Date Arithmetic With MySQL

By icarus

2003−07−03

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/MySQL/DateArithmetic/

Not My Type

If you're familiar with MySQL, one of the planet's most popular open−source RDBMS, you already know that it supports a wide variety of data types for numbers, strings and dates. These data types perform two very important
functions: they enforce consistency on the data in a MySQL table (ensuring, for example, that numeric columns do not contain character data) and they optimize the space (bytes) required to store each type of data.

Now, MySQL comes with almost different data types, including types for integers, floating−point numbers, strings, date and time values, and data collections. One of the larger categories among these is the one containing date and time types, primarily because MySQL includes date and time types to meet almost every need you could think of. For example, there's the TIMESTAMP type to store timestamps, the DATE and TIME types to store just dates or times, the hybrid DATETIME type to store both, and the YEAR type to store the year component of a date.

Of course, data types, by themselves are only one piece of the puzzle; in order to do something with them, you need functions. And MySQL scores high points there as well, providing over 40 built−in functions to process and manipulate date and time values. The MySQL date and time API includes functions to extract different components of a timestamp, to format a timestamp in a variety of different ways, to obtain the current date and time, to convert between different date and time formats, and much, much more.

Sadly, this article isn't going to examine the complete date and time API in MySQL; instead, it's going to focus on a very small subset of this API, the functions related to performing date and time arithmetic. Flip the page, and let's get started.

When Two And Two Don't Make Four

When dealing with temporal data, one of the more common (and complex) tasks involves performing addition and subtraction operations on date and time values. Consider, for example, the simple task of calculating a date 91 days hence. Usually, in order to do this with any degree of precision, you need to factor in a number of different variables: the month you're in, the number of days in that month, the number of days in the months following, whether or not the current year is a leap year, and so on.

Writing code to perform such calculations quickly becomes both tedious and complicated. What is really needed in such situations is a date API that supports such date arithmetic, one that takes care of the numerous minor adjustments that have to be made when adding and subtracting intervals to date and time values.

Fortunately, MySQL comes with just such an API, in the form of six functions designed specifically to perform calculations on date and time values. Here they are:

TO_DAYS() − calculates the day number corresponding to a specific date

FROM_DAYS() − calculates the date corresponding to a day number

DATE_ADD() − adds a specified interval to a date and returns a new date

DATE_SUB() − subtracts a specified interval from a date and returns a new date

PERIOD_DIFF() − calculates the difference (in months) between two dates

PERIOD_ADD() − adds an interval (in months) to a date and returns a new date

Let's take a closer look.

Counting Down

The first function in the list, the TO_DAYS() function, returns a number corresponding to a specific date. This number is calculated as the number of days elapsed between year 0 and the specified date. Consider the following examples, which illustrate:

```
mysql> SELECT TO_DAYS('2003−04−06');

+−−−−−−−−−−−−−−−−−−−−−−−+

| TO_DAYS('2003−04−06') |

+−−−−−−−−−−−−−−−−−−−−−−−+

| 731676 |

+−−−−−−−−−−−−−−−−−−−−−−−+

1 row in set (0.04 sec)
```

The input value to the TO_DAYS() function may be a date in either string ("YYYY−MM−DD") or numeric (YYYYMMDD) format. The following example is equivalent to the one above:

```
mysql> SELECT TO_DAYS(20030406);

+−−−−−−−−−−−−−−−−−−−+

| TO_DAYS(20030406) |

+−−−−−−−−−−−−−−−−−−−+

| 731676 |

+−−−−−−−−−−−−−−−−−−−+

1 row in set (0.01 sec)
```

You can obtain the current day number with the addition of the very useful NOW() command:

```
mysql> SELECT TO_DAYS(NOW());

+----------------+

| TO_DAYS(NOW()) |

+----------------+

| 731756 |

+----------------+

1 row in set (0.03 sec)
```

The number returned by the TO_DAYS() function can be easily converted back to a human−readable date with the FROM_DAYS() function, which accepts a day number and returns the corresponding date value. Consider the following examples, which demonstrate:

```
mysql> SELECT FROM_DAYS(731756);

+-------------------+

| FROM_DAYS(731756) |

+-------------------+

| 2003−06−25 |

+-------------------+

1 row in set (0.01 sec)
```

```
mysql> SELECT FROM_DAYS(849302);
+−−−−−−−−−−−−−−−−−−−+
| FROM_DAYS(849302) |
+−−−−−−−−−−−−−−−−−−−+
| 2325−04−24 |
+−−−−−−−−−−−−−−−−−−−+
1 row in set (0.00 sec)
```

```
mysql> SELECT FROM_DAYS(TO_DAYS('1999−05−14'));
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| FROM_DAYS(TO_DAYS('1999−05−14')) |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 1999−05−14 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.00 sec)
```

```
mysql> SELECT FROM_DAYS(0);
+−−−−−−−−−−−−−−+
| FROM_DAYS(0) |
+−−−−−−−−−−−−−−+
| 0000−00−00 |
+−−−−−−−−−−−−−−+
1 row in set (0.00 sec)
```

As you may have guessed, the TO_DAYS() and FROM_DAYS() functions make it very easy to execute the example alluded to in the introduction of this article − adding 91 days to a date value and obtaining the resulting value
− since they automatically adjust for the number of years in a specific month. Consider the following examples, which illustrate by adding 1 day to the last day of February in a leap and non−leap year:

```
mysql> SELECT FROM_DAYS(TO_DAYS('2004−02−28') + 1);

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| FROM_DAYS(TO_DAYS('2004−02−28') + 1) |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| 2004−02−29 |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

1 row in set (0.00 sec)
```

```
mysql> SELECT FROM_DAYS(TO_DAYS('2003−02−28') + 1);
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| FROM_DAYS(TO_DAYS('2003−02−28') + 1) |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 2003−03−01 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.00 sec)
```

It's important to note that the TO_DAYS() and FROM_DAYS() functions do not support dates preceding the year 1582. In case you're wondering why, that was the year Pope Gregory XIII introduced the modern Gregorian calendar to replace the previous Julian calendar. As a result of switching calendars, many countries "lost" 10 or more days. The TO_DAYS() and FROM_DAYS() do not take into account these lost days, and so will return inaccurate results for such dates − as clearly illustrated in the examples below:

```
mysql> SELECT TO_DAYS('0000−00−00');

+−−−−−−−−−−−−−−−−−−−−−−−+

| TO_DAYS('0000−00−00') |

+−−−−−−−−−−−−−−−−−−−−−−−+

| NULL |

+−−−−−−−−−−−−−−−−−−−−−−−+

1 row in set (0.01 sec)
```

```
mysql> SELECT TO_DAYS('0001−01−01');
+−−−−−−−−−−−−−−−−−−−−−−−+
| TO_DAYS('0001−01−01') |
+−−−−−−−−−−−−−−−−−−−−−−−+
| 730851 |
+−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.01 sec)
```

The Number Game

In addition to futzing with days and dates, MySQL also provides you with the ability to perform date arithmetic on specific date and time values, with its DATE_ADD() and DATE_SUB() functions. The syntax of these functions is somewhat more complex than the ones you've seen thus far − here's what it looks like:

```
DATE_ADD(startDate, INTERVAL period periodType)
```

DATE_SUB(startDate, INTERVAL period periodType)

In order to better understand this, consider the following example, which adds 1 year to the specified start date and returns the new value:

```
mysql> SELECT DATE_ADD('2003−04−15', INTERVAL 1 YEAR);

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| DATE_ADD('2003−04−15', INTERVAL 1 YEAR) |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| 2004−04−15 |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

1 row in set (0.18 sec)
```

Here's another one, this one adding 3 hours and 45 minutes to the starting timestamp,

```
mysql> SELECT DATE_ADD('2003−04−15 02:02', INTERVAL "03:45"

mysql> HOUR_MINUTE);

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| DATE_ADD('2003−04−15 02:02', INTERVAL "03:45" HOUR_MINUTE) |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| 2003−04−15 05:47:00 |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

1 row in set (0.01 sec)
```

and yet another, this one adding 7 days, 1 hour, 55 minutes and 10 seconds to December 25 2000:

```
mysql> SELECT DATE_ADD('2000−12−25 12:00:00', INTERVAL "7 01:55:10"

DAY_SECOND);

+-------------------------------------------------------------------+

| DATE_ADD('2000−12−25 12:00:00', INTERVAL "7 01:55:10" DAY_SECOND)
|

+-------------------------------------------------------------------+

| 2001−01−01 13:55:10 |

+-------------------------------------------------------------------+

1 row in set (0.01 sec)
```

In order to better understand this, it's necessary to delve a little deeper into the syntax of the DATE_ADD() and DATE_SUB() functions:

1. The first parameter, startDate, is a date or time value in either string ("YYYY−MM−DD HH:MM:SS") or number (YYYYMMDDHHMMSS) format. This parameter specifies the date to use as a base for the calculation.

2. The second parameter consists of three separate components, which together specify the interval which is to be added to (or subtracted from) the first parameter. The components consist of the keyword INTERVAL, followed by the interval period and a keyword providing information on the interval calculation to be performed.

The relationship between the formatting of the interval period and the keyword following it is fixed, and illustrated in the following table:

period (formatted as)	*periodType*
SECONDS	*SECOND*
MINUTES	*MINUTE*
HOURS	*HOUR*
DAYS	*DAY*
MONTHS	*MONTH*
YEARS	*YEAR*
"MINUTES:SECONDS"	*MINUTE_SECOND*
"HOURS:MINUTES"	*HOUR_MINUTE*
"DAYS HOURS"	*DAY_HOUR*
"YEARS−MONTHS"	*YEAR_MONTH*
"HOURS:MINUTES:SECONDS"	*HOUR_SECOND*
"DAYS HOURS:MINUTES"	*DAY_MINUTE*
"DAYS HOURS:MINUTES:SECONDS"	*DAY_SECOND*

With this in mind, the examples above should become much clearer. If, for example, you wanted to perform a calculation involving interval units of 1 year, you could format your arguments using the YEAR type, whereas if you wanted to perform date arithmetic with hours and minutes, you could format your arguments using the HOUR_MINUTE or HOUR_SECOND types.

Let's look at a few more examples of this in action.

```
mysql> SELECT DATE_ADD(20041130, INTERVAL 1 MONTH);

+-------------------------------------+

| DATE_ADD(20041130, INTERVAL 1 MONTH) |

+-------------------------------------+

| 2004-12-30 |

+-------------------------------------+

1 row in set (0.88 sec)
```

```
mysql> SELECT DATE_ADD(20041130, INTERVAL 31 DAY);
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| DATE_ADD(20041130, INTERVAL 31 DAY) |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 2004−12−31 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.02 sec)

mysql> SELECT DATE_ADD('2003−09−17 03:30:00', INTERVAL "02:30"
HOUR_MINUTE);
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| DATE_ADD('2003−09−17 03:30:00', INTERVAL "02:30" HOUR_MINUTE) |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 2003−09−17 06:00:00 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.01 sec)

mysql> SELECT DATE_SUB('2003−01−01', INTERVAL 24 HOUR);
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| DATE_SUB('2003−01−01', INTERVAL 24 HOUR) |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 2002−12−31 00:00:00 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.00 sec)

mysql> SELECT DATE_SUB('2003−09−17 12:00:00', INTERVAL "1 12" DAY_HOUR);

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| DATE_SUB('2003−09−17 12:00:00', INTERVAL "1 12" DAY_HOUR) |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 2003−09−16 00:00:00 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.00 sec)

mysql> SELECT DATE_SUB('2003−01−01 18:45:10', INTERVAL "17:45:10"
HOUR_SECOND);
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| DATE_SUB('2003−01−01 18:45:10', INTERVAL "17:45:10" HOUR_SECOND) |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 2003−01−01 01:00:00 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.00 sec)
```

Artificial Intelligence

It should be noted that MySQL includes intelligence to automatically format the result of the calculation as either a date−only or date−and−time value. For example, if your arguments to DATE_ADD() or DATE_SUB() contains only year, month and day components, MySQL will output a date−only value as result.

```
mysql> SELECT DATE_SUB(20041130, INTERVAL 3 MONTH);

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| DATE_SUB(20041130, INTERVAL 3 MONTH) |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| 2004−08−30 |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

1 row in set (0.00 sec)
```

However, if your arguments include a time component − hours, minutes or seconds − then MySQL produces a result containing both date and time components.

```
mysql> SELECT DATE_ADD('2010−02−14', INTERVAL "1 1" DAY_HOUR);

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| DATE_ADD('2010−02−14', INTERVAL "1 1" DAY_HOUR) |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| 2010−02−15 01:00:00 |

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

1 row in set (0.01 sec)
```

If you use illegal date or time values, MySQL will still attempt to return a valid result by performing adjustments on the various values.

```
mysql> SELECT DATE_ADD('2010−02−30', INTERVAL 1 DAY);

+---------------------------------------+

| DATE_ADD('2010−02−30', INTERVAL 1 DAY) |

+---------------------------------------+

| 2010−03−03 |

+---------------------------------------+

1 row in set (0.00 sec)
```

MySQL also provides an alternative syntax to DATE_ADD() and DATE_SUB(), which is sometimes more readable − this involves using + and − signs to indicate the type of calculation to be performed. Consider the following examples, which illustrate:

```
mysql> SELECT 20000615 + INTERVAL 5 DAY;

+--------------------------+

| 20000615 + INTERVAL 5 DAY |

+--------------------------+

| 2000−06−20 |

+--------------------------+

1 row in set (0.00 sec)
```

```
mysql> SELECT 20000615000000 − INTERVAL 10 SECOND ;
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 20000615000000 − INTERVAL 10 SECOND |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 2000−06−14 23:59:50 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.00 sec)

mysql> SELECT '2000−06−15 13:00' − INTERVAL "2−1" YEAR_MONTH;
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| '2000−06−15 13:00' − INTERVAL "2−1" YEAR_MONTH |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
| 1998−05−15 13:00:00 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
1 row in set (0.00 sec)
```

A Short Interval

The PERIOD_DIFF() function is primarily used to calculate the number of months between two dates − as illustrated in the following example, which calculates the number of months between December 2002 and December 2003.

```
mysql> SELECT PERIOD_DIFF(200312, 200212);

+-----------------------------+

| PERIOD_DIFF(200312, 200212) |

+-----------------------------+

| 12 |

+-----------------------------+

1 row in set (0.00 sec)
```

The values provided to the PERIOD_DIFF() function must be in the form YYYYMM or YYMM − the following statement is equivalent to the one above:

```
mysql> SELECT PERIOD_DIFF(0312, 0212);

+-------------------------+

| PERIOD_DIFF(0312, 0212) |

+-------------------------+

| 12 |

+-------------------------+

1 row in set (0.00 sec)
```

A corollary to the PERIOD_DIFF() function is the PERIOD_ADD() function, which adds a specified number of months to a date and displays the result. The first argument to PERIOD_ADD() is the start date, the second is the number of months to be added to it. Consider the following example, which adds 3 months to January 2003:

```
mysql> SELECT PERIOD_ADD(200301, 3);

+-----------------------+

| PERIOD_ADD(200301, 3) |

+-----------------------+

| 200304 |

+-----------------------+

1 row in set (0.02 sec)
```

You can even use the PERIOD_ADD() function to perform a subtraction operation, by specifying a negative integer as the second argument to PERIOD_ADD(). Consider the following example, which illustrates by subtracting 5 months from May 2003.

```
mysql> SELECT PERIOD_ADD(200305, −5);

+------------------------+

| PERIOD_ADD(200305, −5) |

+------------------------+

| 200212 |

+------------------------+

1 row in set (0.00 sec)
```

Lather, Rinse, Repeat

Now that you've (hopefully) understood how MySQL's date calculation functions work, it's time to illustrate them with an example. I'll do this in the context of a real−world application, a task scheduler which supports recurring tasks. Without the various date and time arithmetic functions discussed in this tutorial, building such an application is a time−consuming process; with them, it's a snap.

The requirements of this application are fairly basic:

1. It should allow users to enter tasks, task descriptions and email addresses.

2. It should support recurring daily, monthly or yearly tasks.

3. The recurrence interval should be user−defined.

4. When a task becomes due, a message containing the description should be emailed to the corresponding email address.

Let's begin by building a simple table to store our tasks:

```
mysql> CREATE TABLE `tasks` (

−> `id` smallint(6) NOT NULL auto_increment,

−> `msg` varchar(255) NOT NULL default '',

−> `type` set('D','M','Y') NOT NULL default '',

−> `interval` mediumint(9) NOT NULL default '0',

−> `date` date NOT NULL default '0000−00−00',

−> `owner` varchar(255) NOT NULL default '',

−> PRIMARY KEY (id)

−> ) TYPE=MyISAM;

Query OK, 0 rows affected (0.02 sec)
```

```
mysql> DESCRIBE tasks;
+−−−−−−−−−−+−−−−−−−−−−−−−−−−−−+−−−−−−+−−−−−+−−−−−−−−−−−−+−−−−−−−−−−−−−−−−+
| Field | Type | Null | Key | Default | Extra |
+−−−−−−−−−−+−−−−−−−−−−−−−−−−−−+−−−−−−+−−−−−+−−−−−−−−−−−−+−−−−−−−−−−−−−−−−+
| id | smallint(6) | | PRI | NULL | auto_increment |
| msg | varchar(255) | | | | |
| type | set('D','M','Y') | | | | |
| interval | mediumint(9) | | | 0 | |
| date | date | | | 0000−00−00 | |
| owner | varchar(255) | | | | |
+−−−−−−−−−−+−−−−−−−−−−−−−−−−−−+−−−−−−+−−−−−+−−−−−−−−−−−−+−−−−−−−−−−−−−−−−+
6 rows in set (0.00 sec)
```

Here's a quick explanation of the various fields in this table:

1. The "id" column stores a unique identifier for each task. This identifier is created by MySQL by automatically incrementing the identifier of the previous record.

2. The "msg" column stores a message describing the task.

3. The "type" column specifies the type of recurrence, whether daily (D), monthly (M) or yearly (Y).

4. The "interval" column specifies the interval between each recurrence of the task.

5. The "date" column specifies the date on which the task will run next.

6. The "owner" column specifies the email address to which the task notification should be sent.

It is important to note that in the above structure, a recurrence type of "D" and a recurrence interval of 0 would imply that a task would only execute once, on the date specified.

In order to better understand how this works, consider inserting some data into this table, as follows:

```
mysql> INSERT INTO tasks (`id`, `msg`, `type`, `interval`, `date`,

mysql> `owner`)

VALUES ('', 'Run once', 'D', 0, '2003−06−25', 'webmaster@domain'); Query OK,
1 row affected (0.00 sec)
```

```
mysql> INSERT INTO tasks (`id`, `msg`, `type`, `interval`, `date`,
mysql> `owner`)
VALUES ('', 'Run once daily', 'D', 1, '2003−06−25', 'some.user@some.domain.net');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO tasks (`id`, `msg`, `type`, `interval`, `date`,
mysql> `owner`)
VALUES ('', 'Run once every 3 days', 'D', 3, '2003−06−29', 'john@where.am.i');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO tasks (`id`, `msg`, `type`, `interval`, `date`,
mysql> `owner`)
VALUES ('', 'Run once every 2 months', 'M', 2, '2003−06−30', 'user@domain');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO tasks (`id`, `msg`, `type`, `interval`, `date`,
mysql> `owner`)
VALUES ('', 'Run once every year', 'Y', 1, '2003−07−01', 'nobody@some.free.mail.service.com');
Query OK, 1 row affected (0.00 sec)
```

With this structure in place, it's pretty simple to write some code to parse this table on a daily basis and use MySQL's date functions to calculate which tasks to execute. Let's look at that next.

Code Poet

Now that the table structure is defined and some sample data has been stored in it, all that's needed is a script to parse the data and execute the appropriate tasks (based on the scheduled date of each task). Here's an example of what this script might look like:

```
<?php

// define array to convert interval type
// to corresponding SQL clause for DATE_ADD()
$typeArray = array("D" => "DAY", "M" => "MONTH", "Y" => "YEAR");

// what is today?
$today = date("Y−m−d", mktime());

// open connection to database
$connection = mysql_connect("localhost", "root", "secret") or die ("Unable to connect!");
mysql_select_db("db1") or die ("Unable to select database!");

// formulate and execute query
// get a list of all tasks due today
$query = "SELECT `id`,`msg`,`type`,`interval`,`date`, `owner` FROM `tasks` WHERE `date` =
CURRENT_DATE()"; $result = mysql_query($query) or die("Error in query: " . mysql_error());

// iterate over result set
while ($row = mysql_fetch_object($result))
{
// for each task:

// send email
mail($row−>owner, "[REMINDER] $today" , $row−>msg) or die("Unable to send
mail!");

// calculate next run date for task and update tale
$query2 = "UPDATE tasks SET `date` = DATE_ADD(`date`, INTERVAL `interval` " .
$typeArray[$row−>type] . " ) WHERE id = '" . $row−>id . "'";
$result2 = mysql_query($query2) or die("Error in query: " . mysql_error()); }

// close connection
mysql_close($connection);

?>
```

Now, I've written this script in PHP; however, once you understand how it works, it should be fairly easy to write an equivalent piece of code in Perl, Python, Bash or any other language. This script should be executed once a day (maybe via a cron job − note that you will need to compile a PHP binary in order to use the script above as is); it will scan the records in the table, perform date calculations, and send out notifications to the email address specified for each task if it turns out that the task is due on that particular day.

The first thing the script does is figure out the current date (and format it to a MySQL−compliant format). Next, it executes an SQL query in order to obtain a list of all the tasks to be executed on the current date, and sends email notification to the owner of each task. The DATE_ADD() function discussed previously is then used to calculate the next run date of the task (by adding the specified interval period to the current date) and the table is updated appropriately with new dates.

Of course, this is not a foolproof design. Missing even a single script run could well result in some tasks not being executed and consequently, their next run dates not being updated. If these are recurring tasks, then future invocations of the script will not "see" them, and they will never be executed again.

This design flaw may be worked around by intermittently running automated scripts to verify the integrity of the table. Redundant tasks, or tasks with dates outside a specific temporal window, can be sent to a system administrator for inspection and deletion where needed.

Thus, by allowing you to perform arithmetic operations on temporal data, MySQL's date/time API allows you to easily and accurately build applications to manipulate date and time values. The example above is just one of many possibilities this opens up − I'll leave the rest to your imagination. Go on, think about it − and if you come up with something interesting, write in and tell me about it.

Until then, though...stay healthy!

Note: Examples in this article have been tested on Linux/i586 with MySQL 4.0 and PHP 4.2.3. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!