer Discount (%)`,
      r.start_date    AS `Start Date`,
      r.end_date      AS `End Date`,
      DATEDIFF(r.end_date, r.start_date) AS `Duration`,
      -- Compute the rental price
      -- Preferred customer has discount, 20% discount for 7 or more days
      -- CAST the result from DOUBLE to DECIMAL(8,2)
      CAST(
         IF (DATEDIFF(r.end_date, r.start_date) < 7,
             DATEDIFF(r.end_date, r.start_date)*daily_rate*(1-discount),
             DATEDIFF(r.end_date, r.start_date)*daily_rate*(1-discount)*0.8)
         AS DECIMAL(8,2)) AS price
   FROM rental_records AS r
      INNER JOIN vehicles  AS v USING (veh_reg_no)
      INNER JOIN customers AS c USING (customer_id);

   DESC `rental_prices`;
   ```

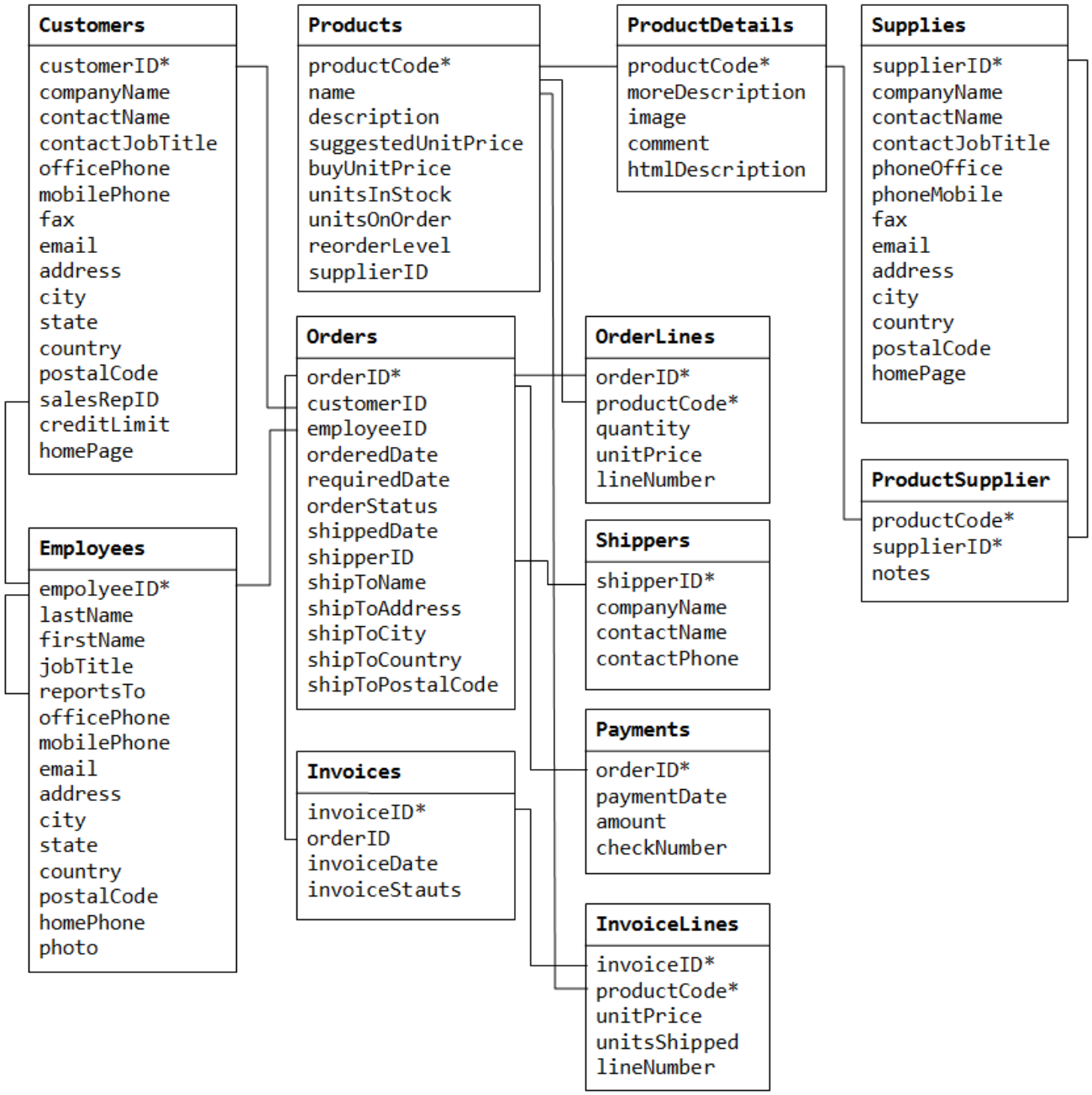
```
SHOW CREATE VIEW `rental_prices` \G

-- Try selecting all rows
SELECT * FROM `rental_prices`;
```

It is probably easier to compute the price using a program/procedure, instead of inside the view.

3. From the `payments` table, create a view to show the outstanding balance.
4. Define more views.
5. `FUNCTION`: Write a function to compute the rental price.
6. Define more procedures and functions.
7. `TRIGGER`: Write a trigger for the `created_date` and `created_by` columns of the `payments` table.
8. Define more triggers.
9. Implement discount on weekday (Monday to Friday, except public holiday): Need to set up a new table called `public_holiday` with columns `date` and `description`. Use function `DAYOFWEEK` (1=Sunday, ..., 7=Saturday) to check for weekday or weekend.

```
-- pseudocode for calculating rental price
price = 0;
for each date from start_date to end_date {
   if date is weekend or public_holiday, price += daily_rate;
   else price += daily_rate*(1-discount);
}
if (duration >= 7) price *= (1 - long_duration_discount);
price *= (1 - preferred_customer_discount);
```

7.2 Product Sales Database



[TODO] Explanation

Link to MySQL References & Resources

Latest version tested: MySQL Community Server 5.6.20
Last modified: September, 2014