the data, it would be easier to create a simple script using one of the APIs to extract, reorder, and then replace the column values.

SUBSTR( )

```
SUBSTRING(string, position[, length])
SUBSTRING(string FROM position FOR length)
```

This function is an alias of SUBSTRING( ). See its description next for details and an example of its use.

SUBSTRING( )

```
SUBSTRING(string, position[, length])
SUBSTRING(string FROM position[ FOR length])
```

This function returns the characters of a given string, starting from the position given. The first character is numbered 1. You can restrict the length of the string retrieved by specifying a limit. The function is similar to MID( ). Here is an example:

```
SELECT CONCAT_WS('-',
   SUBSTRING(soc_sec, 1, 3),
   SUBSTRING(soc_sec FROM 4 FOR 2),
   SUBSTRING(soc_sec FROM 6)
)
AS 'Social Security Nbr.'
FROM students LIMIT 1;

+----------------------+
| Social Security Nbr. |
+----------------------+
| 433-12-3456          |
+----------------------+
```

This example shows the two syntaxes of SUBSTRING( ) for reformatting a Social Security number (the U.S. federal tax identification number) stored without dashes. It uses CONCAT_WS( ) to put the three pieces of data together, separated by the hyphen given.

SUBSTRING_INDEX( )

```
SUBSTRING_INDEX(string, delimiter, count)
```

This function returns a substring of *string*, using *delimiter* to separate substrings and *count* to determine which of the substrings to return. Thus, a *count* of 1 returns the first substring, 2 returns the second, and so on. A negative number instructs the function to count from the right end. Here is an example:

```
SELECT SUBSTRING_INDEX(pre_req, '|', -1)
AS 'Last Prerequisite',
pre_req AS 'All Prerequisites'
FROM courses WHERE course_id = '1245';

+-------------------+--------------------------+
| Last Prerequisite | All Prerequisites        |
+-------------------+--------------------------+
| ENGL-202           | ENGL-101|ENGL-201|ENGL-202 |
+--------------------+---------------------------+
```

In this example, the pre_req column for each course contains prerequisite courses separated by vertical bars. The statement displays the last prerequisite, because –1 was entered for the count.

TRIM()

`TRIM([[BOTH|LEADING|TRAILING] [`*`padding`*`] FROM]` *`string`*`)`

This function returns the given string with any trailing or leading padding removed, depending on which is specified. If neither is specified, `BOTH` is the default, causing both leading and trailing padding to be removed. The default padding is a space if none is specified. The function is multibyte-safe.

As an example, in a table containing the results of a student survey we notice that one of the columns that lists each student's favorite activities contains extra commas at the end of the comma-separated list of activities. This may have been caused by a problem in the web interface, which treated any activities that a student didn't select as blank values separated by commas at the end (e.g., `biking,reading,,,,`):

```
UPDATE student_surveys
SET favorite_activities =
TRIM(LEADING SPACE(1) FROM TRIM(TRAILING ',' FROM favorite_activities));
```

In this example, we're using `TRIM( )` twice: once to remove the trailing commas from the column `favorite_activities` and then again on those results to remove leading spaces. Since the functions are part of an `UPDATE` statement, the double-trimmed results are saved back to the table for the row for which the data was read. This is more verbose than it needs to be, though. Because a space is the default padding, we don't have to specify it. Also, because we want to remove both leading and trailing spaces and commas from the data, we don't have to specify `LEADING` or `TRAILING` and can allow the default of `BOTH` to be used. Making these adjustments, we get this tighter SQL statement:

```
UPDATE student_surveys
SET favorite_activities =
TRIM(TRIM(',' FROM favorite_activities));
```

If we suspected that the faulty web form also added extra commas between the text (not just at the end), we could wrap these concentric uses of `TRIM( )` within `REPLACE( )` to replace any occurrences of consecutive commas with a single comma:

```
UPDATE student_surveys
SET favorite_activities =
REPLACE(TRIM(TRIM(',' FROM favorite_activities)), ',,', ',');
```

UCASE()

`UCASE(`*`string`*`)`

This function converts a given string to all uppercase letters. It's an alias of `UPPER( )`. Here is an example:

```
SELECT course_id AS 'Course ID',
UCASE(course_name) AS Course
FROM courses LIMIT 3;
```

```
+-----------+----------------------+
| Course ID | Course               |
+-----------+----------------------+
|      1245 | CREATIVE WRITING     |
|      1255 | PROFESSIONAL WRITING |
|      1244 | AMERICAN LITERATURE  |
+-----------+----------------------+
```

UNCOMPRESS( )

UNCOMPRESS(*string*)

This function returns the uncompressed string corresponding to the compressed string given, reversing the results of the COMPRESS( ) function. It requires MySQL to have been compiled with a compression library (e.g., **zlib**). It returns NULL if the string is not compressed or if MySQL wasn't compiled with zlib. This function is available as of version 4.1.1 of MySQL. Here is an example:

```
SELECT UNCOMPRESS(essay)
FROM applications_archive
WHERE applicant_id = '1748';
```

UNCOMPRESSED_LENGTH( )

UNCOMPRESSED_LENGTH(*string*)

This function returns the number of characters contained in the given compressed string before it was compressed. You can compress strings using the COMPRESS( ) function. This function is available as of version 4.1 of MySQL. Here is an example:

```
SELECT UNCOMPRESSED_LENGTH(COMPRESS(essay))
FROM student_applications
WHERE applicant_id = '1748';
```

UNHEX( )

UNHEX(*string*)

This function converts hexadecimal numbers to their character equivalents. It reverses the results of the HEX( ) function and is available as of version 4.1.2 of MySQL.

To illustrate its use, suppose that in a table we have a column with a binary character in the data; specifically, tabs were entered through a web interface using an API. However, the column is a VARCHAR data type. The problem is that when the data is retrieved, we want to line up all the results in our display by counting the length of each column, and a tab keeps the display from lining up vertically. So we want to fix the data. We can use UNHEX( ) to locate rows containing the binary character and then replace it with spaces instead:

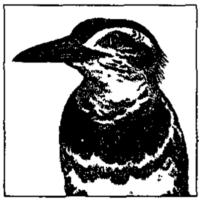
```
UPDATE students
SET comments = REPLACE(comments, UNHEX(09), SPACE(4))
WHERE LOCATE(UNHEX(09), comments);
```

We've looked at an ASCII chart and seen that a tab is represented by the hexadecimal number `09`. Knowing that bit of information, in the `WHERE` clause we're passing that value to `UNHEX( )` to return the binary character for a tab, yielding the search string with which `LOCATE( )` will search the column `comments`. If it doesn't find a tab in the column for a row, it will return 0. Those rows will not be included in the search results. The ones that do contain tabs will have a value of 1 or greater and therefore will be included in the results. Using `UNHEX( )` along with `REPLACE( )` in the `SET` clause, we replace all tabs found with four spaces.

UPPER( )

`UPPER(`*`string`*`)`

This function converts a given string to all uppercase letters. It's an alias of `UCASE( )`. See that function's description earlier in this chapter for an example.



12 Date and Time Functions

By using temporal data type columns, you can use several built-in functions offered by MySQL. This chapter presents those functions. Currently, five temporal data types are available: `DATE`, `TIME`, `DATETIME`, `TIMESTAMP`, and `YEAR`. You would set a column to one of these data types when creating or altering a table. See the descriptions of `CREATE TABLE` and `ALTER TABLE` in Chapter 6 for more details. The `DATE` column type can be used for recording just the date. It uses the *yyyy-mm-dd* format. The `TIME` column type is for recording time in the *hhh:mm:ss* format. To record a combination of date and time, use `DATETIME`: *yyyy-mm-dd hh:mm:ss*. The `TIMESTAMP` column is similar to `DATETIME`, but it is more limited in its range of allowable time: it starts at the Unix epoch time (i.e., 1970-01-01) and stops at the end of 2037. Plus, it has the distinction of resetting its value automatically when the row in which it is contained is updated, unless you specifically instruct MySQL otherwise. Finally, the `YEAR` data type is used only for recording the year in a column. For more information on date and time data types, see Appendix B.

Any function that calls for a date or a time data type will also accept a combined datetime data type. MySQL requires that months range from 0 to 12 and that days range from 0 to 31. Therefore, a date such as February 30 would be accepted prior to version 5.0.2 of MySQL. Beginning in version 5.0.2, MySQL offers more refined validation that would reject such a date. However, some date functions accept 0 for some or all components of a date, or incomplete date information (e.g., 2008-06-00). As a general rule, the date and time functions that extract part of a date value usually accept incomplete dates, but date and time functions that require complete date information return NULL when given an incomplete date. The descriptions of these functions in this chapter indicate which require valid dates and which don't, as well as which return 0 or NULL for invalid dates.

The bulk of this chapter consists of an alphabetical listing of date and time functions, with explanations of each. Each of the explanations include an example of the function's use, along with a resulting display, if any. For the examples in this chapter, I used the scenario of a professional services firm (e.g., a law firm or an investment advisory firm) that tracks appointments and seminars in MySQL. For help locating the appropriate function, see the next section or the index at the end of this book.

Date and Time Functions Grouped by Type

Following are lists of date and time functions, grouped according to their purpose: to retrieve the date or time, to extract an element from a given date or time, or to perform calculations on given dates or times.

Determining the Date or Time

CURDATE( ), CURRENT_DATE( ), CURRENT_TIME( ), CURRENT_TIMESTAMP( ), CURTIME( ), LOCALTIME( ), LOCALTIMESTAMP( ), NOW( ), SYSDATE( ), UNIX_TIMESTAMP( ), UTC_DATE( ), UTC_TIME( ), UTC_TIMESTAMP( ).

Extracting and Formatting the Date or Time

DATE( ), DATE_FORMAT( ), DAY( ), DAYNAME( ), DAYOFMONTH( ), DAYOFWEEK( ), DAYOFYEAR( ), EXTRACT( ), GET_FORMAT( ), HOUR( ), LAST_DAY( ), MAKEDATE( ), MAKETIME( ), MICROSECOND( ), MINUTE( ), MONTH( ), MONTHNAME( ), QUARTER( ), SECOND( ), STR_TO_DATE( ), TIME( ), TIME_FORMAT( ), TIMESTAMP( ), WEEK( ), WEEKDAY( ), WEEKOFYEAR( ), YEAR( ), YEARWEEK( ).

Calculating and Modifying the Date or Time

ADDDATE( ), ADDTIME( ), CONVERT_TZ( ), DATE_ADD( ), DATE_SUB( ), DATEDIFF( ), FROM_DAYS( ), FROM_UNIXTIME( ), PERIOD_ADD( ), PERIOD_DIFF( ), SEC_TO_TIME( ), SLEEP( ), SUBDATE( ), SUBTIME( ), TIME_TO_SEC( ), TIMEDIFF( ), TIMESTAMPADD( ), TIMESTAMPDIFF( ), TO_DAYS( ).

Date and Time Functions in Alphabetical Order

The rest of the chapter lists each function in alphabetical order.

ADDDATE( )

ADDDATE(*date*, INTERVAL *value type*)
ADDDATE(*date*, *days*)

This function adds the given interval of time to the date or time provided. This is a synonym for DATE_ADD( ); see its definition later in this chapter for details and interval types. The second, simpler syntax is available as of version 4.1 of MySQL. This shorthand syntax does not work, though, with DATE_ADD( ). Here is an example:

```
UPDATE seminars
SET seminar_date = ADDDATE(seminar_date, INTERVAL 1 MONTH)
WHERE seminar_date = '2007-12-01';
```

```
UPDATE appointments
SET appt_date = DATE_ADD(appt_date, INTERVAL 1 DAY)
WHERE appt_id='1202';
```

In this example, the appointment date is changed to its current value plus one additional day to postpone the appointment by a day. If we changed the 1 to –1, MySQL would subtract a day instead. This would make the function the equivalent of DATE_SUB( ).

If you leave out some numbers in the second argument, MySQL assumes that the leftmost interval factors are 0 and are just not given. In the following example, although we're using the interval HOUR_SECOND, we're not giving the number of hours and the function still works—assuming we don't mean 5 hours and 30 minutes later. MySQL assumes here that we mean '00:05:30' and not '05:30:00':

```
SELECT NOW( ) AS 'Now',
DATE_ADD(NOW( ), INTERVAL '05:30' HOUR_SECOND)
AS 'Later';

+---------------------+---------------------+
| Now                 | Later               |
+---------------------+---------------------+
| 2007-03-14 10:57:05 | 2007-03-14 11:02:35 |
+---------------------+---------------------+
```

When adding the intervals MONTH, YEAR, or YEAR_MONTH to a date, if the given date is valid but the results would be an invalid date because it would be beyond the end of a month, the results are adjusted to the end of the month:

```
SELECT DATE_ADD('2009-01-29', INTERVAL 1 MONTH)
AS 'One Month Later';

+-----------------+
| One Month Later |
+-----------------+
| 2009-02-28      |
+-----------------+
```

Table 12-1 shows the intervals that may be used and how the data should be ordered. For interval values that require more than one factor, a delimiter is used and the data must be enclosed in quotes. Other delimiters may be used besides those shown in the table. For example, *'hh|mm|ss'* could be used for HOUR_SECOND. In case you hadn't noticed, the names for intervals involving more than two time factors use the name of the first and last factor (e.g., DAY_MINUTE and not DAY_HOUR_MINUTE). Keep that in mind when trying to remember the correct interval.

Table 12-1. DATE_ADD( ) intervals and formats

INTERVAL	Format for given values
DAY	*dd*
DAY_HOUR	*'dd hh'*
DAY_MICROSECOND	*'dd.nn'*
DAY_MINUTE	*'dd hh:mm'*
DAY_SECOND	*'dd hh:mm:ss'*

INTERVAL	Format for given values
HOUR	*hh*
HOUR_MICROSECOND	*'hh.nn'*
HOUR_MINUTE	*'hh:mm'*
HOUR_SECOND	*'hh:mm:ss'*
MICROSECOND	*nn*
MINUTE	*mm*
MINUTE_MICROSECOND	*'mm.nn'*
MINUTE_SECOND	*'mm:ss'*
MONTH	*mm*
QUARTER	*qq*
SECOND	*ss*
SECOND_MICROSECOND	*'ss.nn'*
WEEK	*ww*
YEAR	*yy*
YEAR_MONTH	*'yy-mm'*

DATE_FORMAT( )

DATE_FORMAT(*date*, '*format_code*')

This function returns a date and time in a desired format, based on formatting codes listed within quotes for the second argument of the function. Here is an example:

```
SELECT DATE_FORMAT(appointment, '%W - %M %e, %Y at %r')
AS 'Appointment'
FROM appointments
WHERE client_id = '8392'
AND appointment > CURDATE( );

+-------------------------------------+
| Appointment                         |
+-------------------------------------+
| Monday - June 16, 2008 at 01:00:00 PM |
+-------------------------------------+
```

Using the formatting codes, we're specifying in this example that we want the name of the day of the week (%W) followed by a dash and then the date of the appointment in a typical U.S. format (%M %e, %Y), with the month name and a comma after the day. We're ending with the word "at" followed by the full nonmilitary time (%r). The results are returned as a binary string.

As of MySQL version 5.1.15, a string is returned along with the character set and collation of the string, taken from the character_set_connection and the collation_connection system variables. This allows the function to return non-ASCII characters. Here is an example of this function:

```
SELECT NOW( ),
DATE_FORMAT(NOW( ), '%M') AS 'Month in Hebrew';

+---------------------+-----------------+
| Now                 | Month in Hebrew |
+---------------------+-----------------+
| 2008-03-14 12:00:24 | מרץ             |
+---------------------+-----------------+
```

In this example, of course, the client and server were set to display Hebrew characters. Also, the server variable lc_time_names was set to Hebrew (he_IL) so as to return the Hebrew word for March. See MySQL's documentation page on *MySQL Server Locale Support* (*http://dev.mysql.com/doc/refman/5.1/en/locale-support.html*) for a list of locale values available for time names.

Table 12-2 contains a list of all the formatting codes you can use with DATE_FORMAT( ). You can also use these codes with TIME_FORMAT( ) and EXTRACT( ).

Table 12-2. DATE_FORMAT( ) format codes and resulting formats

Code	Description	Results
%%	A literal '%'	
%a	Abbreviated weekday name	(Sun...Sat)
%b	Abbreviated month name	(Jan...Dec)
%c	Month, numeric	(1...12)
%d	Day of the month, numeric	(00...31)
%D	Day of the month with English suffix	(1st, 2nd, 3rd, etc.)
%e	Day of the month, numeric	(0...31)
%f	Microseconds, numeric	(000000...999999)
%h	Hour	(01...12)
%H	Hour	(00...23)
%i	Minutes, numeric	(00...59)
%I	Hour	(01...12)
%j	Day of the year	(001...366)
%k	Hour	(0...23)
%l	Hour	(1...12)
%m	Month, numeric	(01...12)
%M	Month name	(January...December)
%p	A.M. or P.M.	A.M. or P.M.
%r	Time, 12-hour	(hh:mm:ss [AM|PM])
%s	Seconds	(00...59)
%S	Seconds	(00...59)
%T	Time, 24-hour	(hh:mm:ss)

Code	Description	Results
%u	Week, where Monday is the first day of the week	(0...52)
%U	Week, where Sunday is the first day of the week	(0...52)
%v	Week, where Monday is the first day of the week; used with %x	(1...53)
%V	Week, where Sunday is the first day of the week; used with %X	(1...53)
%w	Day of the week	(0=Sunday...6=Saturday)
%W	Weekday name	(Sunday...Saturday)
%x	Year for the week, where Monday is the first day of the week, numeric, four digits; used with %v	(yyyy)
%X	Year for the week, where Sunday is the first day of the week, numeric, four digits; used with %V	(yyyy)
%y	Year, numeric, two digits	(yy)
%Y	Year, numeric, four digits	(yyyy)

DATE_SUB( )

`DATE_SUB(`*`date`*`, INTERVAL `*`number type`*`)`

Use this function to subtract from the results of a date or time data type column. See Table 12-1, under the description of `DATE_ADD( )`, for a list of interval types. Here is an example of this function:

```
SELECT NOW( ) AS Today,
DATE_SUB(NOW( ), INTERVAL 1 DAY)
AS Yesterday;

+---------------------+---------------------+
| Today               | Yesterday           |
+---------------------+---------------------+
| 2007-05-14 14:26:54 | 2007-05-13 14:26:54 |
+---------------------+---------------------+
```

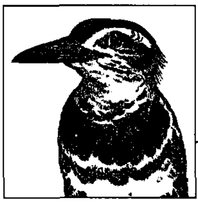
Notice in this example that the time remains unchanged, but the date was reduced by one day. If you place a negative sign in front of the value 1, the reverse effect will occur, giving a result of May 15 in this example. Any intervals that can be used with `DATE_ADD( )` can also be used with `DATE_SUB( )`.

DATEDIFF( )

`DATEDIFF(`*`date, date`*`)`

This function returns the number of days of difference between the two dates given. Although a parameter may be given in date and time format, only the dates are used for determining the difference. This function is available as of version 4.1.1 of MySQL. Here is an example:

```
SELECT CURDATE( ) AS Today,
DATEDIFF('2008-12-25', NOW( ))
AS 'Days to Christmas';
```



13 Mathematical Functions

MySQL has many built-in mathematical functions that you can use in SQL statements for performing calculations on values in databases. Each function accepts either numbers or numeric columns for parameter values. All mathematical functions return NULL on error.

The following functions are covered in this chapter:

ABS( ), ACOS( ), ASIN( ), ATAN( ), ATAN2( ), BIT_COUNT( ), CEIL( ), CEILING( ), CONV( ), COS( ), COT( ), DEGREES( ), EXP( ), FLOOR( ), FORMAT( ), GREATEST( ), INET_ATON( ), INET_NTOA( ), LEAST( ), LN( ), LOG( ), LOG2( ), LOG10( ), MOD( ), OCT( ), PI( ), POW( ), POWER( ), RADIANS( ), RAND( ), ROUND( ), SIGN( ), SIN( ), SQRT( ), TAN( ), TRUNCATE( ).

Functions in Alphabetical Order

The following is a list of MySQL mathematical functions in alphabetical order, along with descriptions of each and examples of their use.

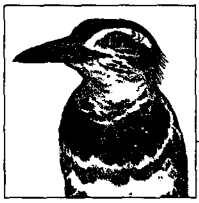
ABS( )

ABS(*number*)

This function returns the absolute value of a given number. Here is an example:

```
SELECT ABS(-10);

+----------+
| ABS(-10) |
+----------+
|       10 |
+----------+
```



14 Flow Control Functions

MySQL has a few built-in flow control functions that you can use in SQL statements for more precise and directed results. This chapter provides the syntax of function and gives examples of their use. For the examples in this chapter, a fictitious database for a stock broker is used.

The following functions are covered in this chapter:

CASE, IF( ), IFNULL( ), ISNULL( ), NULLIF( ).

Functions in Alphabetical Order

The following are the MySQL flow control functions listed alphabetically.

CASE

```
CASE value
  WHEN [value] THEN result
   . . .
  [ELSE result]
END

CASE
  WHEN [condition] THEN result
   . . .
 [ELSE result]
END
```

This function produces results that vary based on which *condition* is true. It is similar to the IF( ) function, except that multiple conditions and results may be strung together. In the first syntax shown, the *value* given after CASE is compared to each WHEN value. If a match is found, the *result* given for the THEN is returned. The second syntax tests each condition independently, and they are not based on a single value. For both syntaxes, if no match is found and an ELSE clause is included, the result given for the ELSE clause is returned. If there is no match and no ELSE clause is given, NULL is returned.

If the chosen *result* is a string, it is returned as a string data type. If *result* is numeric, the result may be returned as a decimal, real, or integer value.

Here's an example of the first syntax shown:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS Client,
telephone_home AS Telephone,
CASE type
  WHEN 'RET' THEN 'Retirement Account'
  WHEN 'REG' THEN 'Regular Account'
  WHEN 'CUS' THEN 'Minor Account'
END AS 'Account Type'
FROM clients;
```

This SQL statement retrieves a list of clients and their telephone numbers, along with a description of their account types. However, the account type is a three-letter abbreviation, so CASE( ) is used to substitute each type with a more descriptive name.

This example uses the syntax in which a common parameter is evaluated to determine the possible result. The following SQL statement utilizes the other syntax for the function:

```
SELECT CONCAT(name_last, SPACE(1), name_first) AS Prospect,
CASE
  WHEN YEAR(NOW( )) - YEAR(birth_date) ≤ 17 THEN 'Minor'
  WHEN YEAR(NOW( )) - YEAR(birth_date) > 17 < 26 THEN 'Too Young'
  WHEN YEAR(NOW( )) - YEAR(birth_date) > 60 THEN 'Elderly'
  ELSE home_telephone;
END
AS Telephone
FROM prospects;
```

In this example, the SQL statement analyzes a table containing a list of people that the broker might call to buy an investment. The table contains the birth dates and the telephone numbers of each prospect. The SQL statement provides the telephone numbers only for prospects aged 26 to 60 because anyone younger or older would not be suitable for this particular investment. However, a message for each prospect that is disqualified is given based on the clauses of the CASE( ) statement.

When using a CASE statement within a stored procedure, it cannot be given a NULL value for the ELSE clause. Also, a CASE statement ends with END CASE.

IF()

IF(*condition*, *result*, *result*)

This function returns the ***result*** given in the second argument if the *condition* given in the first argument is met (i.e., the *condition* does not equal 0 or NULL). If the condition does equal 0 or NULL, the function returns the ***result*** given in the third argument. Note that the value of *condition* is converted to an integer. Therefore, use a comparison operator when trying to match a string or a floating-point value. The function returns a numeric or a string value depending on its use. As of version 4.0.3 of MySQL, if the second or the third argument is NULL, the type (i.e., string, float, or integer) of the other non-NULL argument will be returned:

```
SELECT clients.client_id AS ID,
CONCAT(name_first, SPACE(1), name_last) AS Client,
telephone_home AS Telephone, SUM(qty) AS Shares,
IF(
   (SELECT SUM(qty * price)
    FROM investments, stock_prices
    WHERE stock_symbol = symbol
    AND client_id = ID )
    > 100000, 'Large', 'Small') AS 'Size'
FROM clients, investments
WHERE stock_symbol = 'GT'
AND clients.client_id = investments.client_id
GROUP BY clients.client_id LIMIT 2;

+------+----------------+-----------+--------+-------+
| ID   | Client         | Telephone | Shares | Size  |
+------+----------------+-----------+--------+-------+
| 8532 | Jerry Neumeyer | 834-8668  |  200   | Large |
| 4638 | Rusty Osborne  | 833-8393  |  200   | Small |
+------+----------------+-----------+--------+-------+
```

This SQL statement is designed to retrieve the names and telephone numbers of clients who own Goodyear stock (the stock symbol is *GT*) because the broker wants to call them to recommend that they sell it. The example utilizes a subquery (available as of version 4.1 of MySQL) to tally the value of all the clients' stocks first (not just Goodyear stock), as a condition of the `IF( )` function. It does this by joining the `investments` table (which contains a row for each stock purchase and sale) and the `stock_prices` table (which contains current prices for all stocks). If the sum of the value of all stocks owned by the client (the results of the subquery) is more than $100,000, a label of `Large` is assigned to the `Size` column. Otherwise, the client is labeled `Small`. The broker wants to call her large clients first. Notice in the results shown that both clients own the same number of shares of Goodyear, but one has a large portfolio.

Note that the `IF` statement used in stored procedures has a different syntax from the `IF( )` function described here. See Chapter 17 for more information on the `IF` statement.

IFNULL( )

`IFNULL(`*`condition, result`*`)`

This function returns the results of the *`condition`* given in the first argument of the function if its results are not NULL. If the condition results are NULL, the results of the expression or string given in the second argument are returned. It will return a numeric or a string value depending on the context:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS Client,
telephone_home AS Telephone,
IFNULL(goals, 'No Goals Given') AS Goals
FROM clients LIMIT 2;
```

```
+----------------+-----------+----------------+
| Client         | Telephone | Goals          |
+----------------+-----------+----------------+
| Janice Sogard  | 835-1821  | No Goals Given |
| Kenneth Bilich | 488-3325  | Long Term      |
+----------------+-----------+----------------+
```

This SQL statement provides a list of clients and their telephone numbers, along with their investment goals. If the client never told the broker of an investment goal (i.e., the goals column is NULL), the text "No Goals Given" is displayed.

ISNULL( )

ISNULL(*column*)

Use this function to determine whether the value of the argument given in parentheses is NULL. It returns 1 if the value is NULL and 0 if it is not NULL. Here is an example:

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS Client,
telephone_work AS 'Work Telephone'
FROM clients
WHERE ISNULL(telephone_home);
```

In this example, after realizing that we don't have home telephone numbers for several of our clients, we use the `ISNULL( )` function in the `WHERE` clause of a `SELECT` statement to list client names and their work telephone numbers so that we can call them to get their home telephone numbers. Only rows in which the `home_telephone` column is NULL will result in a value of 1 and will therefore be shown in the results.

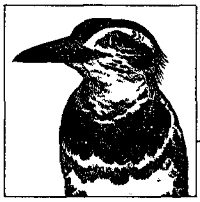
NULLIF( )

NULLIF(*condition1*, *condition2*)

This function returns NULL if the two arguments given are equal. Otherwise, it returns the value or results of the first argument. Here is an example:

```
SELECT clients.client_id AS ID,
CONCAT(name_first, SPACE(1), name_last) AS Client,
telephone_home AS Telephone,
NULLIF(
  (SELECT SUM(qty * price)
   FROM investments, stock_prices
   WHERE stock_symbol = symbol
   AND client_id = ID ), 0)
AS Value
FROM clients, investments
WHERE clients.client_id = investments.client_id
GROUP BY clients.client_id;
```

In this example, NULL is returned for the `Value` column if the value of the client's stocks is 0 (i.e., the client had stocks but sold them all). If there is a value to the stocks, however, the sum of their values is displayed.



15 MySQL Server and Client

The primary executable file making up the MySQL server is the *mysqld* daemon, which listens for requests from clients and processes them. The general-purpose client provided with MySQL is the *mysql* program. This chapter presents the many options available for both the *mysqld* MySQL server and the *mysql* client. A few scripts provided with MySQL that are used to start the server (`mysqld_multi` and `mysqld_safe`) are also explained. The daemons and scripts are listed in alphabetical order.

mysql Client

mysql

`mysql` *`options`* `[`*`database`*`]`

The `mysql` client can be used to interact with MySQL in terminal or monitor mode. To enter monitor mode, enter something like the following from the command line:

```
mysql -u russell -p
```

If the MySQL server is running, the client will prompt the user for a password (thanks to the `-p` option). Once in monitor mode, you can enter SQL statements to view or to change data as well as the status of the server.

As an alternative to monitor mode, when performing straightforward tasks in MySQL, you can still use the `mysql` client from the command line. For instance, to execute a batch file that contains several SQL statements that will insert data into a database, you could do something like this:

```
mysql -u russell -pmy_pwd db1 < stuff.sql
```

In this example, the password is given so that the user isn't prompted. It's entered immediately after the `-p` option without a space in between. Although including the password on the command line poses a security risk for interactive use, it's a valuable feature for using `mysql` in scripts.

Next, the database name `db1` is given. The Unix redirect (the less-than sign) tells the shell to input the test file *stuff.sql* to the command. When the client has finished processing the text file, the user is returned to the command prompt.

To handle even smaller tasks, you can execute a single SQL command against the database by running `mysql` with the `--execute` or `-e` option.

Several options may be given when calling the `mysql` client at the command line. They can also be included in the options file (*my.cnf* or *my.ini*, depending on your system) under the group heading of `[client]`. If used in the options file, the leading double-dashes are not included. The options are listed alphabetically here:

`--auto-rehash`
: This option generates a hash of table and column names to complete the names for users when typing in monitor mode; users invoke autocompletion by pressing the Tab key after having entered the first few letters of the name.

`--batch, -B`
: This option causes the client to display data selected with fields separated by tabs and rows by carriage returns. The client won't prompt the user, won't display error messages to the `stdout`, and won't save to the history file.

`--character-sets-dir=`*`path`*
: This option specifies the local directory containing character sets for the client to use.

`--column-names`
: This option instructs the client to return the names of columns in a results set. This is more relevant when executing SQL statements from the command line.

`--column-type-info, -m`
: This option instructs the client to return the metadata for columns in a results set. This option is available as of version 5.1.14 of MySQL; the short form is available as of version 5.1.21.

`--compress, -C`
: This option instructs the client to compress data passed between it and the server if supported.

`--database=database, -D database`
: This option sets the default database for the client to use. This is equivalent to executing the `USE` statement.

`--debug[=options], -#[options]`
: This option instructs the client to record debugging information to the log file specified. The set of flags used by default is `d:t:o,logname`. See Table 16-1 at the end of the list of options for `mysqldump` in the next chapter for an explanation of these flags and others that may be used.

`--debug-check`
: This option causes the client to display debugging information when finished. This option is available as of version 5.1.21 of MySQL.

`--debug-info, -T`
: This option adds debugging, CPU usage, and memory usage information to the log when the utility ends.

`--default-character-sets-dir=`*`path`*
: This option specifies the local directory that contains the default character sets for the client to use. Enter `SHOW CHARACTER SET;` on the server for a list of character sets available.

`--defaults-group-suffix=`*`value`*
: The client looks for options in the options file under the group headings of `[mysql]` and `[client]`. Use this option to specify option groups that the client is to use, based on their suffixes. For instance, the value given might be just `_special` so that groups such as `[mysql_special]` and `[client_special]` will be included.

`--delimiter=`*`string`*, `-F` *`string`*
: This option use this option to specify the delimiter used to terminate each SQL statement when entered into the client. By default, the client expects a semicolon.

`--execute='`*`statement`*`'`, `-e '`*`statement`*`'`
: This option executes the SQL statement contained in single or double quotes, then terminates the client.

`--force`, `-f`
: This option makes the client continue executing or processing a statement even if there are SQL errors.

`--help`, `-?`
: This option displays basic help information.

`--hostname=`*`host`*, `-h` *`host`*
: This option specifies the hostname or IP address of the MySQL server. The default is *localhost*, which connects to a server on the same system as the client.

`--html`, `-H`
: This option instructs the client to return results in an HTML format when executing an SQL statement at the command line or from a file containing SQL statements.

`--ignore-spaces`, `-i`
: This option instructs the client to ignore spaces after function names (e.g., `CUR_DATE( )`) when executing SQL statements at the command line or from a text file containing SQL statements.

`--line-numbers`
: When the client is accepting SQL statements from an input file, this option instructs the client to display the line number of an SQL statement that has returned an error. This is the default option; use `--skip-line-numbers` to disable this option.

`--local-infile[={0|1}]`
: The SQL statement `LOAD DATA INFILE` imports data into a database from a file. That file could be located on the server or on the computer in which the client is running (i.e., locally). To indicate that a file is local, you would add the `LOCAL` flag to that statement. This option sets that flag: a value of 1 enables the `LOCAL`, whereas a value of 0 indicates that the file is on the server. If the server is set so it imports data only from files on the server, this option will have no effect.

`--named-commands`, `-G`
: This option permits named commands on the client. See the next section for this client program for a description of commands. Enter `help` or `\h` from the *mysql* client to get a list of them. This option is enabled by default. To disable it, use the `--skip-named-commands` option.

`--no-auto-rehash, -A`
: Automatic rehashing is normally used to let the user complete table and column names when typing in monitor mode by pressing the Tab key after having entered the first few letters of the name. This option disables autocompletion and thereby decreases the startup time of the client. This option is deprecated as of version 4 of MySQL.

`--no-beep`
: This option instructs client not to emit a warning sound for errors.

`--no-named-commands`
: This option disables named commands on the client, except when at the start of a line (i.e., named commands cannot appear in the middle of an SQL statement). This option is enabled by default. See the description of the `--named-commands` option and the following section for more information.

`--no-tee`
: This option instructs the client not to write results to a file.

`--one-database, -o`
: This option instructs the client to execute SQL statements only for the default database (set by the `--database` option) and to ignore SQL statements for other databases.

`--pager[=`*`utility`*`]`
: With this option, on a Unix type of system, you can pipe the results of an SQL statement executed from the command line to a pager utility (e.g., `more`) that will allow you to view the results one page at a time and possibly scroll up and down through the results. If this option is given without specifying a particular pager utility, the value of the environment variable `PAGER` will be used. This option is enabled by default. Use the `--skip-pager` option to disable it.

`--password[=`*`password`*`], -p[`*`password`*`]`
: This option provide the password to give to the MySQL server. No spaces are allowed between the `-p` and the password. If this option is entered without a password, the user will be prompted for one.

`--port=`*`port`*`, -P `*`port`*
: This option specifies the socket port to use for connecting to the server. The default is 3306. If you run multiple daemons for testing or other purposes, you can use different ports for each by setting this option.

`--prompt=`*`string`*
: This option sets the prompt for monitor mode to the given string. By default, it's set to `mysql>`.

`--protocol=`*`protocol`*
: This option specifies the protocol to use when connecting to the server. The choices are `TCP`, `SOCKET`, `PIPE`, and `MEMORY`.

`--quick, -q`
: This option causes the client to retrieve and display data one row at a time instead of buffering the entire results set before displaying data. With this option, the history file isn't used and it may slow the server if the output is suspended.

`--raw, -r`
: For data that may contain characters that would normally be converted in batch mode to an escape-sequence equivalent (e.g., newline to `\n`), this option may be used to have the client print out the characters without converting them.

`--reconnect`
: This option instructs the client to attempt to reconnect to the server if the connection is lost. The client tries only once, though. This is enabled by default. To disable it, use `--skip-reconnect`. To make the client wait until the server is available, use `--wait`.

`--safe-updates, -U`
: This option helps prevent inadvertent deletion of multiple and possibly all rows in a table. It requires that when the `DELETE` or `UPDATE` statements are used, a `WHERE` clause be given with a key column and value. If this option is included in the options file, using it at the command line when starting the client will disable it.

`--secure-auth`
: This option prevents authentication of users with passwords created prior to version 4.1 of MySQL or connecting to servers that permit the old format.

`--set-variable` *`var=value`*`, -o` *`var=value`*
: This option sets a server variable. Enter `mysql --help` for the current values for a particular server's variables.

`--show-warnings`
: This option instructs the client not to suppress warning messages, but to display them after an SQL statement is executed in which a warning is generated, even if there was no error.

`--silent, -s`
: This option suppresses all messages except for error messages. Enter the option multiple times to further reduce the types of messages returned.

`--skip-column-names`
: This option instructs the client not to return column names in the results.

`--skip-line-numbers`
: When the client is accepting SQL statements from an input file, this option instructs the client not to display the line number of an SQL statement that has returned an error. This disables `--line-numbers`, the default.

`--skip-named-commands`
: This option disables named commands on the client. See the description of the `--named-commands` option and the following section for more information.

`--skip-pager`
: This option disables paged results on Unix types of systems. See the `--pager` option for more information.

`--skip-reconnect`
: This option instructs the client not to attempt to reconnect to the server if the connection is lost. It disables the default option `--reconnect`.

`--skip-ssl`
: This option specifies that an SSL connection should not be used, if SSL is enabled by default.

[illegible]

`--socket=`*`socket`*`, -S `*`socket`*
: This option provides the path and name of the server's socket file on Unix systems, or the named pipe on Windows systems.

`--ssl`
: This option specifies that an SSL connection should be used. It requires the server to have SSL enabled. If this option is enabled on the utility by default, use `--skip-ssl` to disable it.

`--ssl-ca=`*`pem_file`*
: This option specifies the name of the file (i.e., the *pem* file) containing a list of trusted SSL CAs.

`--ssl-capath=`*`path`*
: This option specifies the path to the trusted certificates file (i.e., the *pem* file).

`--ssl-cert=`*`filename`*
: This option specifies the name of the SSL certificate file to use for SSL connections.

`--ssl-cipher=`*`ciphers`*
: This option gives a list of ciphers that may be used for SSL encryption.

`--ssl-key=`*`filename`*
: This option specifies the SSL key file to use for secure connections.

`--ssl-verify-server-cert`
: This option verifies the client's certificate against the server's certificate for the client at startup. It is available as of version 5.1.11 of MySQL.

`--table, -t`
: This option displays results from a query in ASCII format, which is the format normally used in monitor mode. The alternative is the `--xml` option.

`--tee=`*`filename`*
: This option instructs the client to write results to the given file. You can include an absolute or relative pathname, or a simple filename. This option doesn't work in batch mode.

`--unbuffered, -n`
: This option flushes the memory buffer after each query is performed.

`--user=`*`user`*`, -u `*`user`*
: This option instructs the client to access MySQL with a username different from the current system user.

`--verbose, -v`
: This option displays more information. Use -vv or -vvv to increase verbosity.

`--version, -V`
: This option displays the version of the utility.

`--vertical`
: This option displays results in a vertical format instead of putting each row of data on a single line. This is similar to using the end of `\G` of an SQL statement in monitor mode.

`--wait, -w`
: If the client cannot connect to the server, this option tells the client to wait and retry repeatedly until it can connect.