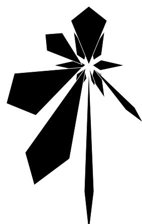
# Short Guide to Practical MySQL Query

Revision 1.1

July 16, 2010

by

Aloysius Indrayanto



(C) 2010 AnemoneSoft.com

This document is multi-licensed under the Creative Commons Attribution Share-Alike (CC-BY-SA) license version 3.0 and the GNU Free Documentation License (GNU FDL) version 1.3 or later.

# 1. Introduction

MySQL Community Server is a free and open source RDBMS (Relational Database Management System) that provides a multi-user database system. MySQL was initially developed by Michael Widenius. Currently, MySQL is owned and sponsored by Sun Microsystems (a subsidiary of Oracle Corporation). MySQL has been used in large-scale products, such as Wikipedia, Google, and Facebook.

This tutorial can be regarded as the continuation of our previous MySQL-related tutorial (Introduction to Database Programming using PHP and MySQL). This tutorial will present some practical MySQL queries. This tutorial utilizes some of MySQL's extensions to the standard SQL that may not be supported by other database vendors.

# 2. Requirements

A basic knowledge in SQL will be needed to understand the topic discussed in this tutorial. A system with an installation of MySQL Community Server version 5.x will be needed to run the examples. A test database will be also needed to run the examples.

Please login to a MySQL console as the *root* user (or any user with at least `CREATE`, `DROP`, `INDEX`, `SELECT`, `INSERT`, `UPDATE`, and `DELETE` privileges) and enter (execute) all the statements shown below. You can copy and paste the statements directly to your MySQL console.

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

CREATE TABLE WeaponData (
    WeaponID    INT         NOT NULL PRIMARY KEY AUTO_INCREMENT,
    Name        VARCHAR(64) NOT NULL,
    MedianPower TINYINT     NOT NULL,
    Span        TINYINT     NOT NULL
) ENGINE = InnoDB CHARSET = UTF8;
INSERT INTO WeaponData
            (Name, MedianPower, Span)
VALUES      ('Sword', 10, 2),
            ('Axe', 15, 5);

CREATE TABLE UserData (
    UserID           INT         NOT NULL PRIMARY KEY AUTO_INCREMENT,
    Name             VARCHAR(64) NOT NULL,
    RegistrationDate DATETIME    NOT NULL, # The date & time is stored in UTC
    PrevLoginDate    DATETIME        NULL, # The date & time is stored in UTC
    LastLoginDate    DATETIME        NULL  # The date & time is stored in UTC
) ENGINE = InnoDB CHARSET = UTF8;
INSERT INTO UserData
            (Name, RegistrationDate)
VALUES      ('Mr. A', DATE_SUB(UTC_TIMESTAMP(), INTERVAL 2 DAY)),
            ('Mr. B', DATE_SUB(UTC_TIMESTAMP(), INTERVAL 1 DAY));
```

```
CREATE TABLE UserAideData (
    UserID INT NOT NULL,
    AideID INT NOT NULL,
    INDEX Idx_UserID(UserID),
    INDEX Idx_AideID(AideID),
    FOREIGN KEY(UserID) REFERENCES UserData(UserID) ON UPDATE RESTRICT ON DELETE CASCADE,
    FOREIGN KEY(AideID) REFERENCES UserData(UserID) ON UPDATE RESTRICT ON DELETE CASCADE
) ENGINE = InnoDB CHARSET = UTF8;
```

The above statements will create a test database named `MyTestDB`. Note that if a **database with the same name** already exists, **it will be destroyed**. Therefore, please adjust the database name if you already have a database with the same name.

The sections in this tutorial are meant to be followed sequentially. It is possible to jump to a certain section directly, however, one must ensure that the needed tables contain the initial necessary records before proceeding. In the following sections, each SQL statement, together with the expected results, will be shown in a code block. SQL statements are prefixed with the MySQL prompt. You can copy and paste the statement (without the `mysql>` prompt) directly to your MySQL console. An example of MySQL welcome texts and prompt is shown below.

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.1.39 Source distribution

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

mysql>
```

# 3. Arithmetic

Before continuing, it is important to check if the needed table contains the necessary records by executing the statement below.

```
mysql> SELECT * FROM WeaponData;
+----------+-------+-------------+------+
| WeaponID | Name  | MedianPower | Span |
+----------+-------+-------------+------+
|        1 | Sword |          10 |    2 |
|        2 | Axe   |          15 |    5 |
+----------+-------+-------------+------+
2 rows in set (0.00 sec)
```

## *3.1. Counting the Number of Records*

The `COUNT()` function can be used to count the number of records that will be returned by a query. The simplest usage of the function is shown below.

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

The above statement will count the number of all the records, including records that contain `NULL` fields. To count only records that do not contain `NULL` in a particular field, one can pass the field as a parameter in the call to the `COUNT()` function.

```
mysql> SELECT COUNT(WeaponID) FROM WeaponData;
```

In case some of the fields may contain duplicated values, one can count only unique values by using the `DISTINCT` keyword.

```
mysql> SELECT COUNT(DISTINCT WeaponID) FROM WeaponData;
```

Due to the example table does not contain any `NULL` nor duplicated value, the results of the last two queries will be the same with the first query.

### *3.2. Minimum, Maximum, Average, and Total Value*

The `MIN()`, `MAX()`, `AVG()`, and `SUM()` functions can be used to find the minimum, maximum, average, and total values of a particular field in a query. They will exclude fields that contain `NULL`. The keyword `DISTINCT` can also be used to exclude duplicated values (although usually this is not something you want to do).

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

```
mysql> SELECT MAX(Span) FROM WeaponData;
+-----------+
| MAX(Span) |
+-----------+
|         5 |
+-----------+
1 row in set (0.00 sec)
```

```
mysql> SELECT AVG(Span) FROM WeaponData;
+-----------+
| AVG(Span) |
+-----------+
|    3.5000 |
+-----------+
1 row in set (0.00 sec)
```

```
mysql> SELECT SUM(Span) FROM WeaponData;
+-----------+
| SUM(Span) |
+-----------+
|         7 |
+-----------+
1 row in set (0.00 sec)
```

### 3.3. Custom Calculation

MySQL allows users to perform custom calculation in a query. One can add, subtract, etc. a field with another field, with a constant, etc. as shown in the example below.

```
mysql> SELECT Name,
              MedianPower - Span,
              MedianPower + Span,
              Span * 2
       FROM   WeaponData;
+-------+--------------------+--------------------+----------+
| Name  | MedianPower - Span | MedianPower + Span | Span * 2 |
+-------+--------------------+--------------------+----------+
| Sword |                  8 |                 12 |        4 |
| Axe   |                 10 |                 20 |       10 |
+-------+--------------------+--------------------+----------+
2 rows in set (0.00 sec)
```

It is also possible to add alias names to the resulting calculation for better clarity.

```
mysql> SELECT Name                   AS 'WeaponName',
              MedianPower - Span AS 'MinimumPower',
              MedianPower + Span AS 'MaximumPower',
              Span * 2           AS 'Interval'
       FROM   WeaponData;
+------------+--------------+--------------+----------+
| WeaponName | MinimumPower | MaximumPower | Interval |
+------------+--------------+--------------+----------+
| Sword      |            8 |           12 |        4 |
| Axe        |           10 |           20 |       10 |
+------------+--------------+--------------+----------+
2 rows in set (0.00 sec)
```

Please note that the texts in the two examples above have been slightly reformatted to ease copy and paste operations to your MySQL console. The console actually prefixes the second, third, etc. lines of a multi-line statement with `->` as shown in the example below.

```
mysql> SELECT Name,
    ->        MedianPower - Span,
    ->        MedianPower + Span,
    ->        Span * 2
    -> FROM   WeaponData;
...
```

## 4. Concatenate with Separator

Before continuing, it is important to check if the needed table contains the necessary records by executing the statement below.

```
mysql> SELECT * FROM WeaponData;
+----------+-------+-------------+------+
| WeaponID | Name  | MedianPower | Span |
+----------+-------+-------------+------+
|        1 | Sword |          10 |    2 |
|        2 | Axe   |          15 |    5 |
+----------+-------+-------------+------+
2 rows in set (0.00 sec)
```

The function `CONCAT_WS()` can be used to concatenate multiple fields and/or strings with custom separator. The function is prototyped as:

```
CONCAT_WS(separator, field1/string1, field2/string2, ...)
```

The function skips all `NULL` values after the separator. If the separator itself is `NULL`, the final result will be also `NULL`. The example below demonstrate the use of this function.

```
mysql> SELECT Name,
              CONCAT_WS(' +/- ', MedianPower, Span) AS 'PowerSpec'
       FROM   WeaponData;
+-------+-----------+
| Name  | PowerSpec |
+-------+-----------+
| Sword | 10 +/- 2  |
| Axe   | 15 +/- 5  |
+-------+-----------+
2 rows in set (0.00 sec)
```

If the separator is an empty string, the result is the same as calling the simpler version of the function, `CONCAT()` that does not support custom separator.

# 5. Converting `INSERT` to `UPDATE` for Existing Keys

Before continuing, it is important to check if the needed table contains the necessary records by executing the statement below.

```
mysql> SELECT * FROM WeaponData;
+----------+-------+-------------+------+
| WeaponID | Name  | MedianPower | Span |
+----------+-------+-------------+------+
|        1 | Sword |          10 |    2 |
|        2 | Axe   |          15 |    5 |
+----------+-------+-------------+------+
2 rows in set (0.00 sec)
```

Sometimes, it may be necessary to insert a record with a custom primary key to a table. Due to MySQL does not allow duplicated primary keys, one must ensure that the same key has not yet exist in the table or the insert will fail. Performing `SELECT` query and replacing the `INSERT` with an `UPDATE` if the key already exist will cause additional overhead in the application code. In some case, it may even impossible (or inefficient) to check if a record with a specific primary key is already in the table. There is another problem, using just the `UPDATE` statement also will not work. The `UPDATE` statement will not work on a non existent record. It would be

nice if MySQL can automatically convert an `INSERT` statement to an `UPDATE` statement in case the key already exist.

Assume that now we want to add a new weapon (with a custom primary key) to the `WeaponData` table by executing the statement below.

```
mysql> INSERT INTO WeaponData (WeaponID, Name, MedianPower, Span)
       VALUES                (3, 'Spear', 10, 5);
Query OK, 1 row affected (0.00 sec)
```

The query is a success. Display the whole content of the `WeaponData` table to verify the `INSERT`.

```
mysql> SELECT * FROM WeaponData;
+----------+-------+-------------+------+
| WeaponID | Name  | MedianPower | Span |
+----------+-------+-------------+------+
|        1 | Sword |          10 |    2 |
|        2 | Axe   |          15 |    5 |
|        3 | Spear |          10 |    5 |
+----------+-------+-------------+------+
3 rows in set (0.00 sec)
```

Now, suppose we want to add another weapon (still a `Spear`, but with a higher `MedianPower`). The problem is that the game rule defines that there can be only one `Spear` in the entire game (a bit unrealistic, but just assume that this is the rule). Deleting the old `Spear` and inserting a new one (with the same `WeaponID`) would not work. If the `WeaponID` has been referenced in other tables, the `ON DELETE CASCADE` foreign-key-constraint rule will also delete all records (in all tables!) that reference to the `Spear`. Executing the `INSERT` query again (albeit with a different `MedianPower`) will also not work.

```
mysql> INSERT INTO WeaponData (WeaponID, Name, MedianPower, Span)
       VALUES                (3, 'Spear', 12, 5);
ERROR 1062 (23000): Duplicate entry '3' for key 'PRIMARY'
```

Fortunately, it is possible to instruct MySQL to convert an `INSERT` statement to an `UPDATE` statement by adding an `ON DUPLICATE KEY UPDATE` clause. Therefore, it is possible to construct one statement that both insert and update a record. The example below demonstrate this feature.

```
mysql> INSERT INTO WeaponData (WeaponID, Name, MedianPower, Span)
       VALUES                (3, 'Spear', 12, 5)
       ON DUPLICATE KEY UPDATE Name        = VALUES(Name),
                               MedianPower = VALUES(MedianPower),
                               Span        = VALUES(Span);
Query OK, 0 rows affected (0.00 sec)
```

The query is a success. Display the whole content of the `WeaponData` table to verify it.

```
mysql> SELECT * FROM WeaponData;
+----------+-------+-------------+------+
| WeaponID | Name  | MedianPower | Span |
+----------+-------+-------------+------+
|        1 | Sword |          10 |    2 |
|        2 | Axe   |          15 |    5 |
|        3 | Spear |          12 |    5 |
+----------+-------+-------------+------+
3 rows in set (0.00 sec)
```

# 6. Date and Time

Basically, date/time-related data types and functions in MySQL can be grouped in two major groups:

- Data types and functions that are *affected* by the current time zone setting of MySQL. In this group, date and time data are stored in the server's local time zone.
- Data types and functions that are *not affected* by the current time zone setting of MySQL. In this group, date and time data are stored in UTC (Coordinated Universal Time). UTC has replaced the GMT (Greenwich Mean Time) from January 1, 1972.

According to the MySQL manual, only the data type `TIMESTAMP` and the functions `NOW()` and `CURTIME()` that are affected by the current time zone setting; all other data types and functions are working in UTC. Setting the time zone for the current MySQL connection can be done by setting the system variable `time_zone`. The example below shows how to set the time offset to zero (which basically setting it to UTC).

```
mysql> SET time_zone = '+0:00';
Query OK, 0 rows affected (0.00 sec)
```

While it is possible to always set the time zone to UTC anytime a new MySQL connection is created, it will be a good idea to use the second group of data types and functions that work in UTC. Generally, if the application will need to display date and time in the client's (browser's) time zone, it is better to store the date and time as UTC in the database. If the date and time are to be used for internal reference only, then it does not matter much.

Before continuing, it is important to check if the needed table contains the necessary records by executing the statement below. Note that, you will most likely to get different date and time for most of the examples.

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+---------------+---------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate | LastLoginDate |
+--------+-------+---------------------+---------------+---------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | NULL          | NULL          |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL          | NULL          |
+--------+-------+---------------------+---------------+---------------+
2 rows in set (0.00 sec)
```

## *6.1. Current Date and/or Time in UTC*

The example below shows how to get the current date and/or time in UTC. Those three functions can be used in almost any MySQL query.

```
mysql> SELECT UTC_DATE(), UTC_TIME(), UTC_TIMESTAMP();
+------------+------------+---------------------+
| UTC_DATE() | UTC_TIME() | UTC_TIMESTAMP()     |
+------------+------------+---------------------+
| 2010-07-15 | 02:55:31   | 2010-07-15 02:55:31 |
+------------+------------+---------------------+
1 row in set (0.00 sec)
```

Now, add a new record in the `UserData` table.

```
mysql> INSERT INTO UserData (Name, RegistrationDate)
       VALUES               ('Mr. C', UTC_TIMESTAMP());
Query OK, 1 row affected (0.00 sec)
```

The query is a success. Display the whole content of the table to verify it. Confirm that the `RegistrationDate` of the new user has been set to the current date and time (in UTC).

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+---------------+---------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate | LastLoginDate |
+--------+-------+---------------------+---------------+---------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | NULL          | NULL          |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL          | NULL          |
|      3 | Mr. C | 2010-07-15 02:55:44 | NULL          | NULL          |
+--------+-------+---------------------+---------------+---------------+
3 rows in set (0.00 sec)
```

## *6.2. Converting Date and Time Values to Plain Numeric Values*

Sometimes, we want to perform calculation on a date and/or time values. Due to our application programming/scripting language may use different date and time format than MySQL, it may be better to convert the date and/or time values into a plain numeric values. The `TO_DAYS()` function converts a date value to the number of days since year 0 (note that it is not intended for use with values before the advent of the Gregorian calendar in 1582 due to changes in calendar system). The `UNIX_TIMESTAMP()` function converts a date or date-time value to the number of seconds since `'1970-01-01 00:00:00' UTC`. The example below demonstrate these two functions.

```
mysql> SELECT Name,
              TO_DAYS       (RegistrationDate),
              UNIX_TIMESTAMP(RegistrationDate)
       FROM   UserData;
+-------+--------------------------+----------------------------------+
| Name  | TO_DAYS(RegistrationDate) | UNIX_TIMESTAMP(RegistrationDate) |
+-------+--------------------------+----------------------------------+
| Mr. A |                   734331 |                       1278989700 |
| Mr. B |                   734332 |                       1279076100 |
| Mr. C |                   734333 |                       1279162544 |
+-------+--------------------------+----------------------------------+
3 rows in set (0.00 sec)
```

## *6.3. Calculation/Manipulation on Date and Time Values*

As shown as in the previous section, we can convert the date and time values to plain numeric values and operating on the converted values. However, MySQL does provide a quite powerful date and time calculation/manipulation functions.

The example below demonstrates how to calculate difference (in days) between the current date and the `RegistrationDate` (both in UTC).

```
mysql> SELECT Name, DATEDIFF(UTC_TIMESTAMP(), RegistrationDate) FROM UserData;
+-------+--------------------------------------------+
| Name  | DATEDIFF(UTC_TIMESTAMP(), RegistrationDate) |
+-------+--------------------------------------------+
| Mr. A |                                          2 |
| Mr. B |                                          1 |
| Mr. C |                                          0 |
+-------+--------------------------------------------+
3 rows in set (0.00 sec)
```

Suppose we want to display date and time in the client's (browser's) time zone. What can we do to achieve this? For start, we can find out the browser's time zone offset using JavaScript code such as:

```
(new Date()).getTimezoneOffset() * 60
```

that will return the offset in seconds. Next, we can use MySQL's date and time manipulation function to adjust the date and time. The example below shows how to do this. Assume that the client's (browser's) location has +8 hours (28.800 seconds) differences with UTC.

```
mysql> SELECT Name, DATE_ADD(RegistrationDate, INTERVAL 28800 SECOND) FROM UserData;
+-------+---------------------------------------------------+
| Name  | DATE_ADD(RegistrationDate, INTERVAL 28800 SECOND) |
+-------+---------------------------------------------------+
| Mr. A | 2010-07-13 10:55:00                               |
| Mr. B | 2010-07-14 10:55:00                               |
| Mr. C | 2010-07-15 10:55:44                               |
+-------+---------------------------------------------------+
3 rows in set (0.00 sec)
```

Some of the available `INTERVAL` modifiers are: `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, and `YEAR`. The `DATE_SUB()` function can also be used as demonstrated below.

```
mysql> SELECT Name, DATE_SUB(RegistrationDate, INTERVAL -28800/3600 HOUR) FROM UserData;
+-------+-------------------------------------------------------+
| Name  | DATE_SUB(RegistrationDate, INTERVAL -28800/3600 HOUR) |
+-------+-------------------------------------------------------+
| Mr. A | 2010-07-13 10:55:00                                   |
| Mr. B | 2010-07-14 10:55:00                                   |
| Mr. C | 2010-07-15 10:55:44                                   |
+-------+-------------------------------------------------------+
3 rows in set (0.00 sec)
```

# 7. Group Concatenate

Before continuing, it is important to check if the needed tables contains the necessary records by executing the statements below.

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+----------------+---------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate  | LastLoginDate |
+--------+-------+---------------------+----------------+---------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | NULL           | NULL          |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL           | NULL          |
|      3 | Mr. C | 2010-07-15 02:55:44 | NULL           | NULL          |
+--------+-------+---------------------+----------------+---------------+
3 rows in set (0.00 sec)
```

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

Now, we need to insert some more records for this section of the tutorial.

```
mysql> INSERT INTO UserData (Name, RegistrationDate)
       VALUES               ('Mr. D', UTC_TIMESTAMP());
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO UserAideData(UserID, AideID)
       VALUES                  (1, 2), (1, 3), (2, 4);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Verify that the records are really inserted.

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+----------------+---------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate  | LastLoginDate |
+--------+-------+---------------------+----------------+---------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | NULL           | NULL          |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL           | NULL          |
|      3 | Mr. C | 2010-07-15 02:55:44 | NULL           | NULL          |
|      4 | Mr. D | 2010-07-15 03:18:32 | NULL           | NULL          |
+--------+-------+---------------------+----------------+---------------+
4 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM UserAideData;
+--------+--------+
| UserID | AideID |
+--------+--------+
|      1 |      2 |
|      1 |      3 |
|      2 |      4 |
+--------+--------+
3 rows in set (0.00 sec)
```

From the content of the `UserAideData` table, we find out that user `1` has two aides: user `2` and `3`. Using group concatenate, we can present the content of the table in a more “presentable” way as shown in the example below.

```
mysql> SELECT   UserID, GROUP_CONCAT(AideID)
       FROM     UserAideData
       GROUP BY UserID;
+--------+----------------------+
| UserID | GROUP_CONCAT(AideID) |
+--------+----------------------+
|      1 | 2,3                  |
|      2 | 4                    |
+--------+----------------------+
2 rows in set (0.00 sec)
```

Using custom separator is also possible.

```
mysql> SELECT   UserID, GROUP_CONCAT(AideID SEPARATOR ':')
       FROM     UserAideData
       GROUP BY UserID;
+--------+------------------------------------+
| UserID | GROUP_CONCAT(AideID SEPARATOR ':') |
+--------+------------------------------------+
|      1 | 2:3                                |
|      2 | 4                                  |
+--------+------------------------------------+
2 rows in set (0.00 sec)
```

The `GROUP BY` clauses are needed to tell MySQL about how to group and concatenate the values. More examples utilizing the `GROUP_CONCAT()` function will be also shown in the following sections.

# 8. Nested Query

Before continuing, it is important to check if the needed tables contains the necessary records by executing the statements below.

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+---------------+---------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate | LastLoginDate |
+--------+-------+---------------------+---------------+---------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | NULL          | NULL          |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL          | NULL          |
|      3 | Mr. C | 2010-07-15 02:55:44 | NULL          | NULL          |
|      4 | Mr. D | 2010-07-15 03:18:32 | NULL          | NULL          |
+--------+-------+---------------------+---------------+---------------+
4 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM UserAideData;
+--------+--------+
| UserID | AideID |
+--------+--------+
|      1 |      2 |
|      1 |      3 |
|      2 |      4 |
+--------+--------+
3 rows in set (0.00 sec)
```

## 8.1. Nested SELECT Query

Suppose we want to list the name of the users who are not the aide of user `1`. Instead of selecting all users from the `UserAideData` table that are not the aide of user `1` and then looking up their names using the `UserData` table, we could use a nested query.

```
mysql> SELECT UserID, Name
       FROM   UserData
       WHERE  UserID != 1
       AND    UserID NOT IN ( SELECT AideID
                              FROM   UserAideData
                              WHERE  UserID = 1
                            );
+--------+-------+
| UserID | Name  |
+--------+-------+
|      4 | Mr. D |
+--------+-------+
1 row in set (0.00 sec)
```

Note that the `UserID 1` will need to be mentioned twice in the query. The next example shows the same query but with different `UserID`.

```
mysql> SELECT UserID, Name
       FROM   UserData
       WHERE  UserID != 2
       AND    UserID NOT IN ( SELECT AideID
                              FROM   UserAideData
                              WHERE  UserID = 2
                            );
+--------+-------+
| UserID | Name  |
+--------+-------+
|      1 | Mr. A |
|      3 | Mr. C |
+--------+-------+
2 rows in set (0.00 sec)
```

Note that the above statements are not necessarily the best solutions. Using other method, such as inner join, may actually produce a more efficient query.

## 8.2. Nested INSERT Query

Using the result of a `SELECT` query in an `INSERT` query is also supported. However, one must ensure that the `SELECT` query only produces one result or the error message:

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

will be returned.

```
mysql> INSERT INTO UserAideData (UserID, AideID)
       VALUES      (2, ( SELECT UserID
                         FROM   UserData
                         WHERE  Name = 'Mr. C'
                         LIMIT 1
                       )
                   );
Query OK, 1 row affected (0.04 sec)
```

Verify that now user `2` will have a new aide (`Mr. C` with `UserID 3`).

```
mysql> SELECT * FROM UserAideData;
+--------+--------+
| UserID | AideID |
+--------+--------+
|      1 |      2 |
|      1 |      3 |
|      2 |      4 |
|      2 |      3 |
+--------+--------+
4 rows in set (0.00 sec)
```

## *8.3. Nested UPDATE Query*

Using the result of a `SELECT` query in an `UPDATE` query is also supported. However, if the source and destination tables are the same, the query will become not so direct as show in the example below.

```
mysql> UPDATE UserData
       SET    LastLoginDate = UTC_TIMESTAMP(),
              PrevLoginDate = ( SELECT LastLoginDate
                                FROM   UserData
                                WHERE  UserID = 1
                              )
       WHERE  UserID = 1;
ERROR 1093 (HY000): You can't specify target table 'UserData' for update in FROM clause
```

MySQL needs a temporary table to store the original value.

```
mysql> UPDATE UserData
       SET    LastLoginDate = UTC_TIMESTAMP(),
              PrevLoginDate = ( SELECT LastLoginDate
                                FROM   ( SELECT LastLoginDate
                                         FROM   UserData
                                         WHERE  UserID = 1
                                         LIMIT  1
                                       ) AS ORIGINAL
                              )
       WHERE  UserID = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The name of the temporary table does not have to be `ORIGINAL`, it can be anything as long as it is unique in the query. The above query will copy the value of the `LastLoginDate` field to the `PrevLoginDate` field before updating the `LastLoginDate` field to the current date and time. Verify the result by executing the statement below.

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+---------------+---------------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate | LastLoginDate       |
+--------+-------+---------------------+---------------+---------------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | NULL          | 2010-07-15 03:43:45 |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL          | NULL                |
|      3 | Mr. C | 2010-07-15 02:55:44 | NULL          | NULL                |
|      4 | Mr. D | 2010-07-15 03:18:32 | NULL          | NULL                |
+--------+-------+---------------------+---------------+---------------------+
4 rows in set (0.00 sec)
```

If the same `UPDATE` query is executed again, the values will be shifted and updated. Try and verify it!

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+---------------------+---------------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate       | LastLoginDate       |
+--------+-------+---------------------+---------------------+---------------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | 2010-07-15 03:43:45 | 2010-07-15 03:47:02 |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL                | NULL                |
|      3 | Mr. C | 2010-07-15 02:55:44 | NULL                | NULL                |
|      4 | Mr. D | 2010-07-15 03:18:32 | NULL                | NULL                |
+--------+-------+---------------------+---------------------+---------------------+
4 rows in set (0.00 sec)
```

# 9. Inner Join

Before continuing, it is important to check if the needed tables contains the necessary records by executing the statements below.

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+---------------------+---------------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate       | LastLoginDate       |
+--------+-------+---------------------+---------------------+---------------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | 2010-07-15 03:43:45 | 2010-07-15 03:47:02 |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL                | NULL                |
|      3 | Mr. C | 2010-07-15 02:55:44 | NULL                | NULL                |
|      4 | Mr. D | 2010-07-15 03:18:32 | NULL                | NULL                |
+--------+-------+---------------------+---------------------+---------------------+
4 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM UserAideData;
+--------+--------+
| UserID | AideID |
+--------+--------+
|      1 |      2 |
|      1 |      3 |
|      2 |      4 |
|      2 |      3 |
+--------+--------+
4 rows in set (0.00 sec)
```

Using nested queries are sometimes not the best solution. Using other method, such as inner join, may actually produce a more efficient query. Inner join can be used to retrieve data from multiple tables that match the join conditions. The example below demonstrate an inner join between two tables.

```
mysql> SELECT     UD.Name   AS 'UserName',
                  AD.UserID AS 'AideID'
       FROM       UserData     UD
       INNER JOIN UserAideData AD USING(UserID);
+----------+--------+
| UserName | AideID |
+----------+--------+
| Mr. A    |      1 |
| Mr. A    |      1 |
| Mr. B    |      2 |
| Mr. B    |      2 |
+----------+--------+
4 rows in set (0.00 sec)
```

In the above example, the inner join clause:

```
INNER JOIN UserAideData AD USING(UserID)
```

connect the `UserAideData` table to the `UserData` table using `UserID` as the join condition. The clause:

```
USING(UserID)
```

is a shortcut for:

```
ON UD.UserID = AD.UserID
```

The condition will cause MySQL to return records from tables `UserData` and `UserAideData` with matching `UserID`.

The symbol `UD` and `AD` are table aliases. The original form of the above query (without table aliases) would be:

```
SELECT     UserData    .Name   AS 'UserName',
           UserAideData.UserID AS 'AideID'
FROM       UserData
INNER JOIN UserAideData ON UserData.UserID = UserAideData.UserID;
```

Using table aliases allows developer to write shorter queries. Using table aliases also allow the same table to be referenced more than once in a single query. The example below demonstrate a multiple inner join between the two tables.

```
mysql> SELECT     UD.Name AS 'UserName',
                  DD.Name AS 'AideName'
       FROM       UserData     UD
       INNER JOIN UserAideData AD USING(UserID)
       INNER JOIN UserData     DD ON AD.AideID = DD.UserID;
+----------+----------+
| UserName | AideName |
+----------+----------+
| Mr. A    | Mr. B    |
| Mr. A    | Mr. C    |
| Mr. B    | Mr. D    |
| Mr. B    | Mr. C    |
+----------+----------+
4 rows in set (0.00 sec)
```

## *9.1. Group Concatenate in Inner Join*

Group concatenate is also permitted in inner join. The example below demonstrates the use of group concatenate with inner join to produce a more “presentable” result for the same query. It even possible to perform sorting inside a group concatenate, as shown in the next example.

```
mysql> SELECT     UD.Name                  AS 'UserName',
                  GROUP_CONCAT(DD.Name) AS 'Aides'
       FROM       UserData      UD
       INNER JOIN UserAideData AD USING(UserID)
       INNER JOIN UserData      DD ON AD.AideID = DD.UserID
       GROUP BY   UD.UserID;
+----------+-------------+
| UserName | Aides       |
+----------+-------------+
| Mr. A    | Mr. B,Mr. C |
| Mr. B    | Mr. D,Mr. C |
+----------+-------------+
2 rows in set (0.00 sec)
```

```
mysql> SELECT     UD.Name                                       AS 'UserName',
                  GROUP_CONCAT(DD.Name ORDER BY DD.UserID ASC) AS 'Aides'
       FROM       UserData      UD
       INNER JOIN UserAideData AD USING(UserID)
       INNER JOIN UserData      DD ON AD.AideID = DD.UserID
       GROUP BY   UD.UserID;
+----------+-------------+
| UserName | Aides       |
+----------+-------------+
| Mr. A    | Mr. B,Mr. C |
| Mr. B    | Mr. C,Mr. D |
+----------+-------------+
2 rows in set (0.00 sec)
```

Please remember that a `GROUP BY` clause must always present to tell MySQL about how to group and concatenate the values. Without this clause, the result will be incorrect as shown in the example below.

```
mysql> SELECT     UD.Name                                   AS 'UserName',
                  GROUP_CONCAT(DD.Name ORDER BY DD.UserID) AS 'Aides'
       FROM       UserData      UD
       INNER JOIN UserAideData AD USING(UserID)
       INNER JOIN UserData      DD ON AD.AideID = DD.UserID;
+----------+-------------------------+
| UserName | Aides                   |
+----------+-------------------------+
| Mr. A    | Mr. B,Mr. C,Mr. C,Mr. D |
+----------+-------------------------+
1 row in set (0.00 sec)
```

However, for some case the `GROUP BY` clause can be omitted.

```
mysql> SELECT     UD.Name                                   AS 'UserName',
                  GROUP_CONCAT(DD.Name ORDER BY DD.UserID) AS 'Aides'
       FROM       UserData      UD
       INNER JOIN UserAideData AD USING(UserID)
       INNER JOIN UserData      DD ON AD.AideID = DD.UserID
       WHERE      UD.UserID = 1;
+----------+-------------+
| UserName | Aides       |
+----------+-------------+
| Mr. A    | Mr. B,Mr. C |
+----------+-------------+
1 row in set (0.00 sec)
```

In the above example, there will be at most one records (`UserID` is unique). Therefore, there is not need to specify a `GROUP BY` clause.

## *9.2. Basic Analysis of Inner Join*

Some inner join can be a very complicated query. Complicated queries would cause performance degradation in the application. Therefore, it is important to find out how MySQL will execute a particular query. Adding the clause `EXPLAIN EXTENDED` will tell MySQL to show about how it would execute the query. Ending the statement with `\G` (instead of semi-colon) causes MySQL to produce vertical output.

```
mysql> EXPLAIN EXTENDED
       SELECT     UD.Name                                        AS 'UserName',
                  GROUP_CONCAT(DD.Name ORDER BY DD.UserID ASC) AS 'Aides'
       FROM       UserData     UD
       INNER JOIN UserAideData AD USING(UserID)
       INNER JOIN UserData     DD ON AD.AideID = DD.UserID
       GROUP BY   UD.UserID \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: UD
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 100.00
        Extra: Using temporary; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: AD
         type: ALL
possible_keys: Idx_UserID,Idx_AideID
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 75.00
        Extra: Using where; Using join buffer
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: DD
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: MyTestDB.AD.AideID
         rows: 1
     filtered: 100.00
        Extra:
3 rows in set, 1 warning (0.00 sec)
```

The above output shows information about how MySQL would execute the query. There are two main points of interest in the above result:

- The text `'type: ALL'` for the `UD` and `AD` tables indicate that MySQL would need to process trough all the records of both tables. Basically, it is not a good idea to process all the records from a table, especially if the table is large (contains many records) and not properly indexed.

- The text `'Using temporary; Using filesort'` from the `UD` table indicates that MySQL would need to use an internal temporary table that may be stored on disk. If the table is small, the operating system may never create a physical file for the temporary table (all fit in memory). However, if the table is large (contains many records), the operating system would have no choice except to store the data in physical file/disk. This may degrade performance.

The optimized form of the query can be displayed by showing the warning (the output has been slightly reformatted for clarity).

```
mysql> SHOW WARNINGS \G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: select   `MyTestDB`.`UD`.`Name` AS `UserName`,
                  group_concat(`MyTestDB`.`DD`.`Name` order by `MyTestDB`.`DD`.`UserID` ASC
                               separator ',') AS `Aides`
         from     `MyTestDB`.`UserData` `UD` join `MyTestDB`.`UserAideData` `AD`
         join     `MyTestDB`.`UserData` `DD`
         where    ((`MyTestDB`.`AD`.`UserID` = `MyTestDB`.`UD`.`UserID`)
         and      (`MyTestDB`.`DD`.`UserID` = `MyTestDB`.`AD`.`AideID`))
         group by `MyTestDB`.`UD`.`UserID`
1 row in set (0.00 sec)
```

The example below shows how MySQL query will execute the same query if the developer limits the possible number of results by using explicit `WHERE` clause.

```
mysql> EXPLAIN EXTENDED
       SELECT     UD.Name                                   AS 'UserName',
                  GROUP_CONCAT(DD.Name ORDER BY DD.UserID) AS 'Aides'
       FROM       UserData     UD
       INNER JOIN UserAideData AD USING(UserID)
       INNER JOIN UserData     DD ON AD.AideID = DD.UserID
       WHERE      UD.UserID = 1 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: UD
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
     filtered: 100.00
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: AD
         type: ref
possible_keys: Idx_UserID,Idx_AideID
          key: Idx_UserID
      key_len: 4
          ref: const
         rows: 2
     filtered: 100.00
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: DD
```

```
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: MyTestDB.AD.AideID
         rows: 1
     filtered: 100.00
        Extra:
3 rows in set, 1 warning (0.00 sec)
```

There is no more temporary table nor *filesort*. The optimized form of the query can be displayed by showing the warning (the output has been slightly reformatted for clarity).

```
mysql> SHOW WARNINGS \G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: select 'Mr. A' AS `UserName`,
                group_concat(`MyTestDB`.`DD`.`Name` order by `MyTestDB`.`DD`.`UserID` ASC
                             separator ',') AS `Aides`
          from  `MyTestDB`.`UserData` `UD`
          join  `MyTestDB`.`UserAideData` `AD`
          join  `MyTestDB`.`UserData` `DD`
          where ((`MyTestDB`.`DD`.`UserID` = `MyTestDB`.`AD`.`AideID`)
          and   (`MyTestDB`.`AD`.`UserID` = 1))
1 row in set (0.00 sec)
```

If possible, try to avoid selecting all records from a table and *filesort*.

# 10. Outer Join

Before continuing, it is important to check if the needed tables contains the necessary records by executing the statements below.

```
mysql> SELECT * FROM UserData;
+--------+-------+---------------------+---------------------+---------------------+
| UserID | Name  | RegistrationDate    | PrevLoginDate       | LastLoginDate       |
+--------+-------+---------------------+---------------------+---------------------+
|      1 | Mr. A | 2010-07-13 02:55:00 | 2010-07-15 03:43:45 | 2010-07-15 03:47:02 |
|      2 | Mr. B | 2010-07-14 02:55:00 | NULL                | NULL                |
|      3 | Mr. C | 2010-07-15 02:55:44 | NULL                | NULL                |
|      4 | Mr. D | 2010-07-15 03:18:32 | NULL                | NULL                |
+--------+-------+---------------------+---------------------+---------------------+
4 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM UserAideData;
+--------+--------+
| UserID | AideID |
+--------+--------+
|      1 |      2 |
|      1 |      3 |
|      2 |      4 |
|      2 |      3 |
+--------+--------+
4 rows in set (0.00 sec)
```

Inner join only returns records from both tables that exactly match the join condition. On the other hand, outer join (depends on the type) will replace non-matching values with `NULL`. MySQL supports `LEFT` and `RIGHT` outer join. Left join return all rows from the left-table, even if

there are no matches in the right-table (the values are replaced with `NULL`). Right join return all rows from the right-table, even if there are no matches in the left-table (the values are replaced with `NULL`).

The example below demonstrate a left join between two tables. The two bosses (users `Mr. C` and `Mr. D`) do not have any aide. Therefore the `AideID` for those two users are `NULL`.

```
mysql> SELECT    UD.Name AS 'BossName',
                 AD.AideID
       FROM      UserData     UD
       LEFT JOIN UserAideData AD USING(UserID);
+----------+--------+
| BossName | AideID |
+----------+--------+
| Mr. A    |      2 |
| Mr. A    |      3 |
| Mr. B    |      4 |
| Mr. B    |      3 |
| Mr. C    |   NULL |
| Mr. D    |   NULL |
+----------+--------+
6 rows in set (0.00 sec)
```

The example below demonstrate a right join between two tables. The two bosses (users `Mr. C` and `Mr. D`) do not have any aide. Therefore the `AideID` for those two users are `NULL`.

```
mysql> SELECT     AD.AideID,
                  UD.Name AS 'BossName'
       FROM       UserAideData AD
       RIGHT JOIN UserData     UD USING(UserID);
+--------+----------+
| AideID | BossName |
+--------+----------+
|      2 | Mr. A    |
|      3 | Mr. A    |
|      4 | Mr. B    |
|      3 | Mr. B    |
|   NULL | Mr. C    |
|   NULL | Mr. D    |
+--------+----------+
6 rows in set (0.00 sec)
```

The group concatenate can also be used together with outer join as shown as in the example below. Do not forget to supply a `GROUP BY` clause when it is needed.

```
mysql> SELECT    UD.Name                                      AS 'BossName',
                 GROUP_CONCAT(DD.Name ORDER BY DD.UserID ASC) AS 'AideList'
       FROM      UserData     UD
       LEFT JOIN UserAideData AD USING(UserID)
       LEFT JOIN UserData     DD ON AD.AideID = DD.UserID
       GROUP BY  UD.UserID;
+----------+-------------+
| BossName | AideList    |
+----------+-------------+
| Mr. A    | Mr. B,Mr. C |
| Mr. B    | Mr. C,Mr. D |
| Mr. C    | NULL        |
| Mr. D    | NULL        |
+----------+-------------+
4 rows in set (0.00 sec)
```

The final examples below shows how to use the `COUNT()` function together with left/right join to count the number of aides of a particular user.

```
mysql> SELECT     UD.Name           AS 'UserName',
                  COUNT(AD.UserID) AS 'TotalAides'
       FROM       UserData      UD
       LEFT JOIN UserAideData AD USING(UserID)
       GROUP BY   UD.UserID;
```

```
mysql> SELECT      UD.Name           AS 'UserName',
                   COUNT(AD.UserID) AS 'TotalAides'
       FROM        UserAideData AD
       RIGHT JOIN UserData      UD USING(UserID)
       GROUP BY    UD.UserID;
```

Note that the both queries should produce exactly the same result.

```
+----------+------------+
| UserName | TotalAides |
+----------+------------+
| Mr. A    |          2 |
| Mr. B    |          2 |
| Mr. C    |          0 |
| Mr. D    |          0 |
+----------+------------+
4 rows in set (0.00 sec)
```

In our MySQL server (version 5.1.39), `EXPLAIN EXTENDED` followed by `SHOW WARNINGS` even produce the same analysis and optimized query.

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: UD
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 100.00
        Extra: Using temporary; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: AD
         type: ref
possible_keys: Idx_UserID
          key: Idx_UserID
      key_len: 4
          ref: MyTestDB.UD.UserID
         rows: 1
     filtered: 100.00
        Extra: Using index
2 rows in set, 1 warning (0.00 sec)
```

```
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: select    `MyTestDB`.`UD`.`Name` AS `UserName`,
                   count(`MyTestDB`.`AD`.`UserID`) AS `TotalAides`
         from      `MyTestDB`.`UserData` `UD`
         left join `MyTestDB`.`UserAideData` `AD` on ((`MyTestDB`.`UD`.`UserID` =
                                                    `MyTestDB`.`AD`.`UserID`))
         where     1
         group by  `MyTestDB`.`UD`.`UserID`
1 row in set (0.00 sec)
```

Note that the output of above has been slightly reformatted for clarity.

# References

http://en.wikipedia.org/wiki/MySQL, June 09, 2010

http://dev.mysql.com/doc/refman/5.1/en, July 12, 2010

http://www.xaprb.com/blog/2005, July 12, 2010

http://www.xaprb.com/blog/2006, July 12, 2010

http://en.wikipedia.org/wiki/Coordinated_Universal_Time, July 15, 2010

http://en.wikipedia.org/wiki/Greenwich_Mean_Time, July 15, 2010